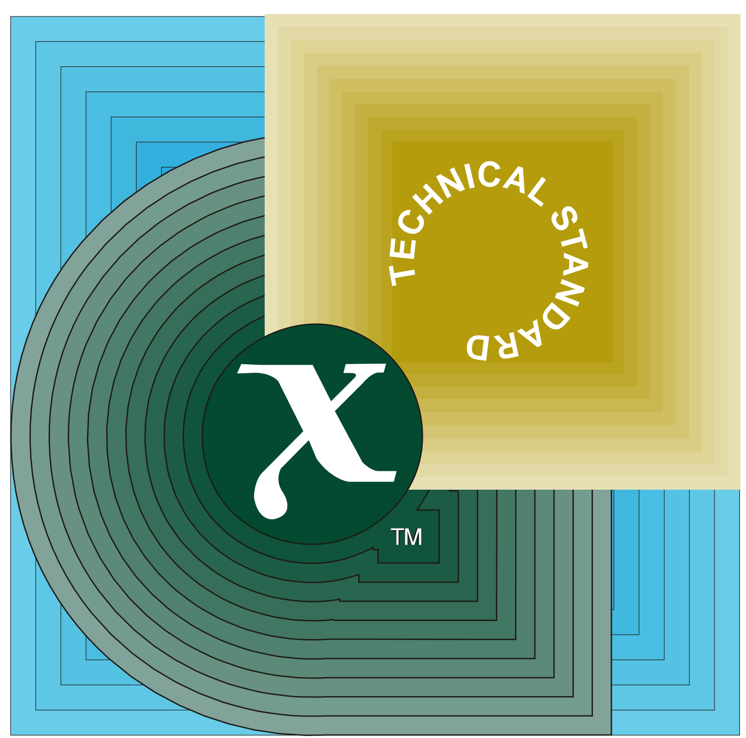
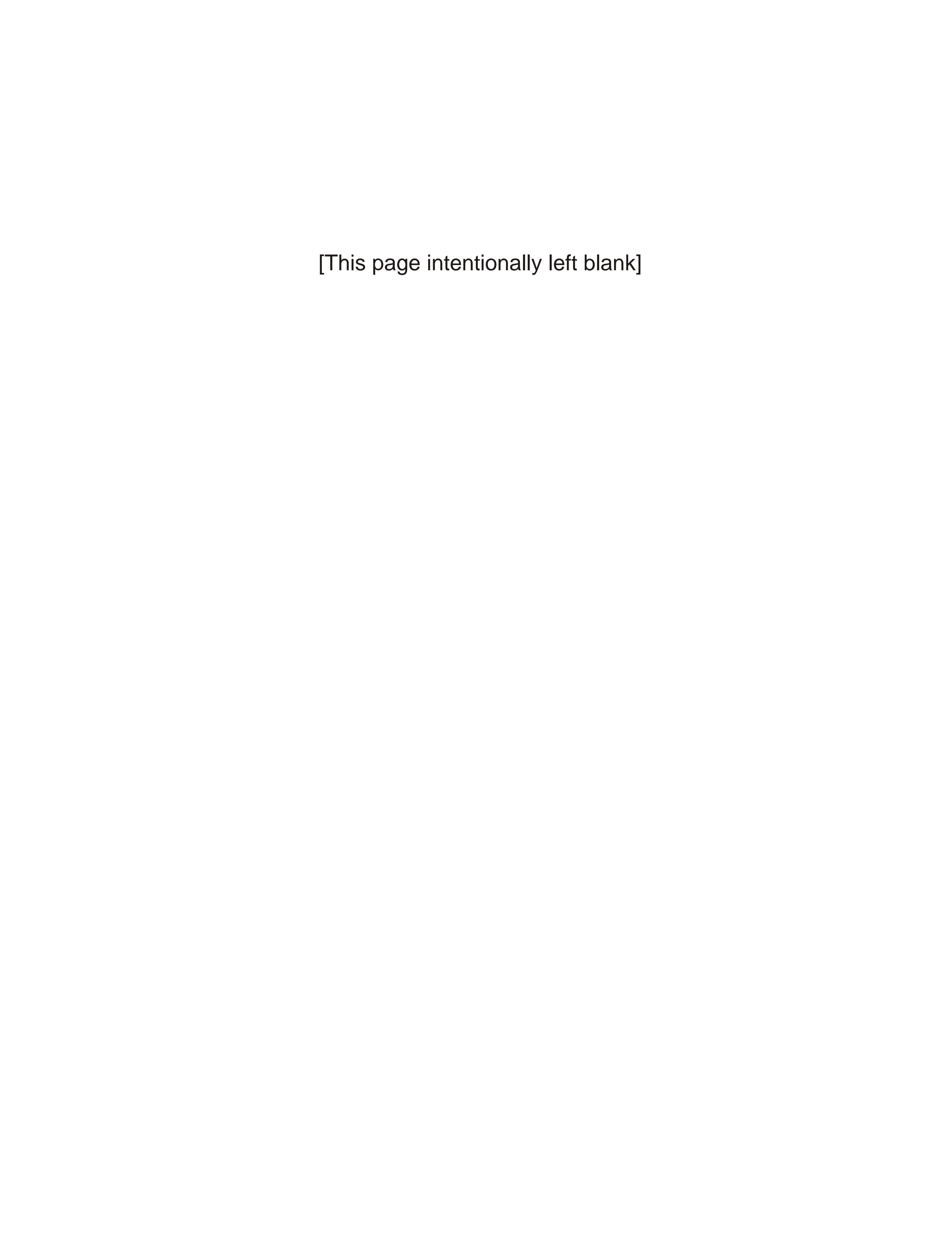
Technical Standard

Common Security: CDSA and CSSM, Version 2 (with Corrigenda)





[This page intentionally left blank]

Technical Standard

Common Security: CDSA and CSSM, Version 2.3

The Open Group

© *May 2000, The Open Group*

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Technical Standard

Common Security: CDSA and CSSM, Version 2.3

ISBN: 1-85912-202-7

Document Number: C914

Published in the U.K. by The Open Group, May 2000.

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)	12
	1.2.5.5	Authorization Computation Modules (ACs)	12
	1.2.5.6	Multi-Service Library Module	13
	1.3	Interoperability Goals	13
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	20
	2.6	Integrity Services	21
	2.6.1	CSSM-Enforced Integrity Verification	21
	2.7	Creating Checkable Components	23
	2.7.1	Verifying Components	23
	2.8	Security Context Services	24
Chapter	**3**	**Multi-Service Modules**	**27**
	3.1	Overview	27
	3.2	Application Developer View of a Multi-Service Add-In Module	28
	3.3	Service Provider View of a Multi-Service Add-In Module	28
Chapter	**4**	**Modules Control Access to Objects**	**29**
	4.1	Overview	29
	4.2	Authentication as Part of Access Control	29
	4.3	Authorization as Part of Access Control	30

	4.4	Resource Owner	31
Chapter	**5**	**System Security Services**	**33**
Part	**2**	**CSSM Core Services**	**35**
Chapter	**6**	**CSSM Core Services**	**37**
	6.1	Common Data Security Architecture	37
	6.2	Selecting CDSA Components	39
	6.3	Core Services	40
	6.3.1	Module Management Services	40
	6.3.2	Memory Management Support	42
	6.3.3	Integrity of the CSSM Environment	42
	6.3.4	CDSA and Privileges	42
	6.3.5	CDSA and USEE Privileges	43
	6.3.6	Module-Granted Use Exemptions	44
	6.3.7	Service Module Requirements if USEE Tags are Supported	45
	6.3.8	Application Privilege	45
	6.3.9	Multiple CSSM Vendors Authenticating Same Application	45
	6.4	Data Structures for Core Services	47
	6.4.1	CSSM_BOOL	47
	6.4.2	CSSM_RETURN	47
	6.4.3	CSSM_STRING	47
	6.4.4	CSSM_DATA	48
	6.4.5	CSSM_GUID	48
	6.4.6	CSSM_KEY_HIERARCHY	49
	6.4.7	CSSM_PVC_MODE	49
	6.4.8	CSSM_PRIVILEGE_SCOPE	50
	6.4.9	CSSM_VERSION	50
	6.4.10	CSSM_SUBSERVICE_UID	51
	6.4.11	CSSM_HANDLE	51
	6.4.12	CSSM_LONG_HANDLE	51
	6.4.13	CSSM_MODULE_HANDLE	51
	6.4.14	CSSM_MODULE_EVENT	52
	6.4.15	CSSM_SERVICE_MASK	52
	6.4.16	CSSM_SERVICE_TYPE	52
	6.4.17	CSSM_API_ModuleEventHandler	52
	6.4.18	CSSM_ATTACH_FLAGS	53
	6.4.19	CSSM_PRIVILEGE	53
	6.4.20	CSSM_NET_ADDRESS_TYPE	56
	6.4.21	CSSM_NET_ADDRESS	56
	6.4.22	CSSM_NET_PROTOCOL	56
	6.4.23	CSSM_CALLBACK	57
	6.4.24	CSSM_CRYPTO_DATA	57
	6.4.25	CSSM_WORDID_TYPE	58
	6.4.26	CSSM_LIST_ELEMENT_TYPE	61
	6.4.27	CSSM_LIST_TYPE	61
	6.4.28	CSSM_LIST	61

6.4.29 CSSM_LIST_ELEMENT 62
6.4.30 CSSM_TUPLE 63
6.4.31 CSSM_TUPLEGROUP 64
6.4.32 CSSM_SAMPLE_TYPE 64
6.4.33 CSSM_SAMPLE 65
6.4.34 CSSM_SAMPLEGROUP 68
6.4.35 CSSM_CHALLENGE_CALLBACK 68
6.4.36 CSSM_CERT_TYPE 70
6.4.37 CSSM_CERT_ENCODING 71
6.4.38 CSSM_ENCODED_CERT 71
6.4.39 CSSM_CERT_PARSE_FORMAT 72
6.4.40 CSSM_PARSED_CERT 72
6.4.41 CSSM_CERTPAIR 73
6.4.42 CSSM_CERTGROUP_TYPE 73
6.4.43 CSSM_CERTGROUP 74
6.4.44 CSSM_BASE_CERTS 76
6.4.45 CSSM_ACCESS_CREDENTIALS 76
6.4.46 CSSM_ACL_SUBJECT_TYPE 77
6.4.47 CSSM_ACL_AUTHORIZATION_TAG 78
6.4.48 CSSM_AUTHORIZATIONGROUP 81
6.4.49 CSSM_ACL_VALIDITY_PERIOD 81
6.4.50 CSSM_ACL_ENTRY_PROTOTYPE 81
6.4.51 CSSM_ACL_OWNER_PROTOTYPE 85
6.4.52 CSSM_ACL_SUBJECT_CALLBACK 86
6.4.53 CSSM_ACL_ENTRY_INPUT 88
6.4.54 CSSM_RESOURCE_CONTROL_CONTEXT 89
6.4.55 CSSM_ACL_HANDLE 90
6.4.56 CSSM_ACL_ENTRY_INFO 90
6.4.57 CSSM_ACL_EDIT_MODE 91
6.4.58 CSSM_ACL_EDIT 91
6.4.59 CSSM_PROC_ADDR 91
6.4.60 CSSM_KR_POLICY_TYPE 92
6.4.61 CSSM_FUNC_NAME_ADDR 92
6.4.62 CSSM_MEMORY_FUNCS and CSSM_API_MEMORY_FUNCS.. 92
6.5 Common Error Return Codes 94
6.5.1 Error Values Derived from Common Error Codes 94
6.5.2 CSSM Module-Specific Error Values 95
6.6 Core Functions 97
CSSM_Init 98
CSSM_Terminate 102
6.7 Module Management Functions 103
CSSM_ModuleLoad 104
CSSM_ModuleUnload 105
CSSM_Introduce 106
CSSM_Unintroduce 107
CSSM_ModuleAttach 108
CSSM_ModuleDetach 110
CSSM_SetPrivilege 111

		CSSM_GetPrivilege	113
		CSSM_GetModuleGUIDFromHandle	114
		CSSM_GetSubserviceUIDFromHandle	115
	6.8	EMM Module Management Functions	116
		CSSM_ListAttachedModuleManagers	117
	6.9	Utility Functions	118
		CSSM_GetAPIMemoryFunctions	119
Part	**3**	**Cryptographic Service Providers (CSP)**	**121**
Chapter	**7**	**Cryptographic Services**	**123**
	7.1	Cryptographic Service Providers	123
	7.2	CDSA Add-In Modules	124
	7.3	CDSA CSP Operation	125
	7.3.1	CSSM Infrastructure	125
	7.3.2	CSP Form Factor	126
	7.3.3	Legacy CSPs	126
	7.3.4	CSP Registration	127
	7.3.5	Cryptographic Services Operations	127
	7.3.6	Key Management	128
	7.3.7	Key Formats for Public Key-Based Algorithms	129
	7.3.8	Buffer Management for Cryptographic Services	130
	7.4	Data Structures	131
	7.4.1	CSSM_CC_HANDLE	131
	7.4.2	CSSM_CSP_HANDLE	131
	7.4.3	CSSM_DATE	131
	7.4.4	CSSM_RANGE	131
	7.4.5	CSSM_QUERY_SIZE_DATA	132
	7.4.6	CSSM_HEADERVERSION	132
	7.4.7	CSSM_KEY_SIZE	132
	7.4.8	CSSM_KEYBLOB_TYPE	133
	7.4.9	CSSM_KEYBLOB_FORMAT	133
	7.4.10	CSSM_KEYCLASS	134
	7.4.11	CSSM_KEYATTR_FLAGS	134
	7.4.12	CSSM_KEYUSE	134
	7.4.13	CSSM_KEYHEADER	134
	7.4.14	CSSM_KEY	139
	7.4.15	CSSM_WRAP_KEY	139
	7.4.16	CSSM_CSP_TYPE	139
	7.4.17	CSSM_CONTEXT_TYPE	140
	7.4.18	CSSM Algorithms	140
	7.4.19	CSSM_ATTRIBUTE_TYPE	143
	7.4.20	CSSM_ENCRYPT_MODE	144
	7.4.21	CSSM_PADDING	145
	7.4.22	CSSM_KEY_TYPE	145
	7.4.23	CSSM_CONTEXT_ATTRIBUTE	145
	7.4.24	CSSM_CONTEXT	147
	7.4.25	CSSM_SC_FLAGS	153

7.4.26 CSSM_CSP_READER_FLAGS ... 153
7.4.27 CSSM_CSP_FLAGS ... 153
7.4.28 CSSM_PKCS_OAEP ... 154
7.4.29 CSSM_PKCS_OAEP_PARAMS ... 155
7.4.30 CSSM_CSP_OPERATIONAL_STATISTICS ... 155
7.4.31 CSSM_PKCS5_PBKDF1_PARAMS ... 156
7.4.32 CSSM_PKCS5_PBKDF2_PRF ... 157
7.4.33 CSSM_PKCS5_PBKDF2_PARAMS ... 157
7.4.34 CSSM_KEA_DERIVE_PARAMS ... 157
7.5 Error Codes and Error Values ... 158
7.5.1 CSP Error Values Derived from Common Error Codes ... 158
7.5.2 General CSP Error Values ... 160
7.5.3 CSP Key Error Values ... 160
7.5.4 CSP Vector of Buffers Error Values ... 162
7.5.5 CSP Cryptographic Context Error Values ... 162
7.5.6 CSP Staged Cryptographic API Error Values ... 165
7.5.7 Other CSP Error Values ... 165
7.6 Cryptographic Context Operations ... 167
CSSM_CSP_CreateSignatureContext ... 168
CSSM_CSP_CreateSymmetricContext ... 170
CSSM_CSP_CreateDigestContext ... 172
CSSM_CSP_CreateMacContext ... 173
CSSM_CSP_CreateRandomGenContext ... 174
CSSM_CSP_CreateAsymmetricContext ... 175
CSSM_CSP_CreateDeriveKeyContext ... 177
CSSM_CSP_CreateKeyGenContext ... 179
CSSM_CSP_CreatePassThroughContext ... 181
CSSM_GetContext ... 182
CSSM_FreeContext ... 183
CSSM_SetContext ... 184
CSSM_DeleteContext ... 185
CSSM_GetContextAttribute ... 186
CSSM_UpdateContextAttributes ... 187
CSSM_DeleteContextAttributes ... 188
7.7 Cryptographic Sessions and Controlled Access to Keys ... 189
CSSM_CSP_Login ... 190
CSSM_CSP_Logout ... 191
CSSM_CSP_GetLoginAcl ... 192
CSSM_CSP_ChangeLoginAcl ... 194
CSSM_GetKeyAcl ... 196
CSSM_ChangeKeyAcl ... 198
CSSM_GetKeyOwner ... 200
CSSM_ChangeKeyOwner ... 201
CSSM_CSP_GetLoginOwner ... 202
CSSM_CSP_ChangeLoginOwner ... 203
7.8 Cryptographic Operations ... 204
SignData ... 205
SignDataInit ... 207

SignDataUpdate 208
SignDataFinal 209
VerifyData 211
VerifyDataInit 213
VerifyDataUpdate 214
VerifyDataFinal 216
DigestData 217
DigestDataInit 219
DigestDataUpdate 220
DigestDataClone 222
DigestDataFinal 224
GenerateMac 226
GenerateMacInit 228
GenerateMacUpdate 229
GenerateMacFinal 231
VerifyMac 233
VerifyMacInit 235
VerifyMacUpdate 236
VerifyMacFinal 238
QuerySize 239
EncryptData 241
EncryptDataP 244
EncryptDataInit 245
EncryptDataInitP 247
EncryptDataUpdate 248
EncryptDataFinal 250
DecryptData 252
DecryptDataP 255
DecryptDataInit 256
DecryptDataInitP 258
DecryptDataUpdate 259
DecryptDataFinal 261
QueryKeySizeInBits 263
GenerateKey 265
GenerateKeyP 268
GenerateKeyPair 269
GenerateKeyPairP 272
GenerateRandom 273
ObtainPrivateKeyFromPublicKey 275
WrapKey 276
WrapKeyP 278
UnwrapKey 279
UnwrapKeyP 282
DeriveKey 283
FreeKey 286
GenerateAlgorithmParams 288
7.9 Miscellaneous Functions 290
GetOperationalStatistics 291

		GetTimeValue	292
		RetrieveUniqueId	294
		RetrieveCounter	295
		VerifyDevice	296
	7.10	Extensibility Function	297
		PassThrough	298
	7.11	Module Management Function	299
		CSP_EventNotify	300
Part	**4**	**Trust Policy (TP) Services**	**303**
Chapter	**8**	**Trust Policy Services API**	**305**
	8.1	Overview	305
	8.1.1	Digital Certificate	305
	8.1.2	Trust Model	305
	8.1.3	Trust Services	307
	8.2	CDSA TP Features	309
	8.3	SPI TP	310
	8.3.1	Add-In Module	310
	8.3.2	Operations	310
	8.4	Data Structures	312
	8.4.1	CSSM_TP_HANDLE	312
	8.4.2	CSSM_TP_AUTHORITY_ID	312
	8.4.3	CSSM_TP_AUTHORITY_REQUEST_TYPE	312
	8.4.4	CSSM_TP_VERIFICATION_RESULTS_CALLBACK	313
	8.4.5	CSSM_TP_POLICYINFO	313
	8.4.6	CSSM_TP_SERVICES	314
	8.4.7	CSSM_TP_ACTION	314
	8.4.8	CSSM_TP_STOP_ON	314
	8.4.9	CSSM_TIMESTRING	314
	8.4.10	CSSM_TP_CALLERAUTH_CONTEXT	315
	8.4.11	CSSM_CRL_PARSE_FORMAT	316
	8.4.12	CSSM_PARSED_CRL	317
	8.4.13	CSSM_CRL_PAIR	317
	8.4.14	CSSM_CRLGROUP_TYPE	318
	8.4.15	CSSM_CRLGROUP	318
	8.4.16	CSSM_FIELDGROUP	320
	8.4.17	CSSM_EVIDENCE_FORM	320
	8.4.18	CSSM_EVIDENCE	320
	8.4.19	CSSM_TP_VERIFY_CONTEXT	321
	8.4.20	CSSM_TP_VERIFY_CONTEXT_RESULT	322
	8.4.21	CSSM_TP_REQUEST_SET	322
	8.4.22	CSSM_TP_RESULT_SET	322
	8.4.23	CSSM_TP_CONFIRM_STATUS	323
	8.4.24	CSSM_TP_CONFIRM_RESPONSE	323
	8.4.25	CSSM_ESTIMATED_TIME_UNKNOWN	324
	8.4.26	CSSM_ELAPSED_TIME_UNKNOWN	324
	8.4.27	CSSM_ELAPSED_TIME_COMPLETE	324

8.4.28 CSSM_TP_CERTISSUE_INPUT 324
8.4.29 CSSM_TP_CERTISSUE_STATUS 325
8.4.30 CSSM_TP_CERTISSUE_OUTPUT 326
8.4.31 CSSM_TP_CERTCHANGE_ACTION 326
8.4.32 CSSM_TP_CERTCHANGE_REASON 327
8.4.33 CSSM_TP_CERTCHANGE_INPUT 328
8.4.34 CSSM_TP_CERTCHANGE_STATUS 329
8.4.35 CSSM_TP_CERTCHANGE_OUTPUT 329
8.4.36 CSSM_TP_CERTVERIFY_INPUT 330
8.4.37 CSSM_TP_CERTVERIFY_STATUS 330
8.4.38 CSSM_TP_CERTVERIFY_OUTPUT 331
8.4.39 CSSM_TP_CERTNOTARIZE_INPUT 331
8.4.40 CSSM_TP_CERTNOTARIZE_STATUS 333
8.4.41 CSSM_TP_CERTNOTARIZE_OUTPUT 333
8.4.42 CSSM_TP_CERTRECLAIM_INPUT 334
8.4.43 CSSM_TP_CERTRECLAIM_STATUS 335
8.4.44 CSSM_TP_CERTRECLAIM_OUTPUT 335
8.4.45 CSSM_TP_CRLISSUE_INPUT 336
8.4.46 CSSM_TP_CRLISSUE_STATUS 337
8.4.47 CSSM_TP_CRLISSUE_OUTPUT 337
8.4.48 CSSM_TP_FORM_TYPE 338
8.5 Error Codes and Error Values 339
8.5.1 TP Error Values Derived from Common Error Codes 339
8.5.2 Common TP Error Values 340
8.6 Trust Policy Operations 344
TP_SubmitCredRequest 345
CSSM_TP_RetrieveCredResult 348
TP_ConfirmCredResult 351
TP_ReceiveConfirmation 354
TP_CertReclaimKey 356
TP_CertReclaimAbort 358
TP_FormRequest 359
TP_FormSubmit 361
8.7 Local Application-Domain-Specific Trust Policy Functions 363
TP_CertGroupVerify 364
TP_CertCreateTemplate 367
TP_CertGetAllTemplateFields 369
TP_CertSign 371
TP_CrlVerify 374
TP_CrlCreateTemplate 377
TP_CertRevoke 379
TP_CertRemoveFromCrlTemplate 382
TP_CrlSign 385
TP_ApplyCrlToDb 388
8.8 Group Functions 391
TP_CertGroupConstruct 392
TP_CertGroupPrune 395
TP_CertGroupToTupleGroup 397

		TP_TupleGroupToCertGroup	399
	8.9	Extensibility Functions	401
		TP_PassThrough	402
Part	**5**	**Authorization Computation (AC) Services**	**405**
Chapter	**9**	**Authorization Computation Services**	**407**
	9.1	Overview	407
	9.2	Authorization, Certificates, and Credentials	407
	9.2.1	Classes of Certificates and Other Credentials	407
	9.2.2	Credential Format Options	408
	9.2.3	Logic of Authorization	410
	9.2.4	Authorization Reduction Process	411
	9.2.5	Example Authorization Request	412
	9.3	Add-In Module	413
	9.4	Data Structures	414
	9.4.1	CSSM_AC_HANDLE	414
	9.5	Error Codes and Error Values	414
	9.5.1	AC Error Values Derived from Common Error Codes	414
	9.5.2	AC Error Values	415
	9.6	Authorization Computation Operations	416
		AC_AuthCompute	417
	9.7	Extensibility Functions	421
		AC_PassThrough	422
Part	**6**	**Certificate Library (CL) Services**	**425**
Chapter	**10**	**Certificate Library Services**	**427**
	10.1	Overview	427
	10.1.1	Certificates and CRLs	427
	10.1.2	Application and Certificate Library Interaction	427
	10.1.3	Operations on Certificates	428
	10.1.4	Add-In Module	429
	10.1.5	Certificate Life Cycle	430
	10.2	Data Structures	431
	10.2.1	CSSM_CL_HANDLE	431
	10.2.2	CSSM_CL_TEMPLATE_TYPE	432
	10.2.3	CSSM_CERT_BUNDLE_TYPE	432
	10.2.4	CSSM_CERT_BUNDLE_ENCODING	433
	10.2.5	CSSM_CERT_BUNDLE_HEADER	433
	10.2.6	CSSM_CERT_BUNDLE	433
	10.2.7	CSSM_OID	434
	10.2.8	CSSM_CRL_TYPE	434
	10.2.9	CSSM_CRL_ENCODING	434
	10.2.10	CSSM_ENCODED_CRL	435
	10.2.11	CSSM_FIELD	435
	10.2.12	CSSM_FIELDVALUE_COMPLEX_DATA_TYPE	435

10.3 Error Codes and Error Values 436
10.3.1 CL Error Values Derived from Common Error Codes 436
10.3.2 CL Error Values 437
10.4 Certificate Operations 438
CL_CertCreateTemplate 439
CL_CertGetAllTemplateFields 441
CL_CertSign 442
CL_CertVerify 444
CL_CertVerifyWithKey 446
CL_CertGetFirstFieldValue 448
CL_CertGetNextFieldValue 450
CL_CertAbortQuery 452
CL_CertGetKeyInfo 454
CL_CertGetAllFields 455
CL_FreeFields 457
CL_FreeFieldValue 458
CL_CertCache 459
CL_CertGetFirstCachedFieldValue 461
CL_CertGetNextCachedFieldValue 463
CL_CertAbortCache 465
CL_CertGroupToSignedBundle 466
CL_CertGroupFromVerifiedBundle 468
CL_CertDescribeFormat 470
10.5 Certificate Revocation List Operations 471
CL_CrlCreateTemplate 472
CL_CrlSetFields 474
CL_CrlAddCert 476
CL_CrlRemoveCert 478
CL_CrlSign 480
CL_CrlVerify 482
CL_CrlVerifyWithKey 484
CL_IsCertInCrl 486
CL_CrlGetFirstFieldValue 487
CL_CrlGetNextFieldValue 489
CL_CrlAbortQuery 491
CL_CrlGetAllFields 492
CL_CrlCache 494
CL_IsCertInCachedCrl 496
CL_CrlGetFirstCachedFieldValue 498
CL_CrlGetNextCachedFieldValue 501
CL_CrlGetAllCachedRecordFields 503
CL_CrlAbortCache 505
CL_CrlDescribeFormat 506
10.6 Extensibility Functions 507
CL_PassThrough 508

Part	**7**	**Data Storage Library (DL) Services**	**511**
Chapter	**11**	**Data Storage Library Services**	**513**
	11.1	Introduction	513
	11.2	CSSM API	513
	11.3	DL SPI	514
	11.3.1	Add-In Module	514
	11.3.2	Operation	515
	11.4	Interoperability	516
	11.5	Categories of Operations	516
	11.6	Data Storage Data Structures	518
	11.6.1	CSSM_DL_HANDLE	518
	11.6.2	CSSM_DB_HANDLE	518
	11.6.3	CSSM_DL_DB_HANDLE	518
	11.6.4	CSSM_DL_DB_LIST	518
	11.6.5	CSSM_DB_ATTRIBUTE_NAME_FORMAT	519
	11.6.6	CSSM_DB_ATTRIBUTE_FORMAT	519
	11.6.7	CSSM_DB_ATTRIBUTE_INFO	520
	11.6.8	CSSM_DB_ATTRIBUTE_DATA	521
	11.6.9	CSSM_DB_RECORDTYPE	521
	11.6.10	CSSM_DB_CERTRECORD_SEMANTICS	527
	11.6.11	CSSM_DB_RECORD_ATTRIBUTE_INFO	528
	11.6.12	CSSM_DB_RECORD_ATTRIBUTE_DATA	528
	11.6.13	CSSM_DB_PARSING_MODULE_INFO	529
	11.6.14	CSSM_DB_INDEX_TYPE	529
	11.6.15	CSSM_DB_INDEXED_DATA_LOCATION	530
	11.6.16	CSSM_DB_INDEX_INFO	530
	11.6.17	CSSM_DB_UNIQUE_RECORD	530
	11.6.18	CSSM_DB_RECORD_INDEX_INFO	531
	11.6.19	CSSM_DB_ACCESS_TYPE	531
	11.6.20	CSSM_DB_MODIFY_MODE	531
	11.6.21	CSSM_DBINFO	532
	11.6.22	CSSM_DB_OPERATOR	533
	11.6.23	CSSM_DB_CONJUNCTIVE	533
	11.6.24	CSSM_SELECTION_PREDICATE	534
	11.6.25	CSSM_QUERY_LIMITS	534
	11.6.26	CSSM_QUERY_FLAGS	535
	11.6.27	CSSM_QUERY	535
	11.6.28	CSSM_DLTYPE	536
	11.6.29	CSSM_DL_PKCS11_ATTRIBUTES	536
	11.6.30	CSSM_DB_DATASTORES_UNKNOWN	536
	11.6.31	CSSM_NAME_LIST	537
	11.6.32	CSSM_DB_RETRIEVAL_MODES	537
	11.6.33	CSSM_DB_SCHEMA_ATTRIBUTE_INFO	537
	11.6.34	CSSM_DB_SCHEMA_INDEX_INFO	538
	11.7	Error Codes and Error Values	539
	11.7.1	DL Error Values Derived from Common Error Codes	539
	11.7.2	DL Error Values Derived from ACL-based Error Codes	539

	11.7.3	DL Error Values for Specific Data Types	540
	11.7.4	General DL Error Values	540
	11.7.5	DL Specific Error Values	541
	11.8	Data Storage Library Operations	544
		DL_Authenticate	545
		DL_GetDbAcl	547
		DL_ChangeDbAcl	549
		DL_GetDbOwner	551
		DL_ChangeDbOwner	552
	11.9	Data Storage Operations	553
		DL_DbOpen	554
		DL_DbClose	556
		DL_DbCreate	557
		DL_DbDelete	560
		DL_CreateRelation	562
		DL_DestroyRelation	564
		DL_GetDbNames	565
		DL_GetDbNameFromHandle	566
		DL_FreeNameList	567
	11.10	Data Record Operations	568
		DL_DataInsert	569
		DL_DataDelete	571
		DL_DataModify	572
		DL_DataGetFirst	575
		DL_DataGetNext	578
		DL_DataAbortQuery	580
		DL_DataGetFromUniqueRecordId	581
		DL_FreeUniqueRecord	583
	11.11	Extensibility Operations	584
		DL_PassThrough	585
Part	**8**	**Module Directory Service (MDS)**	**587**
Chapter	**12**	**Introduction**	**589**
	12.1	Common Data Security Architecture	589
	12.2	MDS in CDSA	591
	12.3	MDS Installation and Access	592
	12.4	Using MDS in Integrity Verification Protocols	592
	12.5	Multi-User Access Model	593
	12.6	API Overview	593
Chapter	**13**	**MDS Schema Definition**	**595**
	13.1	Object Directory Database and the Object Relation	595
	13.2	CDSA Directory Database	597
	13.3	CSSM Relation	598
	13.4	KRMM Relation	599
	13.5	EMM Relation	600
	13.6	Primary EMM Service Provider Relation	601

13.7 Common Relation ... 602
13.8 CSP Primary Relation ... 603
13.9 CSP Capabilities Relation ... 604
13.10 CSP Encapsulated Products Relation ... 606
13.11 CSP SmartcardInfo Relation ... 607
13.12 DL Primary Relation ... 608
13.13 DL Encapsulated Products Relation ... 609
13.14 CL Primary Relation ... 610
13.15 CL Encapsulated Products Relation ... 612
13.16 TP Primary Relation ... 613
13.17 TP Policy-OIDS Relation ... 614
13.18 TP Encapsulated Products Relation ... 615
13.19 MDS Schema Relation ... 617
13.20 AC Primary Relation ... 619
13.21 KR Primary Relation ... 620

Chapter 14 MDS Name Space and Directory Structures ... 621
14.1 MDS Name Space ... 621
14.2 Object Directory ... 621
14.3 CDSA Directory ... 621
14.3.1 CDSA Relation Attributes ... 622
14.4 MDS Meta-Data Names ... 623
14.5 Data Structure ... 625
14.5.1 MDS_HANDLE ... 625
14.5.2 MDS_DB_HANDLE ... 625
14.5.3 MDS_FUNC ... 625

Chapter 15 Module Directory Services APIs ... 627
15.1 MDS Context APIs ... 628
MDS_Initialize ... 629
MDS_Terminate ... 631
15.2 MDS Installation APIs ... 632
MDS_Install ... 633
MDS_Uninstall ... 634
15.3 MDS Database Service APIs ... 635
15.4 Write-Access to MDS Databases ... 635
15.4.1 Updating MDS Schema ... 635
15.4.2 Updating MDS Databases ... 635
15.4.3 Manifest Attributes for MDS Access Control Privileges ... 636

Chapter 16 MDS Administration ... 637
16.1 MDS Installation ... 637
16.2 General Access Control over MDS Databases ... 637
16.2.1 Privileged Application ... 638
16.2.2 File Permissions ... 638
16.2.3 Administrator ACLs ... 638

Part	**9**	**Key Recovery (KR) Services**	**639**
Chapter	**17**	**Overview**	**641**
	17.1	Introduction	641
	17.2	Key Recovery Nomenclature	641
	17.2.1	Key Recovery Types	641
	17.2.2	Key Recovery Phases	643
	17.2.3	Lifetime of Key Recovery Fields	644
	17.2.4	Key Recovery Policy	644
	17.3	Key Recovery in the Common Data Security Architecture	645
Chapter	**18**	**Key Recovery Enablement**	**647**
	18.1	Key Recovery in the CDSA	647
	18.2	Functionality Definition	647
	18.3	Extensions to the Cryptographic Module Manager	648
	18.4	Key Recovery Module Manager	649
	18.4.1	Operational Scenarios for Key Recovery	649
	18.4.2	Key Recovery Profiles	651
	18.4.3	Key Recovery Context	652
	18.4.4	Key Recovery Policy	652
	18.4.5	Key Recovery Enablement Operations	653
	18.4.6	Key Recovery Registration and Request Operations	653
Chapter	**19**	**Key Recovery Interfaces**	**655**
	19.1	Summary of Interface Calls	655
	19.1.1	Module Management Operations	655
	19.1.2	Key Recovery Module Management Operations	655
	19.1.3	Key Recovery Context Operations	655
	19.1.4	Key Recovery Registration Operations	656
	19.1.5	Key Recovery Enablement Operations	656
	19.1.6	Key Recovery Request Operations	656
	19.1.7	Privileged Context Function	657
	19.1.8	Extensibility Function	657
	19.2	Example Application Using Key Recovery APIs	658
	19.3	Data Structures	661
	19.3.1	CSSM_KRSP_HANDLE	661
	19.3.2	CSSM_KR_NAME	661
	19.3.3	CSSM_KR_PROFILE	661
	19.3.4	CSSM_ATTRIBUTE_TYPE Additions	663
	19.3.5	CSSM_KR_POLICY_FLAGS	663
	19.3.6	CSSM_KR_POLICY_LIST_ITEM	663
	19.3.7	CSSM_KR_POLICY_INFO	664
	19.4	Key Recovery MDS Relation	665
	19.4.1	Generic Module Management Operations	665
	19.5	Key Recovery Module Management Operations	666
		CSSM_KR_SetEnterpriseRecoveryPolicy	667
	19.6	Key Recovery Context Operations	668
		CSSM_KR_CreateRecoveryRegistrationContext	669

		CSSM_KR_CreateRecoveryEnablementContext	670
		CSSM_KR_CreateRecoveryRequestContext	671
		CSSM_KR_GetPolicyInfo	672
	19.7	Key Recovery Registration Operations	673
		KR_RegistrationRequest	674
		KR_RegistrationRetrieve	677
	19.8	Key Recovery Enablement Operations	679
		KR_GenerateRecoveryFields	680
		KR_ProcessRecoveryFields	682
	19.9	Key Recovery Request Operations	684
		KR_RecoveryRequest	685
		KR_RecoveryRetrieve	687
		KR_GetRecoveredObject	689
		KR_RecoveryRequestAbort	692
		CSSM_KR_QueryPolicyInfo	693
		CSSM_KR_FreePolicyInfo	695
	19.10	Privileged Context Operation	696
		KRSP_PassPrivFunc	697
	19.11	Extensibility Function	698
		KR_PassThrough	699
Part	**10**	**Embedded Integrity Services Library (EISL)**	**703**
Chapter	**20**	**Introduction**	**705**
	20.1	Problem Statement	705
	20.2	Extending Trust	705
	20.3	Why an Embedded Library?	706
	20.4	A Phased Approach	706
	20.4.1	Phase I. Establishing a Foothold: Self-Check	706
	20.4.2	Phase II. Finding our Friends: Bilateral Authentication	707
	20.4.3	Phase III. Secure Linkage Check	707
	20.5	Using Library Services	707
	20.5.1	Location of Modules and Credentials	707
	20.5.2	Verification of Modules and their Credentials	708
	20.5.3	Secure Linkage	708
	20.5.4	Integrity Credentials	708
	20.6	Use of Other Standards or Specifications	709
Chapter	**21**	**Data Structures**	**711**
	21.1	Object Pointers	711
	21.1.1	Iterator Objects	711
	21.1.2	Verified Signature Root Object	711
	21.1.3	Verified Certificate Chain Object	712
	21.1.4	Verified Certificate Object	712
	21.1.5	Manifest Section Object	712
	21.1.6	Verified Module Object	712
	21.1.7	EISL Object Relationships and Life Cycle	713
	21.2	Types and Data Structure	714

21.2.1 ISL_DATA 714
21.2.2 ISL_CONST_DATA 714
21.2.3 ISL_STATUS 715
21.2.4 ISL_FUNCTION_PTR 715

Chapter 22 EISL Functions 717

22.1 Credential and Attribute Verification Services 717
EISL_SelfCheck 718
EISL_VerifyAndLoadModuleAndCredentialData 719
EISL_VerifyAndLoadModuleAndCredDataWithCert 721
EISL_VerifyAndLoadModuleAndCredentials 723
EISL_VerifyAndLoadModuleAndCredentialsWithCert 725
EISL_VerifyLoadedModuleAndCredentialData 727
EISL_VerifyLoadedModuleAndCredDataWithCert 729
EISL_VerifyLoadedModuleAndCredentials 731
EISL_VerifyLoadedModuleAndCredentialsWithCert 733
EISL_GetCertficateChain 735
EISL_ContinueVerification 736
EISL_DuplicateVerifiedModulePtr 738
EISL_RecycleVerifiedModuleCredentials 739

22.2 Signature Root Methods 740
EISL_CreateVerifiedSignatureRootWithCredentialData 741
EISL_CreateVerifiedSigRootWithCredDataAndCert 743
EISL_CreateVerifiedSignatureRoot 744
EISL_CreateVerifiedSignatureRootWithCertificate 745
EISL_FindManifestSection 746
EISL_CreateManifestSectionEnumerator 747
EISL_GetNextManifestSection 748
EISL_RecycleManifestSectionEnumerator 749
EISL_FindManifestAttribute 750
EISL_CreateManifestAttributeEnumerator 751
EISL_FindSignerInfoAttribute 752
EISL_CreateSignerInfoAttributeEnumerator 753
EISL_GetNextAttribute 754
EISL_RecycleAttributeEnumerator 755
EISL_FindSignatureAttribute 756
EISL_CreateSignatureAttributeEnumerator 757
EISL_GetNextSignatureAttribute 758
EISL_RecycleSignatureAttributeEnumerator 759
EISL_RecycleVerifiedSignatureRoot 760

22.3 Certificate Chain Methods 761
EISL_CreateCertificateChainWithCredentialData 762
EISL_CreateCertificateChainWithCredDataAndCert 763
EISL_CreateCertificateChain 764
EISL_CreateCertificateChainWithCertificate 765
EISL_CopyCertificateChain 766
EISL_RecycleVerifiedCertificateChain 767

22.4 Certificate Attribute Methods 768

		EISL_FindCertificateAttribute	769
		EISL_CreateCertificateAttributeEnumerator	770
		EISL_GetNextCertificateAttribute	771
		EISL_RecycleCertificateAttributeEnumerator	772
	22.5	Manifest Section Object Methods	773
		EISL_GetManifestSignatureRoot	774
		EISL_VerifyAndLoadModule	775
		EISL_VerifyLoadedModule	776
		EISL_FindManifestSectionAttribute	777
		EISL_CreateManifestSectionAttributeEnumerator	778
		EISL_GetNextManifestSectionAttribute	779
		EISL_RecycleManifestSectionAttributeEnumerator	780
		EISL_GetModuleManifestSection	781
	22.6	Secure Linkage Services	782
		EISL_LocateProcedureAddress	783
		EISL_GetReturnAddress	785
		EISL_CheckAddressWithinModule	786
		EISL_CheckDataAddressWithinModule	787
		EISL_GetLibHandle	788
Part	**11**	**Signed Manifest**	**789**
Chapter	**23**	**Introduction**	**791**
	23.1	Signed Manifests	791
	23.2	Common Data Security Architecture	791
Chapter	**24**	**Signed Manifests—Requirements**	**793**
Chapter	**25**	**Signed Manifests—The Architecture**	**795**
Chapter	**26**	**Format Specification**	**799**
	26.1	The Manifest	799
	26.1.1	Manifest Header Specification	799
	26.1.2	Manifest Sections	799
	26.1.3	Format Specification	800
	26.1.4	MAGIC—A Flagging Mechanism	801
	26.1.5	Metadata	801
	26.1.6	Ordering Metadata Values	801
	26.1.7	Manifest Examples	802
	26.2	Signer Information	803
	26.2.1	Signing Information Header	803
	26.2.2	Signer Information Sections	803
	26.2.3	Signing Information Examples	803
	26.3	Signature Blocks	804

Chapter	**27**	**Signed Manifests—Verifying Signatures**	**805**
	27.1	Verifying the Manifest	805
	27.2	Verifying Referents in the Manifest	805
Chapter	**28**	**File-Based Representation of Signed Manifests**	**807**
	28.1	Description	807
	28.2	Representation Constraints	807
Chapter	**29**	**Signed Manifests—Examples**	**809**
	29.1	Static Referent Objects	809
	29.2	Dynamic Referent Objects with Verified Source	809
	29.2.1	Stock Quote Service	810
	29.3	Embedded or Nested Referent Objects	810
	29.3.1	Signed Objects Whose Signatures Serve to Carry the Object	810
	29.3.2	Signed Objects Whose Signature Blocks are Embedded	811
	29.3.3	Nested Manifests	811
	29.3.4	Signed Portion of an HTML Page	815
	29.3.5	Foreign Language Support—Multiple Hash Values	815
	29.3.6	Dynamic Sources with no Associated Data	815
	29.3.7	Resources that Transform Locations	816
Part	**12**	**OIDs for Certificate Library Modules**	**817**
Chapter	**30**	**Introduction**	**819**
Chapter	**31**	**OIDs for X.509 Certificate Library Modules**	**821**
	31.1	Overview	821
	31.2	Interoperable Format Specifications for X.509	821
	31.2.1	Certificate Library Service Provider X.509 Field OIDs	821
	31.2.2	Base of the Object Identifier Name Space	822
	31.2.3	Programmatic Definition of Base Object Identifiers	823
	31.2.4	Terminology	823
	31.3	Object Identifiers for X.509 V3 Certificates	824
	31.3.1	Base Object Identifiers	824
	31.3.2	Programmatic Definition of Base Object Identifiers	824
	31.3.3	Object Identifiers for Fields	825
	31.3.4	Certificate OID Definition	825
	31.3.5	Signature OID Definition	826
	31.3.6	Extension OID Definition	827
	31.4	C Language Data Structures	828
	31.4.1	CSSM_BER_TAG	828
	31.4.2	CSSM_X509_ALGORITHM_IDENTIFIER	829
	31.4.3	CSSM_X509_TYPE_VALUE_PAIR	829
	31.4.4	CSSM_X509_RDN	829
	31.4.5	CSSM_X509_NAME	830
	31.4.6	CSSM_X509_SUBJECT_PUBLIC_KEY_INFO	830
	31.4.7	CSSM_X509_TIME	830
	31.4.8	CSSM_X509_VALIDITY	831

	31.4.9	CSSM_X509_OPTION	831
	31.4.10	CSSM_X509EXT_BASICCONSTRAINTS	831
	31.4.11	CSSM_X509EXT_DATA_FORMAT	832
	31.4.12	CSSM_X509EXT_TAGandVALUE	832
	31.4.13	CSSM_X509EXT_PAIR	833
	31.4.14	CSSM_X509_EXTENSION	833
	31.4.15	CSSM_X509_EXTENSIONS	834
	31.4.16	CSSM_X509_TBS_CERTIFICATE	834
	31.4.17	CSSM_X509_SIGNATURE	835
	31.4.18	CSSM_X509_SIGNED_CERTIFICATE	835
	31.4.19	CSSM_X509EXT_POLICYQUALIFIERINFO	836
	31.4.20	CSSM_X509EXT_POLICYQUALIFIERS	836
	31.4.21	CSSM_X509EXT_POLICYINFO	836
	31.5	Certificate OIDs and Certificate Data Structures	837
Chapter	**32**	**OIDs for X.509 Certificate Revocation Lists**	**839**
	32.1	Base Object Identifiers	839
	32.2	Programmatic Definition of Base Object Identifiers	839
	32.3	Object Identifiers for Fields	839
	32.3.1	CRL OIDs	839
	32.3.2	CRL Entry (CRL CertList) OIDs	840
	32.3.3	CRL Entry (CRL CertList) Extension OIDs	840
	32.3.4	CRL Extension OIDs	841
	32.4	C Language Data Structures for X.509 CRLs	842
	32.4.1	CSSM_X509_REVOKED_CERT_ENTRY	842
	32.4.2	CSSM_X509_REVOKED_CERT_LIST	842
	32.4.3	CSSM_X509_TBS_CERTLIST	842
	32.4.4	CSSM_X509_SIGNED_CRL	843
	32.5	Associating CRL OIDs and CRL Data Structures	844
Part	**13**	**CSSM Elective Module Manager (EMM)**	**847**
Chapter	**33**	**Introduction**	**849**
Chapter	**34**	**Overview of Elective Module Managers**	**851**
	34.1	Transparent, Dynamic Attach	851
	34.2	Registering Module Managers	853
	34.3	Interaction with CSSM	853
	34.4	Integrity and Secure Linkage	853
	34.5	State Sharing Among Module Managers	854
Chapter	**35**	**Administration of Elective Module Managers**	**857**
	35.1	Integrity Verification	857
	35.2	Module Manager Credentials	857
	35.3	Installing an Elective Module Manager	859
	35.3.1	Global Unique Identifiers (GUIDs)	860
	35.4	Loading an Elective Module Manager	860
	35.4.1	Bilateral Authentication	861

	35.4.2	Protocol for Attaching a Service Module	862
	35.4.3	Protocol for Detaching a Service Module	863
	35.4.4	Protocol for Unloading a Service Module	863
Chapter	**36**	**Elective Module Manager Operations**	**865**
	36.1	Data Structures	865
	36.1.1	CSSM_STATE_FUNCS	865
	36.1.2	CSSM_MANAGER_EVENT_TYPES	865
	36.1.3	CSSM_MANAGER_EVENT_NOTIFICATION	866
	36.1.4	CSSM_MANAGER_REGISTRATION_INFO	866
	36.1.5	CSSM_HINT_xxx Parameter	867
	36.2	Elective Module Manager Functions	868
		Initialize	869
		Terminate	870
		ModuleManagerAuthenticate	871
		RegisterDispatchTable	872
		DeregisterDispatchTable	873
		EventNotifyManager	874
		RefreshFunctionTable	875
	36.3	CSSM Service Functions used by an EMM	876
		cssm_GetAttachFunctions	877
		cssm_ReleaseAttachFunctions	878
		cssm_GetAppMemoryFunctions	879
		cssm_IsFuncCallValid	880
		cssm_DeregisterManagerServices	882
Part	**14**	**Add-In Module Structure and Administration**	**883**
Chapter	**37**	**Introduction**	**885**
	37.1	Common Data Security Architecture	885
	37.2	Add-In Module Structure	888
	37.3	Module Installation	889
	37.4	Runtime LifeCycle of the Service Provider Module	890
Chapter	**38**	**Add-In Module Structure**	**891**
	38.1	Security Services	891
	38.2	Module Administration Components	892
	38.2.1	Integrity Verification	892
	38.2.2	Module-Granted Use Exemptions	892
	38.2.3	Service Module Requirements for USEE Tags Support	893
	38.2.4	Initialization and Cleanup	894
Chapter	**39**	**Add-In Module Administration**	**895**
	39.1	Manufacturing an Add-In Module	895
	39.1.1	Authenticating to Multiple CSSM Vendors	898
	39.1.2	Obtaining an Add-In Module Manufacturing Certificate	899
	39.1.3	Issuing an Add-In Module Product Certificate	899
	39.1.4	Manufacturing Add-In Modules	899

39.2 Installing a Service Module ... 900
39.2.1 Global Unique Identifiers (GUIDs) ... 901
39.2.2 The Module Description ... 901
39.3 Attaching a Service Module ... 902
39.3.1 Runtime Life Cycle of the Module ... 902
39.3.2 Bilateral Authentication ... 903
39.3.3 Memory Management Upcalls ... 904
39.4 Modules Control Access to Objects ... 904
39.4.1 Authentication as Part of Access Control ... 905
39.4.2 Authorization as Part of Access Control ... 906
39.4.3 Resource Owner ... 907
39.5 Error Handling ... 908
39.6 Data Structure for Add-in Modules ... 908
39.6.1 CSSM_SPI_ModuleEventHandler ... 908
39.6.2 CSSM_CONTEXT_EVENT_TYPE ... 909
39.6.3 CSSM_MODULE_FUNCS ... 909
39.6.4 CSSM_UPCALLS ... 909

Chapter 40 Add-In Module Interface Functions ... 913
CSSM_SPI_ModuleLoad ... 914
CSSM_SPI_ModuleUnload ... 915
CSSM_SPI_ModuleAttach ... 916
CSSM_SPI_ModuleDetach ... 918

Chapter 41 CSSM Upcalls for Service Provider Modules ... 919
cssm_CcToHandle ... 920
cssm_GetModuleInfo ... 921

Part 15 Appendices, Glossary and Index ... 923

Appendix A CSSM Error Handling ... 925
A.1 Introduction ... 925
A.2 Error Values and Error Codes Scheme ... 925
A.3 Error Codes and Error Value Enumeration ... 926
A.3.1 Configurable CSSM Error Code Constants ... 926
A.3.2 CSSM Error Code Constants ... 926
A.3.3 General Error Values ... 927
A.3.4 Common Error Codes For All Module Types ... 927
A.3.5 Common Error Codes for ACLs ... 928
A.3.6 Common Error Codes for Specific Data Types ... 930

Appendix B Application Memory Functions ... 935
B.1 Introduction ... 935
B.2 CSSM_API_MEMORY_FUNCS Data Structure ... 935

Appendix C Cryptographic Service Provider Behavior **937**
C.1 Introduction 937
C.1.1 Guidelines for Each Service Provider type.. 937
C.1.2 Typographic Conventions.. 937
C.2 Formats 938
C.2.1 Key Formats 938
C.2.1.1 Plaintext Keys 938
C.2.1.2 Key References 939
C.2.1.3 Wrapped Keys 940
C.2.2 Requesting Key Format Types 941
C.3 Events 943
C.3.1 Receiving Context Events 943
C.3.2 Sending Insert and Remove Events 943
C.3.3 Sending Fault Events.. 945
C.4 Memory Management 946
C.4.1 Types of Memory Allocation.. 946
C.4.2 Allocation of Key Information 946
C.4.3 Allocation of Single Output Buffers.. 947
C.4.4 Allocation of Vector-of-Buffers 947
C.5 CSP Query Mechanisms 948
C.5.1 Querying Key Sizes 948
C.5.2 Querying Output Sizes 948
C.5.3 Querying State of the CSP Subservice.. 949
C.6 Client Authentication and Authorization 951
C.6.1 Client Login ACLs 951
C.6.1.1 Managing Client Login ACLs 951
C.6.2 Individual Key ACLs 952
C.6.3 Protected Authentication Paths 953
C.7 Module Directory Service Information 954
C.7.1 Common Relation.. 954
C.7.2 CSP Primary Relation... 955
C.7.3 CSP Encapsulated Product Relation.. 956
C.7.4 CSP Smartcard Relation 957
C.7.5 CSP Capabilities Relation 958
C.7.5.1 Assigning GroupId Values... 958
C.7.5.2 Privileged Capabilities.. 959
C.7.5.3 Required Capability Attributes.. 959
C.8 CSP Multi-Service Modules with DL Interface 964
C.8.1 Purpose of CSP Multi-Service Modules... 964
C.8.2 Identifying Multi-Service Modules.. 964
C.8.3 Assigning Subservice Identifiers 964
C.8.4 Client Authentication and Authorization 964
C.8.5 Managing Multiple Key Storage Databases....................................... 964
C.9 Algorithm Reference 966
C.9.1 Conventions.. 966
C.9.2 Basic Algorithm Usage 967
C.9.2.1 Digital Signatures 967
C.9.3 Algorithm Parameters... 968

C.9.4 Algorithm List ... 968
C.9.4.1 RSA ... 968
C.9.4.2 Combination Signatures with RSA ... 969
C.9.4.3 DSA ... 970
C.9.4.4 Combination Signatures with DSA ... 970
C.9.4.5 Diffie-Hellman (PKCS 3) ... 971
C.9.4.6 Password Based Key Derivation (PKCS 5) ... 971
C.9.4.7 Generic Message Digests ... 973
C.9.4.8 Generic Block Ciphers ... 973
C.9.4.9 Generic Stream Ciphers ... 976
C.9.5 SSL 3.0 Algorithms ... 976
C.9.5.1 Data Structures ... 977
C.9.5.2 Pre-Master Key Generation ... 979
C.9.5.3 Master Key Derivation ... 979
C.9.5.4 Encryption and MACing Secret Key Derivation ... 980
C.9.5.5 MD5 and SHA-1 MACing ... 980

Appendix D Signed Manifests ... 981
D.1 Extensions to the JavaSoft/Netscape Specification ... 981
D.2 Core Set of Name:Value Pairs ... 981
D.3 Metadata ... 982
D.3.1 Integrity Core ... 982
D.3.2 Dublin Core ... 983
D.3.3 PKWARE Archive File Format Specification ... 983

Glossary ... 985

Index ... 991

List of Figures

1-1 The Common Data Security Architecture for all Platforms ... 7
2-1 Services Provided by CSSM ... 15
2-2 Attach Add-In Module and Load its Elective Module Manager ... 17
2-3 CSSM Dispatches Calls to Selected Add-In Security Modules ... 21
2-4 Indirect Creation of a Security Context ... 24
3-1 Multi-Service Add-In Module Serving Three Categories ... 27
3-2 Separate Handles Reference a Single Multi-Service Module ... 28
6-1 The Common Data Security Architecture for all Platforms ... 38
6-2 PKCS 11 Device Using Crypto and Persistent Storage Services ... 41
6-3 Multiple CSSM Vendors Authenticating Same Application ... 45
7-1 CDSA Add-In Module Structure ... 124
8-1 Certificate Life Cycle States and Actions ... 307
8-2 CDSA Add-In Module Structure ... 310
9-1 Credential Classes ... 407
9-2 Logic of Authorization ... 410
9-3 CDSA Add-In Module Structure ... 413
10-1 CDSA Add-In Module Structure ... 429

10-2 Certificate Life Cycle States and Actions 430
11-1 CDSA Add-In Module Structure 514
12-1 Common Data Security Architecture for all Platforms 589
12-2 MDS Architecture 591
12-3 Software Module Cross-Check 592
17-1 Key Recovery Phases 643
18-1 Elective Key Recovery Services in the CSSM 647
19-1 Encrypted Communications without Key Recovery 658
19-2 Encrypted Communications with Key Recovery Enablement 659
20-1 Bilateral Authentication Using Software Credentials 707
23-1 The Common Data Security Architecture for All Platforms 791
25-1 Signed Manifest Architectural View 795
25-2 Relationships of Manifest, Signer's Info and Signature Block 796
29-1 Relationship of Publisher's Archive and Signed Manifest 812
29-2 Relationship of Distributor's Archive to Publisher's Archive 813
29-3 Relationship of Reseller to Distributor to Publisher 814
33-1 Common Data Security Architecture for all Platforms 849
34-1 Steps to Load a Service Module and its Corresponding EMM 852
35-1 Certificate Chain for an Elective Module Manager 858
37-1 Common Data Security Architecture for all Platforms 886
37-2 CDSA Add-In Module Structure 888
39-1 Credentials of an Add-In Service Module 895
39-2 Certificate Chain for an Add-In Service Module 896
39-3 Signature File for Add-In Module with Authentication Capability ... 898

List of Tables

C-1 Plaintext Key Format Descriptor Values for CSSM_KEYHEADER 939
C-2 Default Plaintext Key Formats 939
C-3 Key Reference Format Descriptor Values for CSSM_KEYHEADER .. 940
C-4 Wrapped Key Format Descriptor Values for CSSM_KEYHEADER ... 941
C-5 Key Attribute Format Flags and Corresponding Format Class 942
C-6 APIs and the Appropriate Key Format Attributes 942
C-7 Actions Taken when Returning Values in a Single Buffer 947
C-8 Behavior of CSP_QuerySize for all supported Operation Types 949
C-9 CSSM_CSP_OPERATIONAL_STATISTICS Structure 949
C-10 CSSM_CSP_OPERATIONAL_STATISTICS::DeviceFlags Field 950
C-11 Contents of CDSA Common MDS Relation 955
C-12 Contents of the CSP Primary MDS Relation 956
C-13 Contents of CSP Encapsulated Product MDS Relation 957
C-14 Contents of CSP Smartcard MDS Relation 957
C-15 Contents of CSP Capabilities MDS Relation 958
C-16 Example Representation of Capabilities Set for CSP in MDS 959
C-17 Fixed Attribute Values for No Required Attributes 959
C-18 Capability Attributes for Random Number Generation 960
C-19 Capability Attributes for Message Digest Capabilities 960
C-20 Capability Attributes for Symmetric Key Generation 960
C-21 Capability Attributes for Symmetric Block Cipher 961

C-22 Capability Attributes for Symmetric Stream Cipher 962
C-23 Capability Attributes for Message Authentication Code...................... 962
C-24 Capability Attributes for Asymmetric Key Generation 962
C-25 Capability Attributes for Asymmetric Encryption................................ 963
C-26 Capability Attributes for Asymmetric Signature................................... 963
C-27 Capability Attributes for Key Derivation 963
C-28 Abbreviations for Algorithm Uses 966
C-29 Applicable Modes for CSSM_ALGID_RSA Context Type.................... 968
C-30 Applicable Modes for Combination Signatures with RSA 969
C-31 Applicable Modes for CSSM_ALGID_DSA 970
C-32 Applicable Modes for combination Signatures with DSA.................... 970
C-33 Context Types and Modes for PKCS 3 Diffie-Hellman......................... 971
C-34 Algorithm IDs and Parameter Structures for PKCS-5 PBD 972
C-35 Generic Message Digest Algorithm Identifiers and Standards............ 973
C-36 Algorithm IDs and Standards for Block Ciphers 975
C-37 Padding Modes for Block Ciphers.. 975
C-38 Algorithm IDs and Standards for Stream Ciphers 976
C-39 SSL 3.0 Algorithm IDs, Context Types and Parameter 977
C-40 MD5 and SHA-1 MAC Algorithms for MACs.. 980

Preface

The Open Group

The Open Group is a vendor and technology-neutral consortium which ensures that multi-vendor information technology matches the demands and needs of customers. It develops and deploys frameworks, policies, best practices, standards, and conformance programs to pursue its vision: the concept of making all technology as open and accessible as using a telephone.

The mission of The Open Group is to deliver assurance of conformance to open systems standards through the testing and certification of suppliers' products.

The Open group is committed to delivering greater business efficiency and lowering the cost and risks associated with integrating new technology across the enterprise by bringing together buyers and suppliers of information systems.

Membership of The Open Group is distributed across the world, and it includes some of the world's largest IT buyers and vendors representing both government and commercial enterprises.

More information is available on The Open Group Web Site at *http://www.opengroup.org*.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available on The Open Group Web Site at *http://www.opengroup.org/pubs*.

- Product Standards

  A Product Standard is the name used by The Open Group for the documentation that records the precise conformance requirements (and other information) that a supplier's product must satisfy. Product Standards, published separately, refer to one or more Technical Standards.

  The ‘‘X’’ Device is used by suppliers to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trademark 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 supplier. The Open Group runs similar conformance schemes involving different trademarks and license agreements for other bodies.

- Technical Standards (formerly CAE Specifications)

  Open Group Technical Standards, along with standards from the formal standards bodies and other consortia, form the basis for our Product Standards (see above). The Technical Standards are intended to be used widely within the industry for product development and procurement purposes.

  Technical Standards are published as soon as they are developed, so enabling suppliers to proceed with development of conformant products without delay.

  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand.

- CAE Specifications

  CAE Specifications and Developers' Specifications published prior to January 1998 have the same status as Technical Standards (see above).

- Preliminary Specifications

  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. There is a strong preference to develop or adopt more stable specifications as Technical Standards.

- Consortium and Technology Specifications

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

In addition, The Open Group publishes Product Documentation. This includes product documentation—programmer's guides, user manuals, and so on—relating to the DCE, Motif, and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

Versions and Issues of Specifications

As with all *live* documents, Technical Standards and 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 Open Group Web Site at *http://www.opengroup.org/corrigenda*.

Ordering Information

Full catalog and ordering information on all Open Group publications is available on The Open Group Web Site at *http://www.opengroup.org/pubs*.

This Document

This document is the CDSA Version 2.3 Technical Standard C914. It supersedes the November 1999 CDSA Version 2 Technical Standard (C902).

The changes from C902 are the corrections collected in Corridendum 1, plus an extensive restructuring of the complete document to eliminate unnecessary duplication of definitions and description.

The Common Data Security Architecture (CDSA) is a set of layered security services that address communications and data security problems in the Internet and Intranet application

space. It is designed to provide interoperable security standards covering all the essential components of security capability.

History

The following summary provides a chronological history of the development of CDSA specifications since December 1997. It is intended for clarification purposes, in recognition that there is possibility of confusion over past version numbering assigned to previously-released CDSA documents and related software.

Any CDSA specifications released prior to December 1977 pre-date The Open Group's involvement.

- In December 1997 The Open Group published its first CDSA Technical Standard (C707).
- In Febuary 1998, Intel released a series of documents, all called "Common Data Security Architecture xx Specification, Release 1.2 February 1998". Titles in this Intel release included Application Programming Interface (API), Data Storage Library Interface (DLI), Add-in Module Structure and Administration, Cryptographic Service Provider Interface (SPI), Trust Policy Interface (TPI), and Certificate Library Interface (CLI). Those who licensed the reference software for this Intel documentation release 1.2 have not unnaturally refered to their implementations of it as version 1.2 of CDSA.
- In May 1999, Intel released a document called "Common Security Services Manager, Application Programming Interface (API) Specification, CDSA Version 2.0 release 3.0". This had review status only.
- In November 1999, The Open Group published its CDSA Version 2 Technical Standard (C902). This revision superseded the C707 Standard. The C902 Standard was based on considerable implementation experience, which was reflected in the very substantial changes from the previous C707 Standard
- As a result of further detailed implementation work, including by Apple, and by Intel on their "Open Source" launch of their implementation, further detailed corrections were collected as a Corrigendum to the C902 (CDSAv2) Standard. At the same time, The Open Group restructured the complete Standard to remove considerable duplication of technical information — particularly in duplicated man-page definitions for related CSSM API and Service Provider interface function calls. The resulting revised CDSAv2 incorporating all these Corrigendum changes was published in May 2000 as the *Common Security: CDSA and CSSM, Version 2.3* Technical Standard, C914.

Trademarks

Motif®, OSF/1®, UNIX®, and the ''X Device'' are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.

Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner's benefit, without intent to infringe.

Acknowledgements

The Open Group gratefully acknowledges the co-operative effort of participating industry leaders, led by Intel Architecture Labs., on this Common Data Security Architecture (CDSA) specification. This work was initiated by Intel Architecture Labs., and led to the development of CDSA and CSSM, having attained the support and participation of organizations such as Apple, 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, who met regularly under the auspices of The Open Group.

The Open Group particularly acknowledges the detailed work contributed by Apple Computer Corporation, Intel Architecture Labs. and the IBM Corporation, to the development of the CDSA Version 2 Technical Standard, and to the ongoing work contributed by Apple Computer Corporation and Intel Architecture Labs in the development of this CDSAv2 plus Corrigendum Technical Standard.

Referenced Documents

The following documents are referenced in this Technical Standard:

ASN.1

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

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.

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.

PKCS #1

RSA Encryption Standard, RSA Data Security, Inc., October 1, 1998, Version 2.0.

PKCS #3

Diffie-Hellman Key-Agreement Standard, RSA Data Security, Inc., November 1, 1993, Version 1.4.

PKCS #7

Cryptographic Message Syntax, RSA Data Security, Inc., November 1, 1993, Version 1.5.

PKCS #8

Private-Key Information Syntax Standard, RSA Data Security, Inc., November 1, 1993, Version 1.2.

PKIX

Public Key Infrastructure Certificate Management Protocols, IETF PKIX Working Group, 1996

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

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

License Agreement for CDSA Specifications

THIS LICENSE AGREEMENT IS IN RESPECT OF THE COMPILATION OF 15 SPECIFICATIONS RELATING TO COMMON DATA SECURITY ARCHITECTURE ''(CDSA)'' AND COMMON SECURITY SERVICES MANAGER ''(CSSM)'', PUBLISHED TOGETHER BY THE OPEN GROUP UNDER THE TITLE ''COMMON SECURITY: CDSA AND CSSM, Version 2'', DOCUMENT NUMBER C902, ISBN 1-85912-236-1 (''THE SPECIFICATION'').

YOU CANNOT USE THIS SPECIFICATION (''THE SPECIFICATION'') FOR SOFTWARE DEVELOPMENT UNTIL YOU HAVE CAREFULLY READ AND AGREED TO THE FOLLOWING TERMS AND CONDITIONS. THE PERSON WHO ORIGINALLY ACQUIRED THIS PUBLICATION THROUGH THE WORLD-WIDE WEB OR AS HARD COPY EXPLICITLY AGREED TO THESE TERMS AND CONDITIONS. AS THE READER OF THIS DOCUMENT YOU ARE BOUND BY THE SAME TERMS. THE TERMS OF THIS LICENSE AGREEMENT ALSO APPLY TO REVISIONS OF THIS SPECIFICATION MADE AVAILABLE TO YOU BY THE OPEN GROUP.

LICENSE: The Open Group grants you a non-exclusive copyright license to read and display the Specification, and to use the Specification to develop and distribute a conformant software implementation of the Specification on the terms set out in this Agreement. For the avoidance of doubt, this License does not authorize you to edit, republish or distribute the Specification or create any derivative work therefrom.

CONFORMANCE: A software implementation must be and remain a complete and conformant implementation of the CSSM. A conforming implementation of CSSM provides and supports all the application programming interfaces and service provider interfaces defined in the Specification, and for each elective module the implementation must provide and support all the application programming interfaces and service provider interfaces for that module. A software implementation of CSSM may be tested for conformance using the CDSA Conformance Test Suite (''the Test Suite''), available from The Open Group web site. You are not permitted to use the Test Suite for any other purpose, nor to disclose or make any claim that any product has ''passed'' the Test Suite test. You can not make any claims that your software product conforms to CDSA or CSSM or the Specification unless such product is registered under the Open Brand program.

LIABILITY: THE SPECIFICATION AND ANY OTHER MATERIALS PROVIDED BY THE OPEN GROUP UNDER THIS AGREEMENT ARE PROVIDED ''AS IS'', AND THE OPEN GROUP MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AND EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS AND FITNESS FOR A PARTICULAR PURPOSE.

TO THE MAXIMUM EXTENT PERMITTED BY LAW, THE OPEN GROUP HEREBY EXCLUDES ALL LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING OUT OF OR RELATING TO THE USE BY ANY PERSON OF THE SPECIFICATION OR ANY OTHER MATERIAL PROVIDED HEREUNDER. IN NO EVENT SHALL THE OPEN GROUP BE LIABLE FOR ANY INDIRECT OR CONSEQUENTIAL LOSSES INCLUDING, WITHOUT LIMITATION, ANY LOSS OF PROFITS, CONTRACTS, PRODUCTION OR USE.

TERMINATION OF THIS LICENSE: The Open Group may terminate this license at any time if you are in breach of any of its terms and conditions. Upon termination, you will immediately cease use of the Specification.

APPLICABLE LAW: This Agreement is governed by the laws of England and Wales, and you hereby agree to the non-exclusive jurisdiction of the English courts.

Technical Standard

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 addresses 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 pre-defined, global policies)
- Supporting application-specific policies by providing an extensibility mechanism that manages domain-specific policy 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, integrity and authentication credentials, and documents
- Supporting existing secure protocols such as Secure Sockets Layer (SSL), Secure Multipurpose Internet Mail Extensions (SMIME), and Secure Electronic Transaction protocol (SET), as layered system services
- Encapsulating existing industry standard security service Application Programming Interfaces, such as PKCS#11 for cryptographic tokens.

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 CDSA 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 begin with fundamental components such as cryptographic algorithms, random numbers, and unique identification information. The layers build up to digital certificates, key management mechanisms, integrity and authentication credentials, 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, data storage library modules, and authorization computation 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
- Contains the kernel of the trusted computing base and serves as the integrity watch-dog in the dynamic environment.

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. Five basic types of module managers are defined:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Library Services Manager
- Data Storage Library Services Manager
- Authorization 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. Applications can use multiple service providers of all types concurrently. 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.

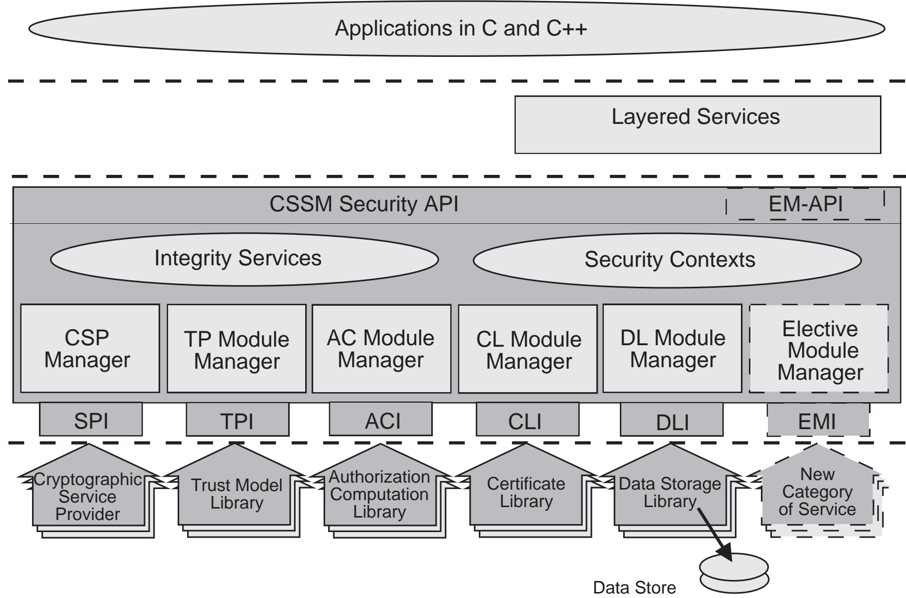


Figure 1-1 The Common Data Security Architecture for all Platforms

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 integrity, authentication, and exemption verification when dynamically attaching services
- Secure linkage checks on calls to service provider modules
- 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 Module Manager concurrently manages service modules for the manager's respective functional category. CSSM defines five basic categories of

service and their corresponding managers:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Services Manager
- Data Store Services Manager
- Authorization Computation 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 attached and used by any caller in the system. 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 group of digital certificates. All domain-specific policy 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.

The **Authorization Computation Services Manager** administers the Authorization Computation Service Providers that may be installed on the local system. It defines a common API for accessing Authorization Computation Service Providers. The API defines a general authorization evaluation service that computes whether a set of credentials and samples are authorized to perform a specific operation on a specific object. All authorization evaluation functions are implemented by an AC service provider.

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 the integrity, identity, and use authorizations of 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 Embedded Integrity Services Library API* 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 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 five basic service categories are:

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

Every instance of an add-in module must be installed with the Module Directory Services (MDS) making the module accessible for use on the local system. The installation process persistently records the module's identifying name, a description of the services it provides, and the information required to dynamically load the module. Applications can query MDS to obtain information used to select one or more service 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 inserts multiple MDS records defining 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, and carried out by DL modules.

CL modules may implement their services locally or remotely. For example, certificates can be issued by a remote CA. Access to that CA process can be implemented using standard message protocols such as PKCS#10, and may be transport-independent.

1.2.5.4 Data Storage Library Modules (DLs)

Data storage library modules provide stable storage for security-related data objects. These objects can be certificates, certificate revocation lists (CRLs), cryptographic keys, integrity and authentication credentials, 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, credentials, and policy should be interpreted by layered services, TP modules and applications. To ensure necessary interoperability among DL modules, a minimal schema is defined for each DL record type. The schema prescribes the minimum set of attributes that all applications can use to select records from a data store. A DL module can support additional application-defined and modules-defined attributes for a DL record type.

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 Authorization Computation Modules (ACs)

Authorization Computation modules implement an authorization evaluation mechanism based on caller inputs. Callers provide:

- The assumptions forming the basis of the caller’s policy
- The request for which authorization is being checked
- The credentials, samples, and exhibits that could demonstrate authorization to perform the request

The AC evaluation engine determines whether the request is authorized based on the assumptions and the caller credentials. The AC Module can provide other services related to authorization computations through the *CSSM_AC_PassThrough*( ) function.

1.2.5.6 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 distinct runtime identifiers to reference the module for each category of service. The multiple identifiers reference a single, dynamic service module implementation. Multi-service add-in modules separate 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 service modules. 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 the ability to attach a dummy add-in module and perform all required integrity and authentication checks using a bilateral authentication protocol
- Support for all of the basic APIs, by correct dispatching of calls and parameters to attached, dummy service providers

- 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
- Maintain integrity of the computing base through secure linkage and other periodic integrity checks.

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
- Certificate and CRL Format Conformance Tests
- Remote Service Protocol Conformance 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 than 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—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 component integrity and trusted identification of the component's source.
- Security context management—aggregate and manage input parameters required when performing cryptographic operations.

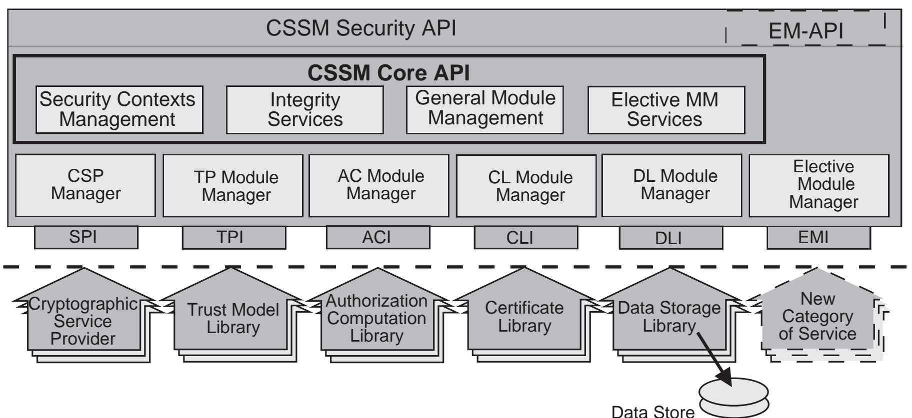


Figure 2-1 Services Provided by CSSM

2.2 General Module Management Services

CDSA components use the Module Directory Services (MDS) to record:

- A description of the component, its capabilities and services
- The component's integrity and authorization credentials
- The location of the component's implementation.

Each installed component, including service modules, Elective Module Managers, and CSSM itself must insert one or more MDS records during installation. This information can be queried by applications, add-in modules, and components of CSSM. MDS protects this information base by controlling access to the information, (particularly write access), and can check the integrity of stored values upon retrieval.

MDS defines a set of relations and their associated schemas for each type of CDSA component. Using this schema information, applications can query MDS to select service modules fit to task. Searches can be performed by name or by features/capabilities. MDS schemas and facilities are fully-defined in the Module Directory Services Specification. Once an applicable module is discovered, the application uses the CSSM *Load* and *Attach* operations 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. 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 application's 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. The uninstall process must delete the MDS records associated with the module.

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. User authentication through biometric devices and the maintenance of audit logs are other potential categories of elective service. Not all platforms will be equipped with biometric devices, and not all applications require an audit trail.

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.

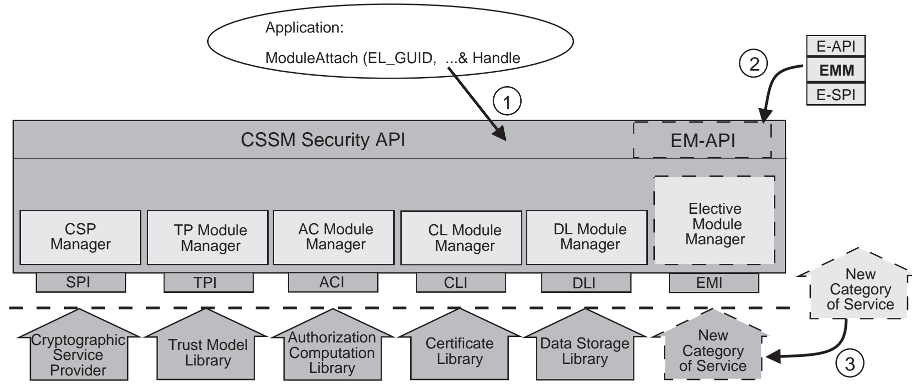


Figure 2-2 Attach Add-In Module and Load its Elective Module Manager

2.3.2 Registering Module Managers

Module managers are installed and registered with MDS in a similar manner to service modules. MDS defines a relation for Elective Module Manager (EMM) information, including:

- A description of the EMM service category
- The identity of the EMM vendor
- The EMM's integrity and authorization credentials
- The location of the EMM's implementation.

This information can be queried, but typically only system administration applications will use MDS-managed 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 affect 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 occur.

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 messaging service

The first two mechanisms depend on platform services outside of CDSA. Module managers that share state information can use all of these mechanisms. However, using custom internal interfaces or OS-specific mechanisms is discouraged as it detracts from creating portable EMMs. When using CSSM-supported messaging, module managers should define and publish a message-based protocol, so that all implementations of the participating module managers could choose to support the message protocol.

CSSM-supported event notification 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.

Generic message types include:

- Request
- Reply.

2.4 Basic Module Managers

CDSA defines module managers for five basic types of service:

- Cryptographic Services Module Manager
- Certificate Library Module Manager
- Data Storage Library Module Manager
- Trust Policy Module Manager
- Authorization Computation Services 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 module 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 that 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.

Calls to the CSSM security API can originate in an application, in another add-in security module, or in CSSM itself. 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.

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.

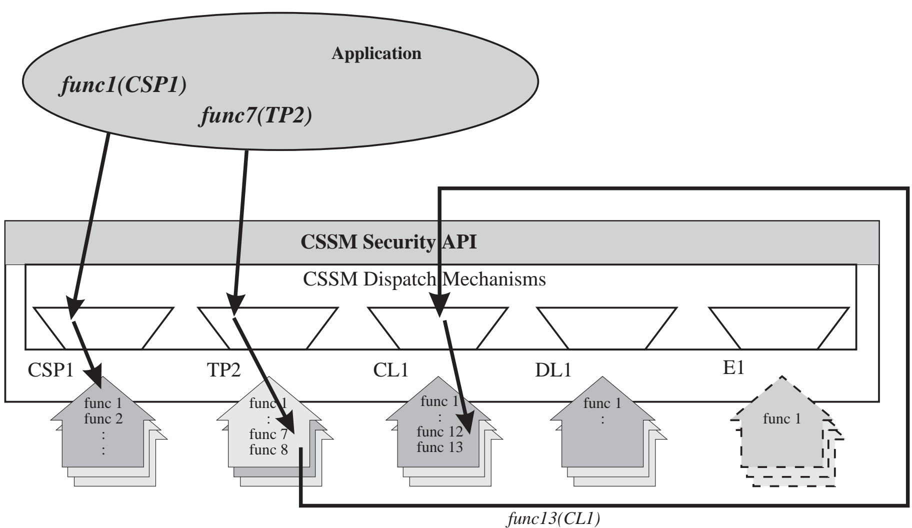


Figure 2-3 CSSM Dispatches Calls to Selected Add-In Security Modules

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 the attaching party to participate 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 could be interrupted.

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

- A certificate, signed with the private key of 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 private key associated with the component's certificate
- A set of object code modules, signed with the private key associated with the component's certificate

These credentials are stored in s Module Directory Services relation that records information about CDSA components. CSSM's credentials are also stored in MDS during CSSM installation.

During *ModuleLoad* and *ModuleAttach* processing, CSSM performs the first half of the bilateral protocol, which proceeds as follows:

- Use MDS to obtain the component's credentials
- Verify the signature covering the integrity of the component's executable object code
- Verify the component's certificate and manifest
- Load the component's executable object code
- 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
- Use MDS to obtain CSSM's credentials
- 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.

Secure linkage checks can be performed each time CSSM invokes a service module. Service modules are encouraged to perform a secure linkage check on CSSM before servicing a call, particularly if the requested operation is a privileged service.

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 in MDS during application installation
- Perform their half of the bilateral authentication process with CSSM

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 the private key associated with a CSSM-recognized certificate owned by the manufacturer.
- Uses that same private key 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 can 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.

The field upgrade procedures necessary to support cross-certification in deployed systems may not be practical. For this reason, a single, industry-wide root of trust for integrity verification is strongly encouraged. Additional signatures representing private business agreements could be required and verified in those situations where deemed necessary.

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 or return a denial of service error to the caller.

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, and other credentials. Applications are responsible for protecting these secrets. Applications desiring maximal protection should use passphrase callback functions that limit the duration in which the passphrase or other credentials are visible 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 the Module Directory Services to obtain the capabilities of a Cryptographic Service Provider (CSP).

The context creator owns the cryptographic context. Because the context can contain secrets, the context can be used by other agents only if the context creator has authorized that agent to use the context on their behalf.

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. Figure 2-4 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 the authorized context 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.

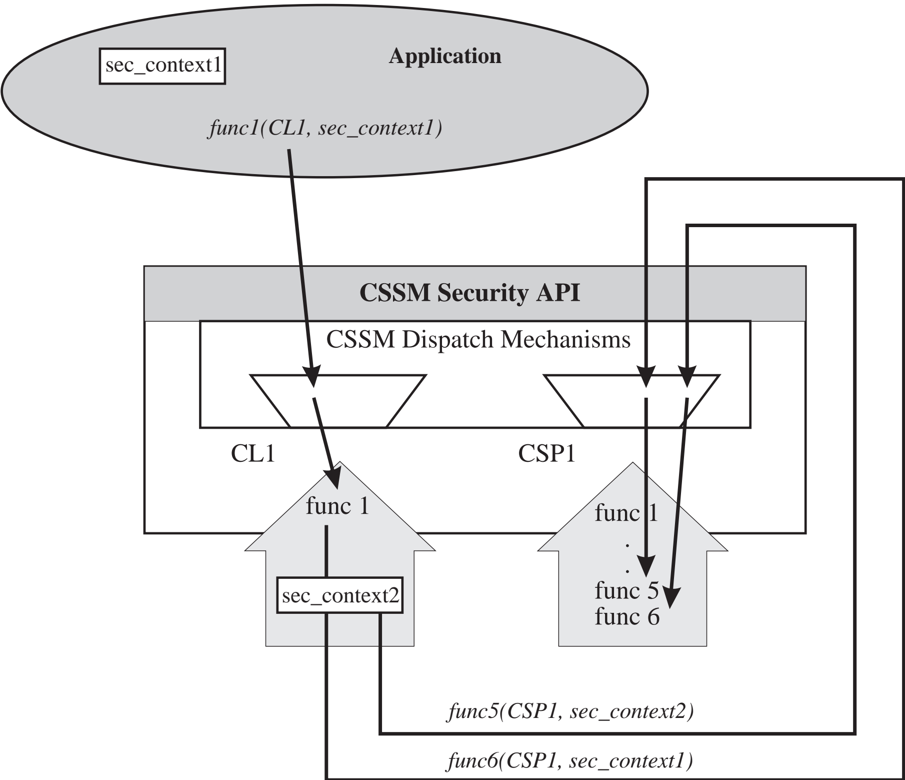


Figure 2-4 Indirect Creation of a Security Context

These hidden 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
- Authorize other agents (such as service modules) to use an application-created cryptographic context.
- Enable development of re-entrant CSPs
- Enable development of re-entrant layered services
- Enable development of re-entrant applications

Chapter 3

Multi-Service Modules

3.1 Overview

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 3-1.

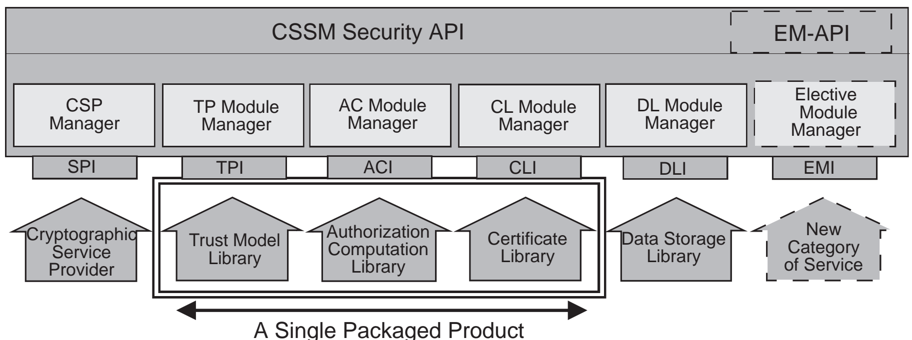


Figure 3-1 Multi-Service Add-In Module Serving Three Categories

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.

3.2 Application Developer View of a Multi-Service Add-In Module

Application developers require no special knowledge of the organization of the service provider modules available through the CSSM framework. Applications attach a multi-service module as they would any other module. Each call to attach a service module returns a handle representing a unique pairing between the caller and the attached module. The caller uses this handle to obtain the single type of service specified in the attach operation. A second attach of the same module for a different type of service returns a second attach handle, but does not load another copy of the service module. Figure 3-2 shows the handles for an attached PKCS#11 service provider that performs cryptographic operations and persistent storage of certificates.

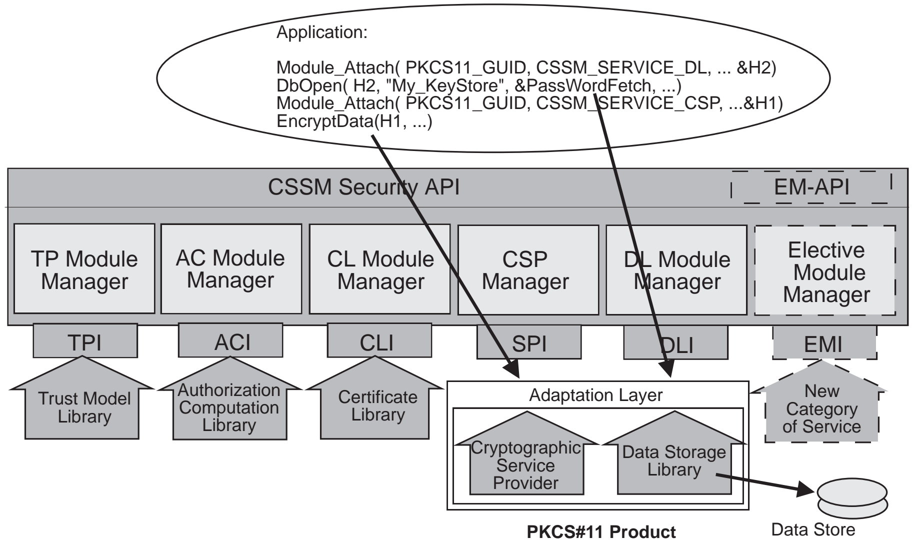


Figure 3-2 Separate Handles Reference a Single Multi-Service Module

Multiple calls to attach are viewed as independent requests with respect to authorized exemptions and access control. The multi-service module can match-up the caller handles if a shared execution state is required.

3.3 Service Provider 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 by adding records to relations in the Module Directory Services (MDS). MDS makes this information 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 category. The CSSM dispatching mechanism invokes only to those interfaces registered with the CSSM.

Chapter 4

Modules Control Access to Objects

4.1 Overview

Service provider modules manage objects that are manipulated through the service provider's APIs. Each service provider can control access to these objects on a per request basis according to a policy enforced by the provider. Most of the access-controlled objects are persistent, but they can exist only for the duration of the current application execution. Examples include:

- Authorization to use a cryptographic key stored by a CSP
- Authorization to use a particular secret managed by a CSP
- Authorization to write records to a particular data store

A service provider must make an access control decision when faced with a request of the form "I am subject S. Do operation X for me." The decision requires the service provider to answer two questions:

- Is the requester really the subject S?
- Is S allowed to do X?

The first question is answered by authentication. The second question is answered by authorization.

4.2 Authentication as Part of Access Control

There are various forms of authentication. Traditionally, the term is applied to authentication of the human user. A human is often authenticated by something he or she knows, such as a passphrase, a PIN, etc. More secure authentication involves multiple factors:

- something the human knows
- something the human possesses
- something the human is, in the form of a biometric authentication.

It is also possible to authenticate an entity using public key cryptography and digital certificates. The entity holding a keypair can be a hardware device or instance of some software. The device or the software acts on behalf of a human user. Each entity is identified by a public signature key. Authentication is performed by challenging the entity to create a digital signature using the private key. The signature can be verified and the entity is authenticated. The digital certificate and the digital signature are credentials presented by the entity for verification in the authentication process.

Each service provider defines a policy regarding the type and number of authentication credentials accepted for verification by the service module. The credentials can be valid for some fixed period of time or can be valid indefinitely, until rescinded by an appropriate revocation mechanism.

CDSA defines a general form of access credential a caller can present to service providers when operating on objects, whose access is controlled by the service provider. A credential set consists of:

- Zero or more digital certificates
- Zero or more samples

If the service provider caches authentication and authorization state information for a session, a caller may not be required to present any certificates or samples for subsequent accesses. Typically at least one sample is required to authenticate a caller and to verify the caller's authorization to perform a CDSA operation.

The general credential structure is used as an input parameter to functions in various categories of security services. A caller can provide samples through the access credentials structure in one of several modes or forms:

- Immediate values contained in the credentials structure - for example, a PIN, or a passphrase
- By reference to another authentication agent who will acquire and verify the credentials - for example, a biometric device and agent to acquire and verify biometric data from the caller, a protected PIN pad or some external authentication mechanism such as PAM.
- By providing a callback function that the service provider can invoke to obtain a sample on-demand - for example, invoking a function challenging the caller to sign a nonce
- Any combination of these forms

The service provider uses credentials to answer the authentication question.

4.3 Authorization as Part of Access Control

Once any necessary authentication samples have been gathered, authorization can proceed. Just providing a password or biometric sample does not imply that the user providing the sample should get the access he or she is requesting. An authorization decision is based on an authorization policy. In CDSA, an authorization policy is expressed in a structure called an Access Control List (ACL). An ACL is a set of ACL entries, each entry specifying a subject that is allowed to have some particular access to some resource. Traditional ACLs, from the days of early time sharing systems, identify a subject by login name. The ACLs we deal with can identify a subject by login name, but more generally, the Subject is specified by the identification template that is used to verify the samples presented through the credentials.

The ACL associated with a resource is the basis for all access control decisions over that resource. Each entry within the ACL contains:

- Subject - a typed identification template (a type designator is part of the Subject, because multiple Subject types are possible)
- Delegation flag - indicating whether the subject can delegate the access rights (this only applies to public key templates)
- Authorization tag - defining the set of operations for which permission is granted to the Subject (the definition of authorization tags is left to the Service Provider developer, but in the interest of increased interoperability, we define tags for the basic operations represented by the defined standard API.
- Validity period - the time period for which the ACL entry is valid
- Entry tag - a user-defined string value associated with and identifying the ACL entry

The ACL entry does not explicitly identify the resource it protects. The service provider module must manage this association.

The basic authentication process verifies one or more samples against templates in the ACL. Each ACL entry with a verified subject yields an authorization available to that subject.

Beyond the basic process, it is possible to mark an ACL entry with the permission to delegate. The delegation happens by one or more authorization certificates. These certificates act to connect the authorization expressed in the ACL entry from the public key subject of that entry to the authorization template in the final (or only) certificate of the chain. That is, an authorization certificate acts as an extension cord from the ACL to the actual authorized subject. Delegation by certificate is an option when scaling issues mitigate against direct editing of the ACL for every change in authorized subject.

Service provider modules are responsible for managing their ACLs. When a new resource is created at least one ACL entry must be created. The implementation of ACLs is internal to the service provider module. The CDSA interface defines an CSSM_ACL_ENTRY_PROTOTYPE that is used by the caller and the service provider to exchange ACL information.

When a caller requests the creation of a new resource, the caller should present two items:

- A set of access credentials
- An initial ACL entry

The access credentials are required if the service provider module restricts the operation of resource creation. In many situations, a service provider allows anyone to create new resources. For example, some CSPs allow anyone to create a key pair. If resource creation is controlled, then the caller must present a set of credentials for authorization. Authentication will be performed based upon the set of samples introduced through the credentials. Upon successful authentication, the resulting authorization computation determines if the caller is authorized to create new resources within a controlled resource pool or container. If so, the new resource is created.

When a resource is created, the caller also provides an initial ACL entry. This entry is used to control future access to the new resource and its associated ACL (see Section 4.4). The service provider can modify the caller-provided initial ACL entry to conform to any innate resource access policy the service provider may define. For example, a smartcard may not allow key extraction. When creating a key pair on the smartcard, a caller can not give permission to perform the CDSA operation CSSM_WrapKey. The attempt will result in an error.

4.4 Resource Owner

How a given resource controller actually records the ownership of the resource is up to the developer of that code, but the "Owner" of a resource can be thought of as being recorded in a one-entry ACL of its own. Therefore, conceptually there are two ACLs for each resource: one that can grow and shrink and give access to users to the resource, and another that always has only one entry and specifies the owner of the resource (and the resource ACL). On resource creation, the caller supplies one ACL entry. That one entry is used to initialize both the Owner entry and the resource ACL. This is to accommodate the common case in which a resource will be owned and used by the same person. In other cases, either the Owner or the ACL can be modified after creation.

Only the "Owner" is authorized to change the ACL on the resource, and only the "Owner" is authorized to change the "Owner" of the resource. Effectively, the "Owner" acts as "the ACL on the ACL" for the full lifetime of the resource. In terms of an ACL entry, it only contains the "subject" (i.e., identifies the "Owner"), and the "delegate" flag (initially set to "No delegation"). The "Authorization tag" is assumed to convey full authority to edit the ACL, and the "Validity

period" is assumed to be the lifetime of the resource. There is no "Entry tag" associated with the "Owner". Note that an "Owner" may be a threshold subject, identifying many "users" who are authorized to change the ACL. Note also that "Ownership" does not convey the right to delete the resource; that right may or may not be conveyed by the ACL.

CDSA defines functions to modify an ACL during the life of the associated resource. ACL updates include:

- Adding new entries
- Replacing/updating existing entries
- Deleting an existing entry
- Changing the "Owner"

Modifying an ACL is a controlled operation. Credentials must be presented and authenticated to prove that the caller is the "Owner".

Chapter 5

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 J-CDSAAPI)
- CSSM management tools (such as a CSSM installation and configuration tool)
- High-level APIs that abstract a subset of CSSM APIs for use in a specific application domain, providing simplified, default behavior.

Technical Standard

Part 2:

CSSM Core Services

The Open Group

Chapter 6

CSSM Core Services

6.1 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 6-1 shows the basic layers of the Common Data Security Architecture:

- Applications
- System Security Services
- Common Security Services Manager
- 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 five 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 five basic categories of security services modules are:

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

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.

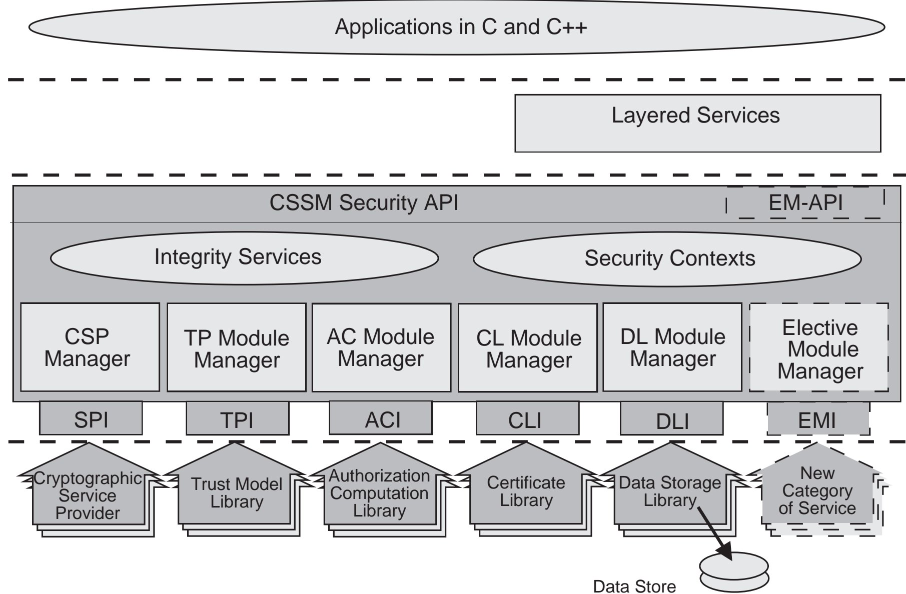


Figure 6-1 The Common Data Security Architecture for all Platforms

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. A standalone registry system called the Module Directory Services (MDS) provides applications with information about the service modules available for use by applications.

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. Service providers can require caller authentication before providing services. Application authentication is based on signed manifest credentials associated with the application.

Service modules can leverage other service modules in the implementation of their own services. Service modules acquire attach handles to other modules by:

- Receiving additional module handles from an invoking application
- Selecting and attach additional service module directly.

To prevent stealth attacks, CSSM performs secure linkage checks on function invocation.

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 runtime access to security service modules.

Security context management provides runtime caching of user-specific, cryptographic context information. Multi-step cryptographic operations, such as staged hashing, require multiple calls to a CSP.

CSSM, security services modules, and optionally applications, 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.

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

- Comprehensive, extensible SPIs for each of five categories of security services.
- Runtime management and access to all security service modules.
- Runtime management and access to elective module managers providing new security services.
- Caching of context information 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.
- Management support for concurrent security operations.

6.2 Selecting CDSA Components

A single system can host 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 compile-time or at runtime, depending on how the CSSM is deployed. The dynamic components of a CDSA configuration require some level of compatibility to interoperate correctly. Three pieces of information form the basis for determining compatibility and interoperability among CSSM, service modules, EMMs, and applications:

- a unique identification GUID, which distinguishes the component and its manufacturer
- major and minor version numbers, which further distinguishes the supported APIs, feature set, and bug fixes of the component.

The Module Directory Services (MDS), which is a standalone service outside of CDSA, implements a database describing CDSA components available from the local platform. Applications and CDSA components can query MDS to obtain the compatibility information and numerous other attributes describing features of the CDSA components. This information can be used as the basis for selecting appropriate and compatible components at runtime.

Every CDSA component must have a unique identification GUID. Not all CDSA applications are required to have an identifying GUID, but it is highly recommended. MDS uses the GUID as the primary database key for locating information about the CDSA component. Specification of the

the version numbers is optional, but believed to be of value as an augmentation to the distinguished name for an executable CDSA component.

When components are selected at runtime, applications use the MDS query functions to select components based on GUID or based on other properties of the component (such as the algorithms or features provided by the component).

If the application is selecting a CSSM at runtime, the application is responsible for loading the selected CSSM using services provided by the platform-specific environment. The application is also responsible for any load-specific initialization, such as symbol resolution. Once a CSSM has been loaded with the application, CSSM initialization proceeds as usual with the application invoking *CSSM_Init*( ) prior to calling any other CSSM interfaces.

Note: Applications can be statically bound with a particular version of CSSM. In this case, the application proceeds with CSSM initialization by invoking *CSSM_Init*( ).

Applications also use MDS to identify a service module providing the features and services required by the application. Once the service module has been identified, the application must use the CSSM services *CSSM_ModuleLoad*( ) and *CSSM_ModuleAttach*( ) to select and initiate a session with the service module. CSSM and service provider enforced integrity checks ensure that the application can not obtain direct access to service module. All service module access is mediated by the selected CSSM.

6.3 Core Services

The CSSM provides a set of core service APIs for:

- Module Management
- Memory Management Support (described in more detail in Appendix B on page 935)
- Security Context Management (described in Chapter 7 on page 123)
- Integrity Verification Services

These APIs are implemented by the CSSM, not by security service modules.

6.3.1 Module Management Services

The CSSM module management functions support dynamic selection and loading of service provider modules.

Applications can have *apriori* knowledge of the service module to be used or can use the Module Directory Services (MDS) to search the CDSA Directory database for service modules that provide the features and services required by the application. When service providers are made accessible on the local platform (typically through platform-specific installation procedures), the installation procedure includes registering the service module in MDS's CDSA database.

MDS records information about each installed service module and elective module manager available on the local system. MDS supports the following services and features with respect to the CDSA database:

- Persistently store values identifying and describing each dynamic CDSA component installed on the platform.
- Retrieval of information from the database upon request.
- Control of write-access to the database.

The database entries are queried by applications, service modules, EMMs, and CSSM.

Applications select the particular security service they will use by selectively loading and attaching service 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 an input parameter when requesting services from the attached module. CSSM uses the handle to match the caller with the appropriate service module and service type.

The calling application attaches to a module once for each type of service it requires. If the service type requested by the application is not implemented by the module, the attach fails. Figure 6-2 shows how the handles for an attached PKCS#11 service provider are used to perform cryptographic operations and persistent storage of certificates.

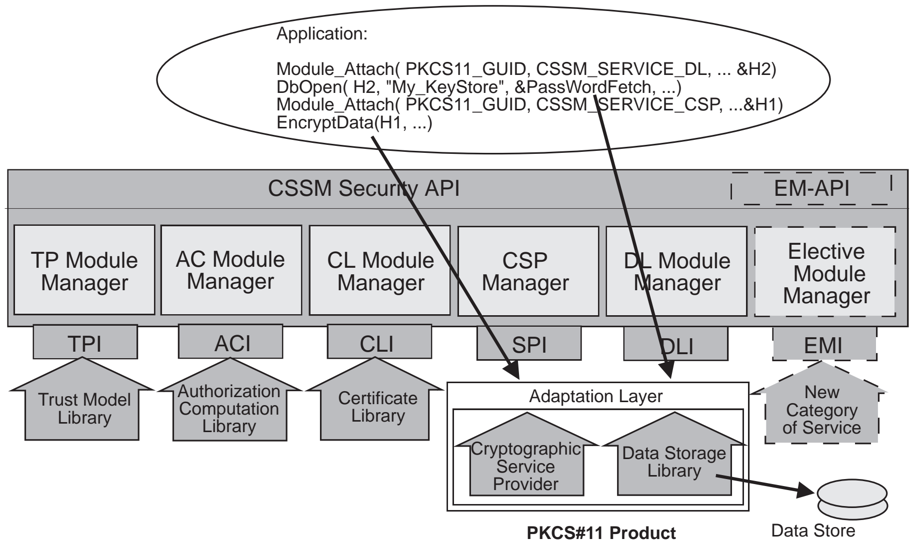


Figure 6-2 PKCS#11 Device Using Crypto and Persistent Storage Services

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 MDS CDSA database 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. The MDS CDSA database stores one record for each service category provided by a

service module.

6.3.2 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.

6.3.3 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 security 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.

CSSM requires successful certificate-based trust verification of all service modules when processing a *CSSM_ModuleLoad*() request. CSSM also requires trust verification of all Elective Module Managers when processing a *CSSM_ModuleAttach*() request.

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

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 verifies and loads a module or a module manager. If verification fails, then the module or the module manager is not linked or loaded.

6.3.4 CDSA and Privileges

The CDSA environment supports modal behavior based on privilege granted through signed manifests and associated certificates. There are two classes of privileges:

- application
- export

Application privileges are specific to a class of applications and service providers. If used, CSSM will ensure the calling module and the target service provider are authorized to switch to the mode implied by a chosen privilege. The export privileges are a class of privileges allowing modal behavior in service providers which coincides with the export regulations over cryptography. More detail on export privileges is provided below.

Application privileges may be defined by The Open Group, and may also be vendor defined. The representation of privileges uses two 4-byte words (or an 8-byte word if the platform

supports 64-bit words) The number space is divided in half between standard privileges and vendor defined privileges. The lower 32-bits are reserved for standard privileges and the upper 32-bits allocated to vendor specific privileges. The application privileges are granted to modules in signed manifests using the tag CDSA_PRIV. The CDSA_PRIV tag is added to the *signer_info* section of the module manifest. The associated value is a string of 4 byte base-64 encoded numbers, in big-endian order. Encoded 4-byte words are separated by colons. These numbers are the same numbers defined for the high-order word of CSSM_PRIVILEGE.

6.3.5 CDSA and USEE Privileges

Export privileges enable strong cryptography to be made available through the CDSA framework to applications performing special functions — where they would qualify for special case exemption from export regulations. Exportable components must enforce export policies to a "reasonable" degree, such that the effort to circumvent the enforcement mechanisms is at least as difficult as engineering the desired functionality from scratch.

All of the CDSA logical components may be subject to export controls. Exemptions to the export policy of public record are modeled using an artifact called USEE tags. Any time an exemption to export policy is granted, a USEE tag may be defined. The semantics of the USEE tag is interpreted in the context of the Exemption Rules. A standards body should control USEE namespace and provide interpretations of the meaning of USEE tags for developers. Specific requirements for USEE tags:

- USEE tags are placed in the *signer_info* section of the module manifest such that the tag semantics can be interpreted to apply to all manifest sections. An authorized authority issues USEE tags.
- The USEE exemption tag lists the exemption categories for cryptographic services. These categories can be supported by a cryptographic service provider. An application can be authorized to receive these services. The granting of one or more exemptions is recorded in an application's signed manifest credentials Cryptographic service providers record their ability to honor such exemptions in their signed manifest credential. Use exemptions may be defined for other categories of security services, as needed.

Exemptions are defined for financial, medical, insurance, key recovery, key exchange and authentication. SSL is also cited as a special case exemption in that SSL ciphersuites makes accommodation for export. As such, the USEE tag for SSL signifies to CSSM that the SSL library will carry the enforcement responsibilities. The tag for domestic use is provided for compatibility reasons. Any module not exported may carry the domestic privilege. The CSSM manifest does not require USEE tags to enforce their semantics. It is anticipated that other tags will be added as export policies change.

Additionally, CSSM and EMM components may use USEE tags to configure starting state. For example, the key recovery tag could instruct CSSM to set an internal switch that prevents use of attach handles that have not been sanitized through the key recovery module. Likewise, the domestic tag could indicate that the CSSM will not enforce export exemptions when deployed in the U.S.

A service module can also check the application manifest directly by retrieving its manifest from the MDS and verifying the signature.

6.3.6 Module-Granted Use Exemptions

Service module vendors can choose to provide enhanced services to selected applications or classes of applications. A module-defined use policy is in addition to the general CSSM integrity policy. Categories of enhanced services are defined as use exemptions. CSSM_USEE_TAG declares the currently defined set of exemption classes. These focus primarily on exemptions for using cryptographic services. New exemption classes can be defined in association with any category of security services.

Service providers should record the exemptions they grant by listing them in an appropriate MDS relation. Currently, Cryptographic Service Providers (CSPs) advertise their exemptions by listing them in the UseeTag attribute of the MDS CSP Primary relation. This is for information only. The verifiable authorization to grant the exemption must be recorded in the service module's signed manifest credentials. If a module grants exemptions, then the module's signed manifest must include a manifest attribute attesting to this authority. The manifest attribute for currently defined exemptions is a name-value pair with name CDSA_USEE. The associated value is a string of 4-byte base-64 encoded numbers in big-endian order. Encoded 4-byte words are separated by colons. These numbers are the same numbers defined for the lower order word of CSSM_PRIVILEGE. An example of CDSA_USEE tag in the manifest (which corresponds to the base64 encoding of the CSSM_USEE_TAG) is:

```
CDSA_USEE: AAAAAg==:AAAABQ==:AAAAAw==
```

Applications can query MDS to retrieve the USEE Tags associated with any CSP.

Privileges which have numbers from the high order word of the CSSM_PRIVILEGE will use the manifest tag CDSA_PRIV, as opposed to the tag CDSA_USEE. Other than that exception, they are used exactly like the tags listed as CDSA_USEE*.

The CSSM_Init function needs to be called before calling any other CSSM interfaces. At the time of the first CSSM_Init call, a CSSM_PRIVILEGE_SCOPE and CSSM_KEY_HIERARCHY can be passed in. Based on the platform-specific implementation of CSSM, *CSSM_Init*() will succeed or return an error if the privilege scope is not supported. The Caller has to introduce itself to the CSSM framework for requesting privileges using the *CSSM_Introduce,*() *CSSM_Init*() or *CSSM_ModuleAttach*() call. Once the caller is introduced to the CSSM, CSSM remembers the privilege and key hierarchy of the callers.

There are two ways for an Application to request a privilege:

1. An application can request a privilege while making a call to the "P" functions, for example, *EncryptDataP*, *DecryptDataP*, *WrapKeyP* or *UnwrapKeyP*, by passing the USEE_TAG as the privilege parameter. (If *SetPrivilege* is called and then a call is made to one of the "P" functions the USEE_TAG that is requested in the "P" function call will be sent to the service provider.)

2. An application can set the privilege for all of its calls to a service provider by calling *CSSM_Introduce*() and *CSSM_SetPrivilege*() functions. After calling *CSSM_SetPrivilege*(), CSSM will then forward the USEE_TAG that was set to the service provider when a call is made to *EncryptData*, *DecryptData*, etc.

The appropriate USEE tags that validates the requested exemption may be in either the signed manifest of the Application (stored in the MDS Object Directory relation), and/or in the signed manifest of the CSSM. Either the Application or the CSSM (or both) must have a signed manifest with the appropriate USEE tag.

6.3.7 Service Module Requirements if USEE Tags are Supported

- Service modules must be signed by recognized authorities who can grant the use of the USEE tags
- Service modules must verify that the direct calling module is a CSSM or an EMM which has been introduced by a CSSM.
- Service module may support multiple USEE privileges.
- Service module must verify that the USEE supplied by the caller matches one of its supported USEE tags.
- Service module must describe the supported USEE tags in a manifest attribute (for example, CDSA_USEE).
- Use of crypto operations must be verified for each instance and must fall within the restrictions implied by the USEE tag.
- Every module comprising the CSP, containing cryptographic interfaces, must be included in (one of) the CSP manifest(s).
- Other components that influence the operation of the CSP should be included in the manifest.
- Multi-service modules that also implement CSP functionality must enforce the rules defined here, but only for interfaces to cryptographic functions.
- A CSP must not allow any of its component modules' cryptographic interfaces to be directly called by an application module

6.3.8 Application Privilege

Applications that require privileged use of cryptographic technology must obtain the necessary approval from their controling authority. In the USA, in order to get export privilege, an Application that uses cryptographic technology based on CDSA must undergo a one-time review by the National Security Agency (NSA). Applications may use CDSA USEE tags to obtain that privilege.

6.3.9 Multiple CSSM Vendors Authenticating Same Application

Different CSSMs can cross-check the same application, even if it requires application credentials based on difference roots of trust. Figure 6-3 shows a complete set of credentials for an application module that can be cross-checked by different CSSM vendors. The credentials include three certificate chains. Each chain has a distinct root, and all chains sign the product. All three certificate chains are included in the credentials for this application module. When CSSM #1 attempts to verify the application module's credential, a verified certificate chain will be constructed from the application module's leaf certificate to the root certificate containing either public key PK2 or public key PK3, which are recognized as a points of trust by CSSM #1. Hence the application module's credentials will be successfully verified. CSSM #2 would verify the application module using public key PK5.

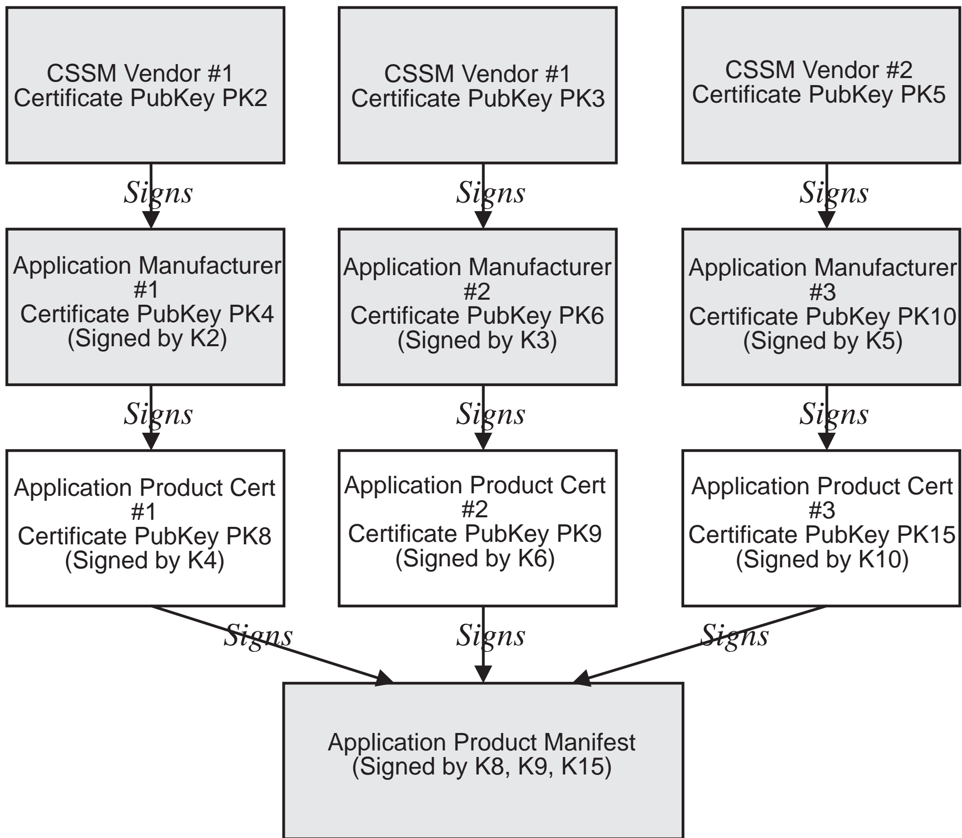


Figure 6-3 Multiple CSSM Vendors Authenticating Same Application

6.4 Data Structures for Core Services

6.4.1 CSSM_BOOL

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

```
typedef uint32 CSSM_BOOL

#define CSSM_FALSE (0)
#define CSSM_TRUE (!CSSM_FALSE)
```

Definitions

CSSM_TRUE
Indicates a true result or a true value.

CSSM_FALSE
Indicates a false result or a false value.

6.4.2 CSSM_RETURN

This data type is returned by most CDSA functions. The permitted values include:

- CSSM_OK all error codes defined in this specification
- module-specific error codes defined and used by a specific service provider module

```
typedef uint32 CSSM_RETURN;

#define CSSM_OK (0)
```

Definitions

CSSM_OK
Indicates operation was successful

All other values
Indicates the operation was unsuccessful and identifies the specific, detected error that resulted in the failure. Specific error values are defined for each function.

6.4.3 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];
```

6.4.4 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 security service 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;
```

Definitions

Length
Length of the data buffer in bytes.

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

6.4.5 CSSM_GUID

This structure designates a global unique identifier (GUID) that distinguishes one security service 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;
```

Definitions

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.

6.4.6 CSSM_KEY_HIERARCHY

The CSSM_KEY_HIERARCHY is a bitmask of values defining classes of key hierarchies from which CDSA components may be signed. Components may be signed by a single key hierarchy, multiple hierarchies or not signed. A bitmask is used to indicate which hierarchies are important.

```
typedef uint32 CSSM_BITMASK;
typedef CSSM_BITMASK CSSM_KEY_HIERARCHY;
#define CSSM_KEY_HIERARCHY_NONE (0)
#define CSSM_KEY_HIERARCHY_INTEG (1)
#define CSSM_KEY_HIERARCHY_EXPORT (2)
```

Definitions

CSSM_KEY_HIERARCHY_NONE
: This value indicates the key hierarchy is unspecified. Unspecified key hierarchy values usually indicate the intended hierarchy will be inferred. For modules without a manifest, CSSM_KEY_HERARCHY_NONE may be supplied.

CSSM_KEY_HIERARCHY_INTEG
: This value indicates the embedded key used for integrity checking is associated with a public key infrastructure that vouches for the integrity of executable modules. This flag helps the CSSM identify which embedded key will be used to determine the root of trust when performing cross-check operations.

CSSM_KEY_HIERARCHY_EXPORT
: This value indicates the embedded key used for validating export privileges should be used to perform cross-check operations.

6.4.7 CSSM_PVC_MODE

Pointer validation checking policy can be configured globally for the CSSM when *CSSM_Init*() is called. See *CSSM_Init* on page 98 for additional information. The possible values are as follows:

Value	Description
0	PVC validation is not performed
1	PVC validation is performed on application modules
2	PVC validation is performed on service provider modules
3	Both types of PVC validations are performed

```
typedef CSSM_BITMASK CSSM_PVC_MODE;

#define CSSM_PVC_NONE (0)
#define CSSM_PVC_APP (1)
#define CSSM_PVC_SP (2)
```

6.4.8 CSSM_PRIVILEGE_SCOPE

The privilege scope identifies whether the privilege specified using the *CSSM_SetPrivilege*( ) call applies to the entire process, CSSM_PRIVILEGE_SCOPE_PROCESS, or to the current thread, CSSM_PRIVILEGE_SCOPE_THREAD. The scope will determine the conditions under which other calls to *CSSM_SetPrivilege*( ) will be visible by other threads and libraries.

```
typedef uint32 CSSM_PRIVILEGE_SCOPE;

#define CSSM_PRIVILEGE_SCOPE_NONE (0)
#define CSSM_PRIVILEGE_SCOPE_PROCESS (1)
#define CSSM_PRIVILEGE_SCOPE_THREAD (2)
```

6.4.9 CSSM_VERSION

This structure is used to represent the version of CDSA components. The major number begins at 1 and is incremented by 1 for each major release.

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

Definitions

Major
The major version number of the component.

Minor
The minor version number of the component.

The minor number uses two digits to represent minor releases and revisions. The revision number is represented in the least significant digit. The remaining more significant digits represent minor numbers. The first release has the value of zero. There can be 9 subsequent releases then the minor number must be incremented. For example, the minor number for the very first release of a product would be represented as "00". Subsequent releases would be "01", "02", "03" etc., to "09". If version number changes at each release then the minor numbers would increment from "00", "10", "20" etc., to "90". A minor version of 10 release 1 would be "100".

Examples:

1.0.0 - Major: 1 Minor: 0
1.1.0 - Major: 1 Minor: 10
1.1.1 - Major: 1 Minor: 11
1.24 - Major: 1 Minor: 240
1.24.9 - Major: 1 Minor: 249
1.24.38 - not possible

6.4.10 CSSM_SUBSERVICE_UID

This structure uniquely identifies a set of behaviors within a subservice within a CSSM security service module.

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

Definitions

Guid
A unique identifier for a CSSM security service module.

Version
The version of the security service module.

SubserviceId
An identifier for the subservice within the security service module.

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

6.4.11 CSSM_HANDLE

A unique identifier for an object managed by CSSM or by an security service module.

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

6.4.12 CSSM_LONG_HANDLE

A unique, long identifier for an object managed by CSSM or by a service module.

```
#if defined (WIN32)
typedef unsigned __int64 uint64; /* MSVC++ 5 declaration */
#else
typedef unsigned long long uint64; /* gcc 2.7.2 declaration */
#endif
typedef uint64 CSSM_LONG_HANDLE, *CSSM_LONG_HANDLE_PTR;
```

6.4.13 CSSM_MODULE_HANDLE

A unique identifier for an attached service provider module.

```
typedef CSSM_HANDLE CSSM_MODULE_HANDLE, *CSSM_MODULE_HANDLE_PTR;
```

6.4.14 CSSM_MODULE_EVENT

This enumeration defines the event types that can be raised by any service module. Callers can define event handling callback functions of type *CSSM_API_ModuleEventHandler* to receive and manage these events. Callback functions are registered using the *CSSM_ModuleLoad*() function. Example events include insertion and removal of a subservice. Events are asynchronous.

```
typedef enum cssm_module_event {
    CSSM_NOTIFY_INSERT = 1,
    CSSM_NOTIFY_REMOVE = 2,
    CSSM_NOTIFY_FAULT = 3,
} CSSM_MODULE_EVENT, *CSSM_MODULE_EVENT_PTR;
```

6.4.15 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_AC (0x20)
#define CSSM_SERVICE_KR (0x40)
```

6.4.16 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;
```

6.4.17 CSSM_API_ModuleEventHandler

This defines the event handler interface that an application must define and implement to receive asynchronous notification of events such as insertion or removal of a hardware service module, or a fault detected by the service module. The event handler is registered with CSSM as part of the *CSSM_ModuleLoad*() function. This is the caller's single event handle for all general module events over all of the caller's attach-sessions with the loaded module. This general event notification is processed through CSSM.

The event handler defined by the CSSM_API_ModuleEventHandler structure must be executed outside the thread context that registered the handler. The handler can queue the event results and schedule a reader thread to service the queue. Applications should use *AppNotifyCallbackCtx* to track thread-id, the head of the queue, and other state information necessary to service the queue and process the events. The event handler must not issue calls through the CSSM API. These circular calls can result in deadlock in numerous situations, hence the event handler must be implemented without using CSSM services.

The CSSM_API_ModuleEventHandler can be invoked multiple times in response to a single event (such as the insertion of a smartcard). The handler and the calling application must track receipt of event notifications and ignore duplicates. Other events (such as the removal of a smartcard) can result in a single event notification.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_API_ModuleEventHandler)
    (const CSSM_GUID *ModuleGuid,
    void* AppNotifyCallbackCtx,
    uint32 SubserviceId,
    CSSM_SERVICE_TYPE ServiceType,
    CSSM_MODULE_EVENT EventType);
```

Definitions

ModuleGuid (input)
The GUID of the service module raising the event.

AppNotifyCallbackCtx (input)
The application context specified during *CSSM_ModuleLoad*().

SubserviceId (input)
The *subserviceId* of the service module raising the event.

ServiceType (input)
The service mask of the module raising the event.

EventType (input)
The CSSM_MODULE_EVENT that has occurred.

6.4.18 CSSM_ATTACH_FLAGS

This bitmask is used to specify the behaivor of the service provider being attached.

```
typedef uint32 CSSM_ATTACH_FLAGS;
#define CSSM_ATTACH_READ_ONLY (0x00000001)
```

Definitions

CSSM_ATTACH_READ_ONLY
Causes the service provider to block all creation or deletion of persistent resources using the attach handle. An application may later override this flag using module specific facilities.

6.4.19 CSSM_PRIVILEGE

The USE exemption tag lists the exemption categories for cryptographic services. These categories can be supported by a cryptographic service provider. An application can be authorized to receive these services. The granting of one or more exemptions is recorded in an application's signed manifest credentials Cryptographic service providers record their ability to honor such exemptions in their signed manifest credential. Use exemptions can be defined to other cateogries of security services.

The representation of privileges uses two 4-byte words (or an 8 byte word if the platform supports 64-bit words). The high-order 4-byte word is reserved for non-export related privileges and vendor-specific privileges. Of the low-order 4-byte word, the lower byte is used to represent United States export policy, also known as USEE tags. The remaining number space (3 bytes) is reserved for future expansion.

Low-Order Word

Byte 3	Byte 2	Byte 1	Byte 0
Reserved	Reserved	Reserved	USEE Tag

typedef uint32 CSSM_USEE_TAG;

There are U.S. export exemptions for strong encryption financial, medical, insurance, key recovery, key exchange and authentication applications. SSL is cited as a special case exemption in that SSL ciphersuites make accommodation for U.S. export rules. As such, the USEE tag for SSL signifies to CSSM that the SSL library will carry the enforcement responsibilities. The tag for domestic use is provided for compatibility reasons. Any U.S. module not exported may carry the domestic privilege. The CSSM manifest does not require USEE tags to enforce their semantics. It is anticipated that other tags will be added as U.S. export policies change and other types of exemptions are identified.

The high-order 4 bytes of privilege vector allow for vendor specific privilege values. The high order bit (31) is set (1). The lower 31 bits are reserved for vendor-specific privileges. When unset (0), the lower 31 bits (0-30) are interpreted as CDSA privileges not related to export. No CDSA non-export privileges are defined at the time of publishing this Technical Standard.

High-Order Word

Byte 7	Byte 6	Byte 5	Byte 4
Other	Other	Other	Other
Privilege	Privilege	Privilege	Privilege

Non-export privilege range: (0x00000000 - 0x7FFFFFFF)
Vendor specific range: (0x80000000 - 0xFFFFFFFF)

```
typedef uint64 CSSM_PRIVILEGE;

#define CSSM_USEE_LAST (0xFF)

#define CSSM_USEE_NONE (0)
#define CSSM_USEE_DOMESTIC (1)
#define CSSM_USEE_FINANCIAL (2)
#define CSSM_USEE_KRLE (3)
#define CSSM_USEE_KRENT (4)
#define CSSM_USEE_SSL (5)
#define CSSM_USEE_AUTHENTICATION (6)
#define CSSM_USEE_KEYEXCH (7)
#define CSSM_USEE_MEDICAL (8)
#define CSSM_USEE_INSURANCE (9)
#define CSSM_USEE_WEAK (10)
```

Definitions

CSSM_USEE_NONE
The policy implemented is unspecified.

CSSM_USEE_DOMESTIC
No restrictions are placed on the use of cryptography. Currently, this privilege cannot be given to general purpose cryptographic services manufactured in the United States.

CSSM_USEE_SSL
This exemption indicates that the application is enforcing U.S. export policy in accordance with mechanisms defined by SSL specifications.

CSSM_USEE_KRLE and CSSM_USEE_KRENT
The KRLE and KRENT tags indicate U.S. export permission is tied to key escrow using either law enforcement agencies or a business entity. The Key Recovery service provider must be used in conjunction with CSSM to gain access to strong crypto. Manifests containing key recovery privileges indicates the module is trusted to abide by the rules of key recovery. A user of CDSA may be exempt from key recovery if the privilege is granted to the calling module. In other words, the calling module is trusted to abide by U.S. export restrictions hence may be exempt from key recovery. Modules containing tags identifying use-based exemption, such as FINANCIAL, can avoid key recovery when operating in the FINANCIAL mode.

CSSM_USEE_WEAK
This flag identifies US export policy controlling minimum grade crypto for general purpose use.

Other USEE Tags
The remaining USEE tags map to exemptions granted in the U.S. export regulations.

The CSSM_USEE_TAG values represent exemptions to blanket export policy. Values map to sections in a countries export regulations that permits use of stronger cryptography if the application and constituant parts abide by the stipulations. Applications and their components can be authorized to receive privileges entitling them to stronger cryptographic services.

At the time of publication, the only country known to provide commodity class exemptions to export regulations is the United States of America. The USEE tags defined herein apply to United States export policy. Subsequent export exemptions may be added to the specification, the country to which the tag applies should be included in the documentation.

Application components, CDSA service providers and CSSM may be entrusted to cooperate in the enforcement of a set of policies. The signed manifest will contain a list of privileges the component implements/enforces. The manifest attribute for currently defined exemptions is a name-value pair with name CDSA_USEE. The associated value is a string of base-64 encoded numbers separated by colons.

6.4.20 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;
```

6.4.21 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;
```

6.4.22 CSSM_NET_PROTOCOL

This data type defines the application-level protocols that could be supported by a Data Storage Library Module that communicates with service-based storage and directory services.

```
typedef  uint32 CSSM_NET_PROTOCOL;
    #define CSSM_NET_PROTO_NONE (0)    /* local */
    #define CSSM_NET_PROTO_CUSTOM (1)  /* proprietary implementation */
    #define CSSM_NET_PROTO_UNSPECIFIED (2) /* implementation default */
    #define CSSM_NET_PROTO_LDAP (3)    /* light weight directory access */
                                       /* protocol */
    #define CSSM_NET_PROTO_LDAPS (4)   /* ldap/ssl where SSL initiates  */
                                       /* the connection */
    #define CSSM_NET_PROTO_LDAPNS (5)  /* ldap where ldap negotiates */
                                       /* an SSL session */
    #define CSSM_NET_PROTO_X500DAP (6) /* X.500 Directory access protocol */
    #define CSSM_NET_PROTO_FTP (7)     /* ftp for cert/crl fetch */
    #define CSSM_NET_PROTO_FTPS (8)    /* ftp/ssl/tls where SSL/TLS   */
                                       /* initiates the connection */
    #define CSSM_NET_PROTO_OCSP (9)    /* online certificate status */
                                       /* protocol */
    #define CSSM_NET_PROTO_CMP (10)    /* the cert request protocol */
                                       /* in PKIX3 */
    #define CSSM_NET_PROTO_CMPS (11)   /* The ssl/tls derivative of CMP */
```

6.4.23 CSSM_CALLBACK

An application uses this data type to request that a security service module call back into the application for certain cryptographic information.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_CALLBACK)
    (CSSM_DATA_PTR OutData, void *CallerCtx);
```

Definitions

OutData (output)
The opaque structured value to be returned by the callback function. The buffer is allocated by the caller and filled in by the callback function. The callback sets the *Length* field to the length of the value written to the buffer.

CallerCtx (input)
A generic pointer to the context information being returned to its originator. This context was created as state information by a caller, passed as input to another function, returned by that function to the original caller through the callback function. The callback handler uses information in the context to properly handle the callback.

6.4.24 CSSM_CRYPTO_DATA

This data structure is used to encapsulate cryptographic information passed from the application to a service provider module via a callback function. Typical information passed by this callback mechanism includes a seed value for cryptographic operations.

```
typedef struct cssm_crypto_data {
    CSSM_DATA Param;
    CSSM_CALLBACK Callback;
    void *CallerCtx;
} CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR;
```

Definitions

Param
The CSSM_DATA structure containing the parameters to be passed as input when invoking the callback function, and the size (in bytes) of those parameters.

Callback
An optional callback routine for a service to invoke to obtain information from the original caller.

CallerCtx
A generic pointer to the context information that should be input to the callback handler when invoking the callback function. The callback handler uses this information to properly handle the callback.

6.4.25 CSSM_WORDID_TYPE

This data type defines common symbols for integer values used to represent sample types, ACL subject types, and authorization tag values. The symbol name can be transported from system to system. The corresponding constant is used for local computation.

```
typedef sint32 CSSM_WORDID_TYPE;

#define CSSM_WORDID__UNK_ (-1)  /* not in dictionary */
#define CSSM_WORDID__NLU_ (0)   /* not yet looked up */

#define CSSM_WORDID__STAR_ (1)
#define CSSM_WORDID_A (2)
#define CSSM_WORDID_ACL (3)
#define CSSM_WORDID_ALPHA (4)
#define CSSM_WORDID_B (5)
#define CSSM_WORDID_BER (6)
#define CSSM_WORDID_BINARY (7)
#define CSSM_WORDID_BIOMETRIC (8)
#define CSSM_WORDID_C (9)
#define CSSM_WORDID_CANCELED (10)
#define CSSM_WORDID_CERT (11)
#define CSSM_WORDID_COMMENT (12)
#define CSSM_WORDID_CRL (13)
#define CSSM_WORDID_CUSTOM (14)
#define CSSM_WORDID_D (15)
#define CSSM_WORDID_DATE (16)
#define CSSM_WORDID_DB_DELETE (17)
#define CSSM_WORDID_DB_EXEC_STORED_QUERY (18)
#define CSSM_WORDID_DB_INSERT (19)
#define CSSM_WORDID_DB_MODIFY (20)
#define CSSM_WORDID_DB_READ (21)
#define CSSM_WORDID_DBS_CREATE (22)
#define CSSM_WORDID_DBS_DELETE (23)
#define CSSM_WORDID_DECRYPT (24)
#define CSSM_WORDID_DELETE (25)
#define CSSM_WORDID_DELTA_CRL (26)
#define CSSM_WORDID_DER (27)
#define CSSM_WORDID_DERIVE (28)
#define CSSM_WORDID_DISPLAY (29)
#define CSSM_WORDID_DO (30)
#define CSSM_WORDID_DSA (31)
#define CSSM_WORDID_DSA_SHA1 (32)
#define CSSM_WORDID_E (33)
#define CSSM_WORDID_ELGAMAL (34)
#define CSSM_WORDID_ENCRYPT (35)
#define CSSM_WORDID_ENTRY (36)
#define CSSM_WORDID_EXPORT_CLEAR (37)
#define CSSM_WORDID_EXPORT_WRAPPED (38)
#define CSSM_WORDID_G (39)
#define CSSM_WORDID_GE (40)
#define CSSM_WORDID_GENKEY (41)
#define CSSM_WORDID_HASH (42)
```

```
#define CSSM_WORDID_HAVAL (43)
#define CSSM_WORDID_IBCHASH (44)
#define CSSM_WORDID_IMPORT_CLEAR (45)
#define CSSM_WORDID_IMPORT_WRAPPED (46)
#define CSSM_WORDID_INTEL (47)
#define CSSM_WORDID_ISSUER (48)
#define CSSM_WORDID_ISSUER_INFO (49)
#define CSSM_WORDID_K_OF_N (50)
#define CSSM_WORDID_KEA (51)
#define CSSM_WORDID_KEYHOLDER (52)
#define CSSM_WORDID_L (53)
#define CSSM_WORDID_LE (54)
#define CSSM_WORDID_LOGIN (55)
#define CSSM_WORDID_LOGIN_NAME (56)
#define CSSM_WORDID_MAC (57)
#define CSSM_WORDID_MD2 (58)
#define CSSM_WORDID_MD2WITHRSA (59)
#define CSSM_WORDID_MD4 (60)
#define CSSM_WORDID_MD5 (61)
#define CSSM_WORDID_MD5WITHRSA (62)
#define CSSM_WORDID_N (63)
#define CSSM_WORDID_NAME (64)
#define CSSM_WORDID_NDR (65)
#define CSSM_WORDID_NHASH (66)
#define CSSM_WORDID_NOT_AFTER (67)
#define CSSM_WORDID_NOT_BEFORE (68)
#define CSSM_WORDID_NULL (69)
#define CSSM_WORDID_NUMERIC (70)
#define CSSM_WORDID_OBJECT_HASH (71)
#define CSSM_WORDID_ONE_TIME (72)
#define CSSM_WORDID_ONLINE (73)
#define CSSM_WORDID_OWNER (74)
#define CSSM_WORDID_P (75)
#define CSSM_WORDID_PAM_NAME (76)
#define CSSM_WORDID_PASSWORD (77)
#define CSSM_WORDID_PGP (78)
#define CSSM_WORDID_PREFIX (79)
#define CSSM_WORDID_PRIVATE_KEY (80)
#define CSSM_WORDID_PROMPTED_PASSWORD (81)
#define CSSM_WORDID_PROPAGATE (82)
#define CSSM_WORDID_PROTECTED_BIOMETRIC (83)
#define CSSM_WORDID_PROTECTED_PASSWORD (84)
#define CSSM_WORDID_PROTECTED_PIN (85)
#define CSSM_WORDID_PUBLIC_KEY (86)
#define CSSM_WORDID_PUBLIC_KEY_FROM_CERT (87)
#define CSSM_WORDID_Q (88)
#define CSSM_WORDID_RANGE (89)
#define CSSM_WORDID_REVAL (90)
#define CSSM_WORDID_RIPEMAC (91)
#define CSSM_WORDID_RIPEMD (92)
#define CSSM_WORDID_RIPEMD160 (93)
#define CSSM_WORDID_RSA (94)
```

```
#define CSSM_WORDID_RSA_ISO9796 (95)
#define CSSM_WORDID_RSA_PKCS (96)
#define CSSM_WORDID_RSA_PKCS_MD5 (97)
#define CSSM_WORDID_RSA_PKCS_SHA1 (98)
#define CSSM_WORDID_RSA_PKCS1 (99)
#define CSSM_WORDID_RSA_PKCS1_MD5 (100)
#define CSSM_WORDID_RSA_PKCS1_SHA1 (101)
#define CSSM_WORDID_RSA_PKCS1_SIG (102)
#define CSSM_WORDID_RSA_RAW (103)
#define CSSM_WORDID_SDSIV1 (104)
#define CSSM_WORDID_SEQUENCE (105)
#define CSSM_WORDID_SET (106)
#define CSSM_WORDID_SEXPR (107)
#define CSSM_WORDID_SHA1 (108)
#define CSSM_WORDID_SHA1WITHDSA (109)
#define CSSM_WORDID_SHA1WITHECDSA (110)
#define CSSM_WORDID_SHA1WITHRSA (111)
#define CSSM_WORDID_SIGN (112)
#define CSSM_WORDID_SIGNATURE (113)
#define CSSM_WORDID_SIGNED_NONCE (114)
#define CSSM_WORDID_SIGNED_SECRET (115)
#define CSSM_WORDID_SPKI (116)
#define CSSM_WORDID_SUBJECT (117)
#define CSSM_WORDID_SUBJECT_INFO (118)
#define CSSM_WORDID_TAG (119)
#define CSSM_WORDID_THRESHOLD (120)
#define CSSM_WORDID_TIME (121)
#define CSSM_WORDID_URI (122)
#define CSSM_WORDID_VERSION (123)
#define CSSM_WORDID_X509_ATTRIBUTE (124)
#define CSSM_WORDID_X509V1 (125)
#define CSSM_WORDID_X509V2 (126)
#define CSSM_WORDID_X509V3 (127)
#define CSSM_WORDID_X9_ATTRIBUTE (128)

#define CSSM_WORDID_VENDOR_START (0x00010000)
#define CSSM_WORDID_VENDOR_END (0xFFFF0000)
```

Definitions

Only those letters that are used in key or signature definitions are defined:

- E, N for RSA public
- P, Q, A, B, C, D for RSA private
- G, P, Q, X, Y for DSA
- R, S for DSA signatures

6.4.26 CSSM_LIST_ELEMENT_TYPE

This data element defines the type of a list element. There are only two possibilities: a byte string or a sub-list.

```
typedef uint32 CSSM_LIST_ELEMENT_TYPE, *CSSM_LIST_ELEMENT_TYPE_PTR;

#define CSSM_LIST_ELEMENT_DATUM   (0x00)
#define CSSM_LIST_ELEMENT_SUBLIST (0x01)
#define CSSM_LIST_ELEMENT_WORDID  (0x02)
```

6.4.27 CSSM_LIST_TYPE

This extensible list defines the type of linked lists used to contain the parsed elements of a certificate.

```
typedef uint32 CSSM_LIST_TYPE,  * CSSM_LIST_TYPE_PTR;

#define CSSM_LIST_TYPE_UNKNOWN  (0)
#define CSSM_LIST_TYPE_CUSTOM   (1)
#define CSSM_LIST_TYPE_SEXPR    (2)
```

6.4.28 CSSM_LIST

This structure defines a linked list. It is defined with the intention of holding parsed certificates. The data structure definition is general and can be used for other lists as indicated in the *ListType*.

```
typedef struct cssm_list_element *CSSM_LIST_ELEMENT_PTR;

typedef struct cssm_list {
    CSSM_LIST_TYPE ListType;      /* type of this list */
    CSSM_LIST_ELEMENT_PTR Head;   /* head of the list */
    CSSM_LIST_ELEMENT_PTR Tail;   /* tail of the list */
} CSSM_LIST, *CSSM_LIST_PTR;
```

Definitions

ListType
The type of linked list whose root is contained in this structure.

Head
A pointer to the CSSM_LIST_ELEMENT that forms the head of the linked list.

Tail
A pointer to the CSSM_LIST_ELEMENT that forms the tail of the linked list.

6.4.29 CSSM_LIST_ELEMENT

This structure defines a single element in a linked list. This structure is defined with the intention of holding parsed certificate elements, but the data structure definition is general and can be used for other parsed or decomposed objects. Each list element contains an immediate data item or a sub-list. Immediate data values can be represented by a *WordID*, or a Word of type CSSM_DATA, or both. When both a *WordID* and a CSSM_DATA word are provided, the two values should be related by a Dictionary that binds the two values under the Dictionary mapping.

```
typedef struct cssm_list_element {
    struct cssm_list_element *NextElement;  /* next list element */

    CSSM_WORDID_TYPE WordID;     /* integer identifier associated */
                                 /* with a Word value */
    CSSM_LIST_ELEMENT_TYPE  ElementType;
    union {
        CSSM_LIST Sublist;       /* sublist */
        CSSM_DATA Word;          /* a byte-string */
    } Element;
} CSSM_LIST_ELEMENT;
```

Definitions

NextElement
A pointer to the next CSSM_LIST_ELEMENT in the list.

WordID
An identifier associated with the Word or Sublist in this element. Valid values for *WordID* are defined for each *ListType*. If the list is of type CSSM_LIST_TYPE_SEXPR, then each list or sub-list must start with a pre-defined type-indicator. The *WordID* is an integer representation of a predefined dictionary word. The meaning of *WordID* depends on the *ElementType* as defined in the following table:

Element Type	Meaning of WordID
CSSM_LIST_ELEMENT_DATUM	WordID is associated with the value in Word. If the pre-defined dictionary does not contain a mapping from the value of Word to a WordID, then WordID is zero.
CSSM_LIST_ELEMENT_SUBLIST	WordID is associated with the Word that begins the Sublist. If the pre-defined dictionary does not contain a mapping from the value of Word to a WordID, then WordID is zero.
CSSM_LIST_ELEMENT_WORDID	WordID is the value of the list element. Word and Sublist are empty.

ElementType
Specifies whether the Element is a datum or a sub-list.

Element
The list element can contain zero values or one value. If a value is present it is of one of two types:

- An immediate data item
  The data value is stored in a single byte-array. The length of the array and a reference to the array are contained a CSSM_DATA structure.
- Another list
  The sub-list is defined by the list type and two references, one to the head of the sub-list and another to the tail of the sub-list.

6.4.30 CSSM_TUPLE

This structure defines a full 5-tuple, representing one certificate (or a fragment of a more complex certificate).

```
typedef struct {                /* 5-tuple definition */
    CSSM_LIST Issuer;           /* issuer, or empty if ACL */
    CSSM_LIST Subject;          /* subject */
    CSSM_BOOL Delegate;         /* permission to delegate */
    CSSM_LIST AuthorizationTag; /* authorization field */
    CSSM_LIST ValidityPeriod;   /* validity information (dates) */
} CSSM_TUPLE, *CSSM_TUPLE_PTR;
```

Definitions

Issuer
A CSSM_LIST structure containing a parse list representing the entity that issued the tuple. If this tuple represents an ACL, the CSSM_LIST structure must contain a NULL list.

Subject
A CSSM_LIST structure containing a parse list representing the subject entity for this tuple.

Delegate
A CSSM_BOOL value indicating whether the *Subject* entity is permitted to delegate the authorizations granted by *AuthorizationTag* during the *ValidityPeriod* to other entities.

AuthorizationTag
A CSSM_LIST structure containing a parse list representing the authorizations granted to the *Subject* by the *Issuer* for *ValidityPeriod* time. The meaning of the field is based on an application domain.

ValidityPeriod
A CSSM_LIST structure containing a parse list representing the time period for which the *Issuer* has granted authorizations to the *Subject*.

6.4.31 CSSM_TUPLEGROUP

This data structure contains a set of CSSM_TUPLE structures. The tuples are grouped from the purpose of input-to or output-from a function or service.

```
typedef struct cssm_tuplegroup {
    uint32 NumberOfTuples;
    CSSM_TUPLE_PTR Tuples;
} CSSM_TUPLEGROUP, *CSSM_TUPLEGROUP_PTR;
```

Definitions

NumberOfTuples
The number of entries in the array *Tuples*.

Tuples
A pointer to an ordered array of CSSM_TUPLE structures. Each structure contains a single, nested tuple.

6.4.32 CSSM_SAMPLE_TYPE

This data type defines integer values identifying the types of samples a caller can present to a service provider. Samples are used to authenticate the caller and to verify authorization to access a resource. Authentication is typically based on zero or more samples and zero or more certificates.

```
typedef CSSM_WORDID_TYPE CSSM_SAMPLE_TYPE;

#define CSSM_SAMPLE_TYPE_PASSWORD CSSM_WORDID_PASSWORD
#define CSSM_SAMPLE_TYPE_PROTECTED_PASSWORD CSSM_WORDID_PROTECTED_PASSWORD
#define CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD CSSM_WORDID_PROMPTED_PASSWORD
#define CSSM_SAMPLE_TYPE_SIGNED_NONCE CSSM_WORDID_SIGNED_NONCE
#define CSSM_SAMPLE_TYPE_SIGNED_SECRET CSSM_WORDID_SIGNED_SECRET
#define CSSM_SAMPLE_TYPE_BIOMETRIC CSSM_WORDID_BIOMETRIC
#define CSSM_SAMPLE_TYPE_PROTECTED_BIOMETRIC CSSM_WORDID_PROTECTED_BIOMETRIC
#define CSSM_SAMPLE_TYPE_THRESHOLD CSSM_WORDID_THRESHOLD
```

CSSM_SAMPLE_TYPE	General Description
CSSM_SAMPLE_TYPE_PASSWORD	The sample is a password or passphrase value.
CSSM_SAMPLE_TYPE_HASHED_PASSWORD	The sample is a password or passphrase value. The service provider receiving this value computes its hash and uses the hash value as the gathered sample.
CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD	The sample is a password or passphrase with an associated prompt value that can be present to an application when requesting this sample.
CSSM_SAMPLE_TYPE_PROTECTED_PASSWORD	The sample is a password or passphrase gathered indirectly from a service provider operating a protected data acquisition path to acquire and verify this sample value.

CSSM_SAMPLE_TYPE_SIGNED_NONCE	A service provider presents a nonce to an application. The application signs the nonce with its private key. The resulting signed nonce is the sample value.
CSSM_SAMPLE_TYPE_SIGNED_SECRET	The application and the service provider have a shared secret. The application signs the shared secret with its private key. The signed secret is the sample value.
CSSM_SAMPLE_TYPE_BIOMETRIC	The sample is a biometric data sample provided by the caller and passed through the API.
CSSM_SAMPLE_TYPE_PROMPTED_BIOMETRIC	The sample is a biometric data sample with an associated prompt value that can be present to an application when requesting this sample.
CSSM_SAMPLE_TYPE_PROTECTED_BIOMETRIC	The sample is a biometric data sample, gathered indirectly from a service provider operating a protected data acquisition path to acquire and verify this sample value.
CSSM_SAMPLE_TYPE_THRESHOLD	The sample is a choice of k-of-n samples. Each option is one of the defined sample types.

6.4.33 CSSM_SAMPLE

This structure contains a typed sample and a Globally Unique ID identifying a service provider module that can verify, authenticate, and process the sample value. An application uses this structure to provide actual sample values or references to sample values to a service provider. An application uses this structure in two situations:

- As an input parameter to a CDSA security service API
- As a response to a Challenge callback from a service provider

When using this structure as an input parameter to a CDSA function, the structure can:

- Contain the sample type and sample value as immediate data values
- Indicate the type of sample the application would like to present to the service provider, and request that the service provider invoke an application-implemented callback function to gather the sample value
- Indicate the type and third-party source of the sample value, and request that the service provider acquire the sample through a protected path with the third party.

When using this structure as a response to a Challenge callback from a service provider, the structure must contain the sample type and sample value as immediate data values.

```
typedef struct cssm_sample {
    CSSM_LIST TypedSample;
    const CSSM_SUBSERVICE_UID *Verifier;
} CSSM_SAMPLE, *CSSM_SAMPLE_PTR;
```

Definitions

TypedSample

A CSSM_LIST structure containing one or two elements. The first element is a sample type. The second element is an optional sample value. The immediate sample value in the list can be one of many types. Samples of type CSSM_SAMPLE_TYPE_PROTECTED_xyz can not be provided as an immediate value in this structure. A sample of this type is acquired through a protected path managed by the specified *Verifier*. In this case, the sample type is the only element in this list. Also the *Verifier* must be specified. All other sample types can be acquired by invoking a callback function. In this case the sample is not provided as an immediate value and this list contains a single element, the sample type. The following table defines the CSSM_LIST elements that can be presented under each valid use scenario.

CSSM_SAMPLE_ TYPE_PASSWORD	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length	No
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_PASSWORD	No
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length	No
CSSM_SAMPLE_ TYPE_HASHED_ PASSWORD	**CSSM_LIST Contents**	**Verifier Required?**
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_HASHED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length	No
CSSM_SAMPLE_ TYPE_PROMPTED_ PASSWORD	**CSSM_LIST Contents**	**Verifier Required?**
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length	No
CSSM_SAMPLE_ TYPE_PROTECTED_ PASSWORD	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Protected Path Acquisition Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_PROTECTED_PASSWORD	Optional
CSSM_SAMPLE_ TYPE_SIGNED_ NONCE	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_NONCE	Yes

Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_NONCE Second element: Word is a CSSM_DATA structure referencing a < signed nonce > value of specified length	Yes
CSSM_SAMPLE_ TYPE_SIGNED_ SECRET	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_SECRET Second element: Word is a CSSM_DATA structure referencing a < signed secret > value of specified length	Yes
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_SECRET	Yes
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_SECRET Second element: Word is a CSSM_DATA structure referencing a < signed secret > value of specified length	Yes
CSSM_SAMPLE_ TYPE_BIOMETRIC	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a < biometric template > value of specified length	Yes
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_BIOMETRIC	Yes
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a < biometric template > value of specified length	Yes
CSSM_SAMPLE_ TYPE_PROMPTED_ BIOMETRIC	**CSSM_LIST Contents**	**Verifier Required?**
Response to a Challenge callback	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a < biometric template > value of specified length	Yes
CSSM_SAMPLE_ TYPE_PROTECTED_ BIOMETRIC	**CSSM_LIST Contents**	**Verifier Required?**
Immediate Type and Protected Path Acquisition Form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_PROTECTED_BIOMETRIC	Yes

Verifier

A pointer to the persistent identifier of a service provider module that is capable of identifying, authenticating or verifying the sample. This value is required in all cases except the immediate value password.

6.4.34 CSSM_SAMPLEGROUP

This structure contains a group of related samples.

```
typedef struct cssm_samplegroup {
    uint32 NumberOfSamples;
    const CSSM_SAMPLE *Samples;
} CSSM_SAMPLEGROUP, *CSSM_SAMPLEGROUP_PTR;
```

Definitions

NumberOfSamples
The number of samples in the array Samples.

Samples
A pointer to an array of CSSM_SAMPLE structures.

6.4.35 CSSM_CHALLENGE_CALLBACK

This type defines the form of the callback function a service provider module must use to challenge a requester to acquire samples during an authentication and authorization verification procedure.

```
typedef CSSM_RETURN (CSSMAPI * CSSM_CHALLENGE_CALLBACK)
    (const CSSM_LIST *Challenge,
    CSSM_SAMPLEGROUP_PTR Response,
    void *CallerCtx,
    const CSSM_MEMORY_FUNCS *MemFuncs);
```

Definitions

Challenge (input)
A pointer to a CSSM_LIST structure containing a typed challenge. The type is specified as the first element in the list. Samples acquired through protected path sources can not be gathered through the callback mechanism, hence samples of type CSSM_SAMPLE_TYPE_PROTECTED_xyz never occur in a challenge list. If more than one type of sample is applicable, then the first list element indicates the type CSSM_SAMPLE_TYPE_THRESHOLD. The first list element is the type of sample the service provider is challenging the original requester to present. The second element provides optional information to assist the requester in replying to the challenge. For example, when requesting a sample of type CSSM_SAMPLE_TYPE_SIGNED_NONCE, the service provider must provide a nonce to be signed by the requester. The list of possible challenge types, their parameters, and associated responses are summarized in a table below.

Response (output)
A pointer to a CSSM_SAMPLEGROUP structure containing the samples returned in response to the challenge. Each sample includes a CSSM_LIST structure and an optional UID for a *Verifier*. The first element of the list is the type of the sample. The returned sample types must be appropriate for the *Challenge*. The list of possible challenge types, their parameters, and associated responses are summarized in a table below.

CallerCtx (input)
A generic pointer to context information that was provided by the original requester and is being returned to its originator.

MemFuncs (input)

A pointer to a CSSM_MEMORY_FUNCS structure. The challenge callback must use the functions in this structure to allocate memory returned to the challenger in the *Response* structure.

CSSM_SAMPLE_ TYPE_PASSWORD	**CSSM_LIST Contents**
Challenge form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_PASSWORD
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
CSSM_SAMPLE_ TYPE_HASHED_ PASSWORD	**CSSM_LIST Contents**
Challenge form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_HASHED_PASSWORD
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_HASHED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
T&	
cb cb	
l l.	
CSSM_SAMPLE_ TYPE_PROMPTED_ PASSWORD	CSSM_LIST Contents
Challenge form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <prompt string> value of specified length
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
CSSM_SAMPLE_ TYPE_SIGNED_ NONCE	**CSSM_LIST Contents**
Challenge form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_NONCE Second element: Word is a CSSM_DATA structure referencing a <nonce to be signed> value of specified length
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_NONCE Second element: Word is a CSSM_DATA structure referencing a < signed nonce > value of specified length
CSSM_SAMPLE_ TYPE_SIGNED_ SECRET	**CSSM_LIST Contents**
Challenge form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_SECRET
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_SIGNED_SECRET Second element: Word is a CSSM_DATA structure referencing a < signed secret > value of specified length

CSSM_SAMPLE_ TYPE_BIOMETRIC	**CSSM_LIST Contents**
Challenge form	A one-element list First element: WordID = CSSM_SAMPLE_TYPE_BIOMETRIC
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a < biometric template > value of specified length
CSSM_SAMPLE_ TYPE_PROMPTED_ BIOMETRIC	**CSSM_LIST Contents**
Challenge form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a non-terminated <prompt string> value of specified length
Response form	A two-element list First element: WordID = CSSM_SAMPLE_TYPE_PROMPTED_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a < biometric template > value of specified length
CSSM_SAMPLE_ TYPE_THRESHOLD	**CSSM_LIST Contents**
Challenge form	An (n+3)-element list First element: WordID = CSSM_SAMPLE_TYPE_THRESHOLD Second element: WordID = <k value> Third element: WordID = <n value> Fourth element: Sublist = (typed_challenge_list-1) Fifth element: Sublist = (typed_challenge_list-2) ... n+3rd element: Sublist = (typed_challenge_list-n)
Response form	A k-entry array of sample responses, each responding to one of the typed challenges in the threshold list

6.4.36 CSSM_CERT_TYPE

This variable specifies the type of certificate format supported by a certificate library. 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_TUPLE   = 0x0B,
    CSSM_CERT_ACL_ENTRY = 0x0C,
    CSSM_CERT_MULTIPLE = 0x7FFE,
    CSSM_CERT_LAST = 0x7FFF
} CSSM_CERT_TYPE, *CSSM_CERT_TYPE_PTR;
```

```
/* Applications wishing to define their own custom certificate
 * type should define and publicly document a uint32 value greater
 * than the CSSM_CL_CUSTOM_CERT_TYPE */
#define CSSM_CL_CUSTOM_CERT_TYPE 0x08000
```

6.4.37 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_SEXPR    = 0x05,
    CSSM_CERT_ENCODING_PGP      = 0x06,
    CSSM_CERT_ENCODING_MULTIPLE = 0x7FFE,
    CSSM_CERT_ENCODING_LAST     = 0x7FFF
} CSSM_CERT_ENCODING, *CSSM_CERT_ENCODING_PTR;
```

```
/* Applications wishing to define their own custom certificate
 * encoding should create a uint32 value greater than the
 * CSSM_CL_CUSTOM_CERT_ENCODING */
#define CSSM_CL_CUSTOM_CERT_ENCODING 0x8000
```

6.4.38 CSSM_ENCODED_CERT

This structure contains a pointer to a certificate in its encoded representation. The certificate is stored as a single contiguous byte array referenced by *CertBlob*. The length of the byte array is contained in the *Length* subfield of the *CertBlob*. The type and encoding of the certificate format are also contained in the structure.

```
typedef struct cssm_encoded_cert {
    CSSM_CERT_TYPE CertType ;          /* type of certificate */
    CSSM_CERT_ENCODING CertEncoding ; /* encoding for this packed cert */
    CSSM_DATA CertBlob ;              /* packed cert */
} CSSM_ENCODED_CERT, *CSSM_ENCODED_CERT_PTR ;
```

Definition

CertType
Indicates the type of the certificate referenced by CertBlob.

CertEncoding
Indicates the encoding of the certificate referenced by CertBlob.

CertBlob
A two field structure containing a reference to a certificate in its opaque data blob format and the length of the byte array that contains the certificate blob.

6.4.39 CSSM_CERT_PARSE_FORMAT

This type defines an extensible list of formats for parsed certificates.

```
typedef uint32 CSSM_CERT_PARSE_FORMAT, *CSSM_CERT_PARSE_FORMAT_PTR;

#define CSSM_CERT_PARSE_FORMAT_NONE      (0x00)
#define CSSM_CERT_PARSE_FORMAT_CUSTOM    (0x01) /* void* */
#define CSSM_CERT_PARSE_FORMAT_SEXPR     (0x02) /* CSSM_LIST */
#define CSSM_CERT_PARSE_FORMAT_COMPLEX   (0x03) /* void* */
#define CSSM_CERT_PARSE_FORMAT_OID_NAMED (0x04) /* CSSM_FIELDGROUP */
#define CSSM_CERT_PARSE_FORMAT_TUPLE     (0x05) /* CSSM_TUPLE */
#define CSSM_CERT_PARSE_FORMAT_MULTIPLE  (0x7FFE)
        /* multiple forms, each cert carries a parse format indicator */
#define CSSM_CERT_PARSE_FORMAT_LAST      (0x7FFF)

 /* Applications wishing to define their own custom parse
    format should create a * uint32 value greater than the
    CSSM_CL_CUSTOM_CERT_PARSE_FORMAT */

#define CSSM_CL_CUSTOM_CERT_PARSE_FORMAT (0x8000)
```

6.4.40 CSSM_PARSED_CERT

This structure contains a parsed representation of a certificate. It will likely have many parts, accessed by pointer. The developer must use a type cast to convert *ParsedCert* to the appropriate type, corresponding to the type and encoding of the certificate format as indicated in the structure.

```
typedef struct cssm_parsed_cert {
    CSSM_CERT_TYPE CertType ;  /* certificate type */
    CSSM_CERT_PARSE_FORMAT ParsedCertFormat ;
                               /* struct of ParsedCert */
    void *ParsedCert ;         /* parsed cert (to be typecast) */
} CSSM_PARSED_CERT, *CSSM_PARSED_CERT_PTR ;
```

Definition

CertType
Indicates the type of certificate that had been parsed to yield *ParsedCert*.

ParsedCertFormat
Indicates the structure and format representation of the parsed certificate. If the parsed representation is not available, then this value is CSSM_CERT_PARSE_FORMAT_NONE.

ParsedCert
A pointer to a parsed certificate represented in the structure and format indicated by *ParsedCertFormat*.

6.4.41 CSSM_CERTPAIR

This structure contains a certificate in two representations:

- A CSSM_ENCODED_CERT structure containing a single, opaque blob of certificate data
- A CSSM_PARSED_CERT structure containing a parsed, structured set of certificate data components.

The representation used for each form (such as type, encoding, and format) is included in the applicable substructure. At least one of the two representations must be present at any given time. The omitted representation is indicated by a NULL pointer value. When the omitted alternate representation is generated, it can be added to this structure at any time. When both representations are included, those representations must be equivalent forms of a single certificate. This structure is provided for performance and convenience reasons. For security purposes, however, only a signed certificate (necessarily encoded) can have its signature checked.

```
typedef struct cssm_cert_pair {
    CSSM_ENCODED_CERT EncodedCert; /* an encoded certificate blob */
    CSSM_PARSED_CERT ParsedCert;   /* equivalent parsed certificate */
} CSSM_CERT_PAIR, *CSSM_CERT_PAIR_PTR;
```

Definition

EncodedCert
A CSSM_ENCODED_CERT structure containing:

- A reference to an opaque, single byte-array representation of the certificate
- A certificate type descriptor
- A certificate encoding descriptor.

The certificate can have an equivalent parsed representation. If the parsed representation is provided it is contained in ParsedCert.

ParsedCert
A CSSM_PARSED_CERT structure containing:

- A certificate type
- A reference to a parsed representation of the certificate
- A parse format descriptor

The certificate can have an equivalent packed representation. If the packed representation is provided it is contained in EncodedCert.

6.4.42 CSSM_CERTGROUP_TYPE

This extensible list defines the type of a certificate group. A group can contain a single type of certificate or multiple types of certificates. Each certificate in the group can be represented in an encoded representation, a parsed representation, or both.

```
typedef uint32 CSSM_CERTGROUP_TYPE, *CSSM_CERTGROUP_TYPE_PTR;

#define CSSM_CERTGROUP_DATA          (0x00)
#define CSSM_CERTGROUP_ENCODED_CERT  (0x01)
#define CSSM_CERTGROUP_PARSED_CERT   (0x02)
```

```
#define CSSM_CERTGROUP_CERT_PAIR      (0x03)
```

6.4.43 CSSM_CERTGROUP

This structure contains an arbitrary number of certificates. The group can be restricted to a single type of certificate or can contain multiple types of certificates as indicated in the *CertGroupType* field. For legacy compatibility, this structure includes the previous definition. In that form, this is a list of encoded, signed certificates all of the same type and all in the same encoding. If one has a group of certificates of mixed types, they can be either encoded, parsed or both.

Each certificate in the mixed group can be represented in one or two forms:

- A CSSM_ENCODED_CERT structure containing a single, opaque blob of certificate data
- A CSSM_PARSED_CERT structure containing a parsed, structured set of certificate data components.

When using encoded and parsed representations, the specific format used for each certificate and each representation must be included in the applicable substructure. At least one of the two representations must be present in each list entry. When a single entry includes both representations, those representations are assumed but not guaranteed to be equivalent representations of a single certificate.

The number of certificates in the group is contained in the structure.

```
typedef struct cssm_certgroup {
    CSSM_CERT_TYPE CertType;
    CSSM_CERT_ENCODING CertEncoding;
        uint32 NumCerts ;        /* # of certificates in this list */
    union {
        CSSM_DATA_PTR CertList; /* legacy list of single type
                                   certificate blobs */
        CSSM_ENCODED_CERT_PTR EncodedCertList ;
                                /* list of multi-type
                                   certificate blobs */
        CSSM_PARSED_CERT_PTR ParsedCertList;
                                /* list of multi-type parsed certs */
        CSSM_CERT_PAIR_PTR PairCertList;
                                /* list of single or multi-type certs
                                   with two representations:
                                   blob and parsed */
    } GroupList;
    CSSM_CERTGROUP_TYPE CertGroupType;
                                /* type of structure in the GroupList */
    void *Reserved ;            /* reserved for implementation
                                   dependent use */
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

Definition

CertType
If all certificates in the *GroupList* are of the same type, this variable lists that type. Otherwise, the type should be CSSM_CERT_MULTIPLE.

CertEncoding
If all certificates in the *GroupList* are of the same encoding, this variable gives that encoding. Otherwise, the type should be CSSM_CERT_ENCODING_MULTIPLE.

NumCerts
The number of entries in the *GroupList* array.

GroupList
An array of certificates. The array contains exactly *NumCerts* entries. *CertGroupType* defines the type of structure contained in the array. The group types are described as follows:

CertGroupType Value	Field Name	Description
CSSM_CERTGROUP_DATA	*CertList*	(Legacy) A pointer to an array of CSSM_DATA structures. Each CertList array entry references a single certificate structure and indicates the length of the structure. A single type and encoding apply to all certificates in this group. The type and encoding are indicated in *CertType* and *CertEncoding* respectively.
CSSM_CERTGROUP_ ENCODED_CERT	*EncodedCertList*	A pointer to an array of CSSM_ENCODED_CERT structures. Each *EncodedCertList* array entry references a certificate in an opaque, single byte-array representation, and describes the format of the certificate data contained in the byte-array. Each certificate encoding and type can be distinct, as indicated in each array element.
CSSM_CERTGROUP_ PARSED_CERT	*ParsedCertList*	A pointer to an array of CSSM_PARSED_CERT structures. Each *ParsedCertList* array entry references a certificate in a parsed representation, and indicates the certificate type and parse format of that certificate.
CSSM_CERTGROUP_ CERT_PAIR	*PairCertList*	A pointer to an array of CSSM_CERT_PAIR structures. Each *PairCertList* array entry aggregates two certificate representations: an opaque encoded certificate blob, and a parsed certificate representation. At least one of the two representations must be present in each array entry. If both are present, they are assumed but not guaranteed to correspond to one another. If the parsed form is being used in a security sensitive operation, then it must have been verified against the packed, encoded form, whose signature must have been verified

Reserved
This field is reserved for future use.

6.4.44 CSSM_BASE_CERTS

This structure contains a group of zero or more certificates and optional handles identifying a Trust Policy Service Provider and a Certificate Library Service Provider that could be used to verify these certificates.

```
typedef struct cssm_base_certs {
    CSSM_TP_HANDLE TPHandle;
    CSSM_CL_HANDLE CLHandle;
    CSSM_CERTGROUP Certs;
} CSSM_BASE_CERTS, *CSSM_BASE_CERTS_PTR;
```

Definitions

TPHandle
The handle of a Trust Policy Service Provider that could be used to verify the certificates contained in the certificate group *Certs*. This handle is optional.

CLHandle
The handle of a Certificate Library Service Provider that could be used to verify the certificates contained in the certificate group *Certs*. This handle is optional.

Certs
A CSSM_CERTGROUP structure containing zero or more typed certificates.

6.4.45 CSSM_ACCESS_CREDENTIALS

This data structure contains the set of credentials a caller must provide when initiating a request for authorized access to a resource managed by a service provider module. The caller can present a set of certificates, and a set of samples. The certificates are optional, but if provided they must be immediate values included with the structure. Typically at least one sample is required, but if access is already authorized, then samples are not required. When samples are required, they can be:

- Provided as immediate values in the CSSM_ACCESS_CREDENTIALS structure
- Acquired from the original requester by the service provider invoking the callback function
- Using a protected path input mechanism, which the service provider uses directly.

```
typedef struct cssm_access_credentials {
    CSSM_STRING EntryTag;
    CSSM_BASE_CERTS BaseCerts;
    CSSM_SAMPLEGROUP Samples;
    CSSM_CHALLENGE_CALLBACK Callback;
    void *CallerCtx;
} CSSM_ACCESS_CREDENTIALS, *CSSM_ACCESS_CREDENTIALS_PTR;
```

Definitions

EntryTag

An optional, user-defined tag value identifying the target ACL entry the caller is attempting to satisfy. If the caller does not know the tag value associated with a particular ACL entry, then this value should be NULL. The tag value is a human-readable value specified when an ACL entry is created. A caller can use this value to selectively identify one or more ACL entries. The value may not be unique among entries in a single ACL.

BaseCerts

A CSSM_BASE_CERT structure containing zero or more typed certificates. The structure also contains optional handles identifying a CL service module and a TP service module that can be used to verify the group of certificates. All certificates required by a service provider must be provided as immediate values in this structure. If the service provider does not require any certificates, this group should contain zero elements.

Samples

An array of CSSM_SAMPLE structures. The array contains zero or more samples. Each sample contains a CSSM_LIST structure and an optional SUBSERVICE_UID. Each CSSM_LIST structure contains a sample type and an optional sample value. If the sample value is not contained in the list, or the sample is not acquirable through a protected mechanism, the *Callback* function must be provided. A service provider uses the callback to obtain a sample of the type specified in the list.

Callback

A CSSM_CHALLENGE_CALLBACK function pointer. The service provider module can use this *callback* function to obtain a sample value of the type specified in a SAMPLE list. If the *callback* function is provided and *Samples* is NULL, then the service provider must invoke the callback providing a complete list of applicable *Response* options in the *Challenge* parameter of the *callback* function.

CallerCtx

A requester-defined structure that a service provider module must pass to the caller as an input parameter when invoking the *Callback* function.

6.4.46 CSSM_ACL_SUBJECT_TYPE

This type defines symbol names for the valid subject types contained in an ACL entry. ACL entries containing these subject types can be:

- Stored by the service provider and used to support the authorization decisions of the service provider
- Passed as input by a caller when initializing or updating an ACL entry
- Returned as output by a service provider in response to a caller query for ACL entries.

```
typedef sint32 CSSM_ACL_SUBJECT_TYPE

#define CSSM_ACL_SUBJECT_TYPE_ANY CSSM_WORDID__STAR_
#define CSSM_ACL_SUBJECT_TYPE_THRESHOLD  CSSM_WORDID_THRESHOLD
#define CSSM_ACL_SUBJECT_TYPE_PASSWORD CSSM_WORDID_PASSWORD
#define CSSM_ACL_SUBJECT_TYPE_PROTECTED_PASSWORD CSSM_WORDID_PROTECTED_PASSWORD
#define CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD CSSM_WORDID_PROMPTED_PASSWORD
#define CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY CSSM_WORDID_PUBLIC_KEY
#define CSSM_ACL_SUBJECT_TYPE_HASHED_SUBJECT CSSM_WORDID_HASHED_SUBJECT
#define CSSM_ACL_SUBJECT_TYPE_BIOMETRIC CSSM_WORDID_BIOMETRIC
#define CSSM_ACL_SUBJECT_TYPE_PROTECTED_BIOMETRIC CSSM_WORDID_PROTECTED_BIOMETRIC
#define CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME CSSM_WORDID_LOGIN_NAME
#define CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME CSSM_WORDID_PAM_NAME
```

CSSM_ACL_SUBJECT_TYPE_*	General Description
ANY	The subject matches everyone. The *AuthorizationTag* associated with this subject represents operations that can be performed by anyone.
THRESHOLD	The subject specifies a nested "k-of-n" subject. Each of the options is another defined ACL subject type.
PASSWORD	The subject value is a password or passphrase value.
PROTECTED_PASSWORD	The subject value is dynamically acquired and matched by a service that manages a protected data acquisition path to gather and verify a password or passphrase value.
PROMPTED_PASSWORD	The subject value is a password or passphrase value with an associated prompt value. The prompt value can be presented to an application.
PUBLIC_KEY	The subject value is a public key of some specified format.
HASHED_SUBJECT	The subject value is the hash value of any ACL Subject of valid type. For example, the hash of a password.
BIOMETRIC	The subject value is an enrolled biometric template value.
PROTECTED_BIOMETRIC	The subject value is acquired and matched by a service that manages a protected data acquisition path to gather and verify a biometric sample from a Biometric Sensor
LOGIN_NAME	The subject value is a login name. The name space of login names is defined and managed by the service provider.
EXT_PAM_NAME	The subject value is a user name. The name space for user names is defined and managed by the Pluggable Authentication Modules (PAM), an external service. PAM is a specification standard of The Open Group.

6.4.47 CSSM_ACL_AUTHORIZATION_TAG

This type defines a set of names for operations that can be performed on ACL entries and on the resource objects that are protected by ACL entries. These names are represented as integer values. When creating an ACL entry, a set of these values should be aggregated into a CSSM_LIST structure that is used to initialize the *AuthorizationTag* item within the ACL entry.

The operations named by these constants correspond to functions defined in the CDSA and CSSM APIs. Service providers can define names for additional controlled operations. Service providers are encouraged to identify two types of operations:

1. Operations that use the resource object
2. Operations that modify the ACLs that protect the resource object

Vendors defining new constants must manage the integer name space to avoid definitions that conflict with other vendors.

```
/* Authorization tag type */
typedef sint32 CSSM_ACL_AUTHORIZATION_TAG;

/* All vendor specific constants must be in the number range
   starting at CSSM_ACL_AUTHORIZATION_TAG_VENDOR_DEFINED_START
 */
#define CSSM_ACL_AUTHORIZATION_TAG_VENDOR_DEFINED_START (0x00010000)

/* No restrictions. Permission to perform all operations on
   the resource or available to an ACL owner.
 */
#define CSSM_ACL_AUTHORIZATION_ANY CSSM_WORDID__STAR_

/* Defined authorization tag values for CSPs */
#define CSSM_ACL_AUTHORIZATION_LOGIN CSSM_WORDID_LOGIN
#define CSSM_ACL_AUTHORIZATION_GENKEY CSSM_WORDID_GENKEY
#define CSSM_ACL_AUTHORIZATION_DELETE CSSM_WORDID_DELETE
#define CSSM_ACL_AUTHORIZATION_EXPORT_WRAPPED CSSM_WORDID_EXPORT_WRAPPED
#define CSSM_ACL_AUTHORIZATION_EXPORT_CLEAR CSSM_WORDID_EXPORT_CLEAR
#define CSSM_ACL_AUTHORIZATION_IMPORT_WRAPPED CSSM_WORDID_IMPORT_WRAPPED
#define CSSM_ACL_AUTHORIZATION_IMPORT_CLEAR CSSM_WORDID_IMPORT_CLEAR
#define CSSM_ACL_AUTHORIZATION_SIGN CSSM_WORDID_SIGN
#define CSSM_ACL_AUTHORIZATION_ENCRYPT CSSM_WORDID_ENCRYPT
#define CSSM_ACL_AUTHORIZATION_DECRYPT CSSM_WORDID_DECRYPT
#define CSSM_ACL_AUTHORIZATION_MAC CSSM_WORDID_MAC
#define CSSM_ACL_AUTHORIZATION_DERIVE CSSM_WORDID_DERIVE

/* Defined authorization tag values for DLs */
#define CSSM_ACL_AUTHORIZATION_DBS_CREATE CSSM_WORDID_DBS_CREATE
#define CSSM_ACL_AUTHORIZATION_DBS_DELETE CSSM_WORDID_DBS_DELETE
#define CSSM_ACL_AUTHORIZATION_DB_READ CSSM_WORDID_DB_READ
#define CSSM_ACL_AUTHORIZATION_DB_INSERT CSSM_WORDID_DB_INSERT
#define CSSM_ACL_AUTHORIZATION_DB_MODIFY CSSM_WORDID_DB_MODIFY
#define CSSM_ACL_AUTHORIZATION_DB_DELETE CSSM_WORDID_DB_DELETE
```

The meaning of each authorization tag is listed in the following table. The tag may have slightly different meanings depending on the ACL type.

Authorization Tag **CSSM_ACL_AUTHORIZATION_***	**Meaning** **The subject of the ACL is authorized to**
ANY	No restrictions. Permission to perform all operations on the resource.
LOGIN	login to the subservice.
GENKEY	generate a new key resource managed by the subservice.
DELETE	delete resources managed by the subservice.
EXPORT_WRAPPED	export the key in a wrapped form.
EXPORT_CLEAR	export the key as cleartext.
IMPORT_WRAPPED	import a new key resource to be managed by the subservice. The new resource is wrapped.
IMPORT_CLEAR	import a new key resource to be managed by the subservice. The new resource is cleartext.
SIGN	perform signature operations using the key.
ENCRYPT	perform encryption operations using the key.
DECRYPT	perform decryption operations using the key.
MAC	perform Message Authentication Code (MAC) operation using the key.
DERIVE	perform key derivation operations using the key as the basis.
DBS_CREATE	create new databases managed by the subservice.
DBS_DELETE	delete existing databases managed by the subservice.
DB_READ	read the contents of record in a database.
DB_INSERT	insert new records into a database.
DB_MODIFY	change the value of meta-data or the data value of records in a database.
DB_DELETE	delete existing records in a database.

Each type of ACL has a set of authorization tags that are valid. The following table lists each defined ACL type and the valid attribute values for those ACLs.

ACL Type	**Valid Authorization Tags**
CSP Login	CSSM_ACL_AUTHORIZATION_ANY CSSM_ACL_AUTHORIZATION_LOGIN CSSM_ACL_AUTHORIZATION_GENKEY CSSM_ACL_AUTHORIZATION_DELETE CSSM_ACL_AUTHORIZATION_IMPORT_WRAPPED CSSM_ACL_AUTHORIZATION_IMPORT_CLEAR
CSP Secret Key	CSSM_ACL_AUTHORIZATION_ANY CSSM_ACL_AUTHORIZATION_DELETE CSSM_ACL_AUTHORIZATION_EXPORT_WRAPPED CSSM_ACL_AUTHORIZATION_EXPORT_CLEAR CSSM_ACL_AUTHORIZATION_ENCRYPT CSSM_ACL_AUTHORIZATION_DECRYPT CSSM_ACL_AUTHORIZATION_MAC CSSM_ACL_AUTHORIZATION_DERIVE
CSP Private Key	CSSM_ACL_AUTHORIZATION_ANY CSSM_ACL_AUTHORIZATION_DELETE CSSM_ACL_AUTHORIZATION_EXPORT_WRAPPED CSSM_ACL_AUTHORIZATION_EXPORT_CLEAR CSSM_ACL_AUTHORIZATION_SIGN CSSM_ACL_AUTHORIZATION_DECRYPT

	CSSM_ACL_AUTHORIZATION_DERIVE
DL Database	CSSM_ACL_AUTHORIZATION_ANY CSSM_ACL_AUTHORIZATION_DBS_CREATE CSSM_ACL_AUTHORIZATION_DBS_DELETE CSSM_ACL_AUTHORIZATION_DB_READ CSSM_ACL_AUTHORIZATION_DB_INSERT CSSM_ACL_AUTHORIZATION_DB_MODIFY

6.4.48 CSSM_AUTHORIZATIONGROUP

This structure contains a group of authorization tags.

```
typedef struct cssm_authorizationgroup {
    uint32 NumberOfAuthTags;
    CSSM_ACL_AUTHORIZATION_TAG *AuthTags;
} CSSM_AUTHORIZATIONGROUP, *CSSM_AUTHORIZATIONGROUP_PTR;
```

Definitions

NumberOfAuthTags
The number of authorization tags in the array *AuthTags*.

AuthTags
A pointer to an array of integers representing authorization tag values.

6.4.49 CSSM_ACL_VALIDITY_PERIOD

This data type defines a structure containing a start date and end date for the validity of an ACL entry. The date and time for an ACL entry, a CSSM_TUPLE or an *AuthCompute* call is an ASCII string of the form, for example, "1999-06-30_15:05:39". Dates are compared by normal string comparison.

```
typedef struct cssm_acl_validity_period {
    CSSM_DATA StartDate;
    CSSM_DATA EndDate;
} CSSM_ACL_VALIDITY_PERIOD, *CSSM_ACL_VALIDITY_PERIOD_PTR;
```

6.4.50 CSSM_ACL_ENTRY_PROTOTYPE

This data type defines the abstract structure of an Access Control List (ACL) entry. The structure is used:

- By an application to provide initial values for an ACL entry
- By a service provider returning ACL entry information in response to an application query.

When an application uses this structure to provide initial values for an ACL entry, the structure is passed as an input parameter to a CDSA security service function. The structure must contain all fields as immediate data values, except the *TypedSubject* field. An application can provide the value of the *TypedSubject* field by:

- Including the subject type and subject value as immediate data in the prototype structure
- Indicating the type of subject value the application would like to present to the service provider, and requesting that the service provider invoke an application-implemented callback function to gather the subject value

- Indicating the type and third-party source of the subject value, and requesting that the service provider acquire the sample through a protected path with the third party.

An ACL entry prototype is considered to be a duplicate of an existing ACL entry if the entries have equal values for all fields except *EntryTag*.

```
typedef struct cssm_acl_entry_prototype {
    CSSM_LIST TypedSubject;
    CSSM_BOOL Delegate;
    CSSM_AUTHORIZATIONGROUP Authorization;
    CSSM_ACL_VALIDITY_PERIOD TimeRange;
    CSSM_STRING EntryTag;
} CSSM_ACL_ENTRY_PROTOTYPE, *CSSM_ACL_ENTRY_PROTOTYPE_PTR;
```

Definitions

TypedSubject
A CSSM_LIST structure containing one or two elements for the subject of an ACL entry. The first list element is the type of the subject. The second element is the subject value. The subject value can be empty if a callback function is provided to acquire the subject value interactively. In either case, the subject type indicates the type of the immediate value or the type of the value to be supplied on-demand. The following table specifies the CSSM_LIST elements for each ACL subject type under each valid use scenario.

CSSM_ACL_SUBJECT_TYPE_ ANY	**CSSM_LIST Contents**
Immediate Type and Value Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_ANY
Query-response Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_ANY
CSSM_ACL_SUBJECT_TYPE_ THRESHOLD	**CSSM_LIST Contents**
Immediate Type and Value Form	An (n+3)-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_THRESHOLD Second element: WordID = <k value> Third element: WordID = <n value> Fourth element: Sublist = (typed_subject_list-1) Fifth element: Sublist = (typed_ subject _list-2) . . . n+3rd element: Sublist = (typed_ subject _list-n)
Immediate Type and Request for Callback Form	A 3-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_THRESHOLD Second element: WordID = <k value> Third element: WordID = <n value>
Query-response Form	An (n+3)-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_THRESHOLD Second element: WordID = <k value> Third element: WordID = <n value> Fourth element: Sublist = (typed_ subject _list-1) Fifth element: Sublist = (typed_ subject _list-2) . . . n+3rd element: Sublist = (typed_ subject _list-n)
CSSM_ACL_SUBJECT_TYPE_ PASSWORD	**CSSM_LIST Contents**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PASSWORD
Query-response Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PASSWORD
CSSM_ACL_SUBJECT_TYPE_ PROMPTED_PASSWORD	**CSSM_LIST Contents**
Immediate Type and Value Form	A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <prompt string> value of specified length Third element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD
Query-response Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <prompt string> value of specified length

CSSM_ACL_SUBJECT_TYPE_ PROTECTED_PASSWORD	**CSSM_LIST Contents**
Immediate Type and Protected Path Acquisition Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_PASSWORD Second element: Word is a CSSM_DATA structure identifying a <service provider subservice uid> in memory (by starting address and length).
Query-response Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_PASSWORD
CSSM_ACL_SUBJECT_TYPE_ PUBLIC_KEY	**CSSM_LIST Contents**
Immediate Type and Value Form	A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY Second element: Word is a CSSM_KEYBLOB_RAW_FORMAT value Third element: Word is the keyblob
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY
Query-response Form	A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY Second element: Word is a CSSM_KEYBLOB_RAW_FORMAT value Third element: Word is the keyblob
CSSM_ACL_SUBJECT_TYPE_ BIOMETRIC	**CSSM_LIST Contents**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_BIOMETRIC Second element: Word is a CSSM_DATA structure referencing a < biometric template> value of specified length
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_BIOMETRIC
Query-response Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_BIOMETRIC
CSSM_ACL_SUBJECT_TYPE_ PROTECTED_BIOMETRIC	**CSSM_LIST Contents**
Immediate Type and Protected Path Acquisition Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_BIOMETRIC Second element: Word is a CSSM_DATA structure identifying a <service provider subservice uid> in memory (by starting address and length).
Query-response Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_BIOMETRIC
CSSM_ACL_SUBJECT_TYPE_ LOGIN_NAME	**CSSM_LIST Contents**
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < login name> value of specified length
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME
Query-response Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < login name> value of specified length

CSSM_ACL_SUBJECT_TYPE_ EXT_PAM_NAME	CSSM_LIST Contents
Immediate Type and Value Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < PAM-user name> value of specified length
Immediate Type and Request for Callback Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME
Query-response Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < PAM_user name> value of specified length

Delegate
A CSSM_BOOL value indicating whether the *TypedSubject* can delegate the access rights defined by the Authorization. Delegation is based on the use of the public key infrastructure (PKI). Therefore, if *Delegate* is true (CSSM_TRUE), then the subject of the ACL entry must be of type CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY.

Authorization
A CSSM_AUTHORIZATIONGROUP structure enumerating the operations for which permission is granted to the *TypedSubject*.

TimeRange
A CSSM_ACL_VALIDITY_PERIOD structure containing the time period for which the ACL entry is valid. The first list element is the start time. The second list element is the stop time.

EntryTag
A user-defined tag value associated with the ACL entry. The tag value is a human-readable value specified at ACL creation time. A caller can use this value to selectively identify one or more ACL entries. The value may not be unique.

6.4.51 CSSM_ACL_OWNER_PROTOTYPE

This data type is the abstract definition of an Access Control List (ACL) Owner (that is, the owner of the resource protected by the ACL). The ACL_OWNER acts as an ACL entry protecting the resource ACL. It is subset of an ACL_ENTRY_PROTOTYPE; consisting only of the *TypedSubject*(who*is*the*Owner*), *Delegate* flag to indicate whether the "Owner" can delegate his rights over the ACL. The value of *TypedSubject* is taken initially from the ACL_ENTRY_PROTOTYPE that forms the initial ACL on the resource (the creator of the resource is the initial owner), and the *Delegate* flag is initially set to CSSM_FALSE.

After creation, the Owner may retrieve the Owner definition in the form of a CSSM_ACL_OWNER_PROTOTYPE, and replace it with a new Owner definition.

```
typedef struct cssm_acl_owner_prototype {
    CSSM_LIST TypedSubject;
    CSSM_BOOL Delegate;
} CSSM_ACL_OWNER_PROTOTYPE, *CSSM_ACL_OWNER_PROTOTYPE_PTR;
```

See CSSM_ACL_ENTRY_PROTOTYPE for a definition of *TypedSubject* and *Delegate*.

6.4.52 CSSM_ACL_SUBJECT_CALLBACK

This type defines the form of the *callback* function a service provider module can use to acquire a value for the subject of a prototype ACL entry. The service provider initializes the *SubjectRequest* and the Application initializes the *SubjectResponse* based on the type of the request.

```
typedef CSSM_RETURN (CSSMAPI * CSSM_ACL_SUBJECT_CALLBACK)
    (const CSSM_LIST *SubjectRequest,
    CSSM_LIST_PTR SubjectResponse,
    void *CallerContext,
    const CSSM_MEMORY_FUNCS *MemFuncs);
```

Definitions

SubjectRequest (input)

A pointer to a CSSM_LIST structure containing a typed request for an ACL subject list. The type is specified as the first element in the list. The service provider selects the subject type from one of two sources:

- The original requester can state a preferred subject type in the original call. If the stated subject type is not supported by the service provider the error condition CSSM_ACL_INVALID_NEW_ACL_ENTRY is returned from the original call and this *callback* function is never invoked.
- The service provider can provide a list of all applicable subject types, in a THRESHOLD list requesting k-of-n subject types from the application.

If the service provider supports more than one subject type, then the first list element indicates the type CSSM_ACL_SUBJECT_TYPE_THRESHOLD with a k-value of one and an n-value equal to the number of supported subject types. The following n list elements describe each supported subject type. A table defining the CSSM_LIST contents for each possible *SubjectRequest* and *SubjectResponse* pair is defined in a table below.

SubjectResponse (output)

A pointer to a CSSM_LIST structure containing the typed response to the *SubjectRequest*. The first element of the list is the type of the response. The returned *Subject* list must be an appropriate response for the type options presented in *SubjectRequest*. The response is a single subject list that can contain a set of subject lists. If the response contains a set of subject lists then the first element must be of type CSSM_ACL_SUBJECT_TYPE_THRESHOLD followed by the number of *Subject* lists that follow. A table defining the CSSM_LIST contents for each possible *SubjectRequest* and *SubjectResponse* pair is defined in a table below.

CallerContext (input)

A generic pointer to context information that was provided by the original requester and is being returned to its originator.

MemFuncs (input)

A pointer to a CSSM_MEMORY_FUNCS structure. The subject callback must use the functions in this structure to allocate memory returned to the service provider in the *Response* structure.

CSSM_ACL_SUBJECT_TYPE_ ANY	**CSSM_LIST Contents**
SubjectRequest Form	Not used.
SubjectResponse Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_ANY
CSSM_ACL_SUBJECT_TYPE_ THRESHOLD	**CSSM_LIST Contents**
SubjectRequest Form	An (n+3)-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_THRESHOLD Second element: WordID = 1 Third element: WordID = <n value> Fourth element: Sublist = (typed_ subject _list-1) Fifth element: Sublist = (typed_ subject _list-2) . . . n+3rd element: Sublist = (typed_ subject _list-n)
SubjectResponse Form	An (n+3)-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_THRESHOLD Second element: WordID = <k value> Third element: WordID = <n value> Fourth element: Sublist = (typed_ subject _list-1) Fifth element: Sublist = (typed_ subject _list-2) . . . n+3rd element: Sublist = (typed_ subject _list-n)
CSSM_ACL_SUBJECT_TYPE_ PASSWORD	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PASSWORD SubjectResponse Form!A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
CSSM_ACL_SUBJECT_TYPE_ PROMPTED_PASSWORD	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD SubjectResponse Form!A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROMPTED_PASSWORD Second element: Word is a CSSM_DATA structure referencing a non-terminated <prompt string> value of specified length Third element: Word is a CSSM_DATA structure referencing a non-terminated <password string> value of specified length
CSSM_ACL_SUBJECT_TYPE_ PROTECTED_PASSWORD	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_PASSWORD SubjectResponse Form!A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_PASSWORD Second element: Word is a CSSM_DATA structure identifying a <service provider subservice uid> in memory (by starting address and length).
CSSM_ACL_SUBJECT_TYPE_ PUBLIC_KEY	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY

SubjectResponse Form	A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PUBLIC_KEY Second element: Word is a CSSM_KEYBLOB_RAW_FORMAT value Third element: Word is the keyblob
CSSM_ACL_SUBJECT_TYPE_ BIOMETRIC	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_BIOMETRIC
SubjectResponse Form	A three-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_BIOMETRIC Second element: Word is a CSSM_DATA structure identifying a <service provider subservice uid> in memory (by starting address and length). Third element: Word is a CSSM_DATA structure referencing a <biometric template> value of specified length
CSSM_ACL_SUBJECT_TYPE_ PROTECTED_BIOMETRIC	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_BIOMETRIC
SubjectResponse Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_PROTECTED_BIOMETRIC Second element: Word is a CSSM_DATA structure identifying a <service provider subservice uid> in memory (by starting address and length).
CSSM_ACL_SUBJECT_TYPE_ LOGIN_NAME	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME
SubjectResponse Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_LOGIN_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < login name> value of specified length
CSSM_ACL_SUBJECT_TYPE_ EXT_PAM_NAME	**CSSM_LIST Contents**
SubjectRequest Form	A one-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME
SubjectResponse Form	A two-element list First element: WordID = CSSM_ACL_SUBJECT_TYPE_EXT_PAM_NAME Second element: Word is a CSSM_DATA structure referencing a non-terminated < PAM_user name> value of specified length

6.4.53 CSSM_ACL_ENTRY_INPUT

This data structure is used by a caller to provide an ACL entry prototype to a service provider. An ACL entry prototype must be provided as input to a service provider when a new controlled resource is being created and when a new ACL entry is being added to an existing access-controlled resource.

```
typedef struct cssm_acl_entry_input {
    CSSM_ACL_ENTRY_PROTOTYPE Prototype;
    CSSM_ACL_SUBJECT_CALLBACK Callback;
    void *CallerContext;
}  CSSM_ACL_ENTRY_INPUT, *CSSM_ACL_ENTRY_INPUT_PTR;
```

Definitions

Prototype
A CSSM_ACL_ENTRY_PROTOTYPE structure containing the initial values for an ACL entry. All sub-components of the ACL entry, except the ACL entry *TypedSubject*, must be provided as immediate data in this structure. The *Prototype.TypedSubject* field is of type CSSM_LIST. The ACL entry *TypedSubject* value can be provided by:

- Including the subject type and subject value as immediate data in this structure.
- Including a preferred subject type with no value as immediate data, and referencing a protected data path and service for acquiring the value
- Including a preferred subject type with no value as immediate data, and referencing an external authentication or authorization service that can provide the value
- Including a preferred subject type with no value as immediate data, and requesting that the service provider invoke an application-implemented callback function to gather the subject value from the application

Callback
A function pointer the service provider can de-reference (if necessary) to acquire the Subject of an ACL entry prototype from the original requester. The original requester must implement this function if the Subject value is not provided through alternate means, such as immediate data, a protected data path and service, or an external authentication or authorization service. If the initial value of the ACL entry Subject is provided by one of these other means, then the specified *Callback* must be NULL.

CallerCtx
A requester-defined structure that a service provider module must pass as an input parameter to the original requester when invoking the *Callback* function.

6.4.54 CSSM_RESOURCE_CONTROL_CONTEXT

This data structure is used by a caller when creating a new resource. The caller can provide *AccessCredentials* as evidence that the caller is allowed to create a new resource. The caller can also provide an ACL entry prototype to be used by the service provider in constructing the initial ACL entry controlling access to the new resource.

```
typedef struct cssm_resource_control_context {
    CSSM_ACCESS_CREDENTIALS_PTR AccessCred;
    CSSM_ACL_ENTRY_INPUT InitialAclEntry;
} CSSM_RESOURCE_CONTROL_CONTEXT, *CSSM_RESOURCE_CONTROL_CONTEXT_PTR;
```

Definitions

AccessCred
A pointer to the set of credentials required to prove the caller's right to create a new resource. Required credentials can include zero or more certificates and zero or more samples. If certificates are provided, they must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure. If no additional credentials are required to demonstrate the caller's right to create a new resource, then *AccessCred* can be NULL.

InitialAclEntry
The prototype for the initial ACL entry that will control access to the newly created

resource. All sub-components of the ACL entry, except the ACL entry Subject, must be provided as immediate data within this structure. The ACL entry Subject can be provided as immediate data or be acquired through a callback function included in the *InitialAclEntry* structure.

6.4.55 CSSM_ACL_HANDLE

This data type defines an opaque handle that uniquely identifies a single ACL entry associated with a particular resource.

```
typedef  CSSM_HANDLE CSSM_ACL_HANDLE;
```

6.4.56 CSSM_ACL_ENTRY_INFO

This data structure is used to return ACL entry information from a service provider module to a caller. The structure includes:

- An ACL entry prototype that can be used by the caller to create an input prototype when updating an ACL entry.
- An ACL entry handle, which is a unique value, defined and managed by the service provider module.

The ACL information structure does not indicate the resource for which access control is being defined. A caller requests ACL information for a specific resource. The returned ACL entries are associated with the target resource.

```
typedef  struct  cssm_acl_entry_info  {
    CSSM_ACL_ENTRY_PROTOTYPE EntryPublicInfo;
    CSSM_ACL_HANDLE EntryHandle;
} CSSM_ACL_ENTRY_INFO, *CSSM_ACL_ENTRY_INFO_PTR;
```

Definitions

EntryPublicInfo
A CSSM_ACL_ENTRY_PROTOTYPE containing the public information from an ACL entry. The structure includes:

- The subject type - A CSSM_LIST structure containing one element identifying the type of subject stored in the ACL entry.
- Delegation flag - a boolean value indicating whether the subject can delegate these permissions.
- Authorization array - defines the set of operations for which permission is granted to the subject.
- Validity period - the time period for which the ACL entry is valid.
- ACL entry tag - A user-defined tag value associated with the ACL entry. The tag value is a human-readable value specified at ACL creation time. A caller can use this value to identify an ACL entry. The value may not be unique.

EntryHandle
A unique, opaque identifier for the ACL entry. The value is defined and managed by the service provider who manages the ACL entries. The handle is valid for the current service provider attach session.

6.4.57 CSSM_ACL_EDIT_MODE

This data type defines identifiers for operations that modify an existing access control list (ACL). The three operations are:

- Add a new entry to the ACL
- Remove an existing entry from the ACL
- Replace an existing entry in the ACL

```
typedef  uint32 CSSM_ACL_EDIT_MODE;

#define CSSM_ACL_EDIT_MODE_ADD (1)
#define CSSM_ACL_EDIT_MODE_DELETE (2)
#define CSSM_ACL_EDIT_MODE_REPLACE (3)
```

6.4.58 CSSM_ACL_EDIT

This data structure contains the description of an edit operation to be applied to an existing Access Control List (ACL). The editing instructions include the edit operation, a handle to an existing ACL managed by a service provider, and information a service provider could use to update the existing ACL entry or to create a new ACL entry managed by the service provider module.

```
typedef struct cssm_acl_edit {
    CSSM_ACL_EDIT_MODE EditMode;
    CSSM_ACL_HANDLE OldEntryHandle;
    const CSSM_ACL_ENTRY_INPUT *NewEntry;
} CSSM_ACL_EDIT, *CSSM_ACL_EDIT_PTR;
```

Definitions

EditMode
The type of edit operation to be performed on one of a set of ACL entries that control access to some resource.

OldEntryHandle
A unique, opaque value identifying an existing ACL entry. The name space for these identifiers is defined and managed by a service provider module. The identified ACL entry can be replaced or deleted.

NewEntry
A pointer to a prototype ACL entry. The prototype entry is used to update the ACL entry identified by *OldEntryHandle* according to the operation specified by *EditMode*. If *EditMode* is CSSM_ACL_EDIT_MODE_DELETE, this value must be NULL.

6.4.59 CSSM_PROC_ADDR

Generic pointer to a CSSM function.

```
#if defined(WIN32)
typedef FARPROC CSSM_PROC_ADDR;
#else
typedef void (CSSMAPI *CSSM_PROC_ADDR) ();
#endif
typedef CSSM_PROC_ADDR *CSSM_PROC_ADDR_PTR;
```

6.4.60 CSSM_KR_POLICY_TYPE

Support for the KRMM relation.

```
typedef uint32 CSSM_KR_POLICY_TYPE;
#define CSSM_KR_INDIV_POLICY (0x00000001)
#define CSSM_KR_ENT_POLICY (0x00000002)
#define CSSM_KR_LE_MAN_POLICY (0x00000003)
#define CSSM_KR_LE_USE_POLICY (0x00000004)
```

6.4.61 CSSM_FUNC_NAME_ADDR

This structure binds a function to the runtime address of the procedure that implements the named function. Function names are limited in length to the size of a CSSM_STRING.

```
typedef struct cssm_func_name_addr {
    CSSM_STRING Name;
    CSSM_PROC_ADDR Address;
}CSSM_FUNC_NAME_ADDR, *CSSM_FUNC_NAME_ADDR_PTR;
```

Definition

Name
The name of the function represented as a fixed-length string.

Address
The runtime address of the procedure implementing the named function.

6.4.62 CSSM_MEMORY_FUNCS and CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the CSSM and the security service modules. The functions are used when memory needs to be allocated by the CSSM or security services for returning data structures to the applications

```
typedef void * (CSSMAPI *CSSM_MALLOC)
    ( uint32 size,
    void * allocref)

typedef void (CSSMAPI *CSSM_FREE)
    (void * memblock,
    void * allocref)

typedef void * (CSSMAPI *CSSM_REALLOC)
    (void * memblock,
    uint32 size,
    void * allocref)

typedef void * (CSSMAPI *CSSM_CALLOC)
    (uint32 num,
    uint32 size,
    void * allocref)

typedef struct cssm_memory_funcs {
    CSSM_MALLOC malloc_func;
    CSSM_FREE free_func;
    CSSM_REALLOC realloc_func;
    CSSM_CALLOC calloc_func;
    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 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 (referenced by *memblock*) from heap *allocref*.

realloc_func
: Pointer to function that returns a void pointer to the reallocated memory block (referenced by *memblock*) 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 the default memory heap if *allocref* is NULL in specific calls to these functions.

See Appendix B on page 935 for details about the application memory functions.

6.5 Common Error Return Codes

This section defines all the Error Values that can be returned by CSSM operations.

The Error Values that can be returned by CSSM functions can be either derived from the Common Error Codes defined in Appendix A on page 925, or they are specific to the CSSM function.

6.5.1 Error Values Derived from Common Error Codes

```
#define CSSMERR_CSSM_INTERNAL_ERROR \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)

#define CSSMERR_CSSM_MEMORY_ERROR \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)

#define CSSMERR_CSSM_MDS_ERROR \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)

#define CSSMERR_CSSM_INVALID_POINTER \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)

#define CSSMERR_CSSM_INVALID_INPUT_POINTER \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)

#define CSSMERR_CSSM_INVALID_OUTPUT_POINTER \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)

#define CSSMERR_CSSM_FUNCTION_NOT_IMPLEMENTED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)

#define CSSMERR_CSSM_SELF_CHECK_FAILED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)

#define CSSMERR_CSSM_OS_ACCESS_DENIED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)

#define CSSMERR_CSSM_FUNCTION_FAILED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)

#define CSSMERR_CSSM_MODULE_MANIFEST_VERIFY_FAILED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_MODULE_MANIFEST_VERIFY_FAILED)

#define CSSMERR_CSSM_INVALID_GUID \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INVALID_GUID)

#define CSSMERR_CSSM_INVALID_CONTEXT_HANDLE \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INVALID_CONTEXT_HANDLE)

#define CSSMERR_CSSM_ INCOMPATIBLE_VERSION \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_INCOMPATIBLE_VERSION)

#define CSSMERR_CSSM_PRIVILEGE_NOT_GRANTED \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRCODE_PRIVILEGE_NOT_GRANTED)
```

6.5.2 CSSM Module-Specific Error Values

The first 16 CSSM Error Codes are reserved for general errors, see Appendix A on page 925.

```
#define CSSM_CSSM_BASE_CSSM_ERROR \
    (CSSM_CSSM_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT+0x10)
```

```
#define CSSMERR_CSSM_SCOPE_NOT_SUPPORTED (CSSM_CSSM_BASE_CSSM_ERROR+1)
```

Privilege scope requested is not supported in the platform

```
#define CSSMERR_CSSM_PVC_ALREADY_CONFIGURED (CSSM_CSSM_BASE_CSSM_ERROR+2)
```

PvcPolicy is already configured in the first call to CSSM_Init

```
#define CSSMERR_CSSM_INVALID_PVC  (CSSM_CSSM_BASE_CSSM_ERROR+3)
```

PvcPolicy requested is invalid

```
#define CSSMERR_CSSM_EMM_LOAD_FAILED (CSSM_CSSM_BASE_CSSM_ERROR+4)
```

EMM load failed

```
#define CSSMERR_CSSM_EMM_UNLOAD_FAILED (CSSM_CSSM_BASE_CSSM_ERROR+5)
```

EMM unload failed

```
#define CSSMERR_CSSM_ADDIN_LOAD_FAILED (CSSM_CSSM_BASE_CSSM_ERROR+6)
```

Addin Load function failed

```
#define CSSMERR_CSSM_INVALID_KEY_HIERARCHY (CSSM_CSSM_BASE_CSSM_ERROR+7)
```

Invalid key hierarchy requested

```
#define CSSMERR_CSSM_ADDIN_UNLOAD_FAILED (CSSM_CSSM_BASE_CSSM_ERROR+8)
```

Addin Unload function failed

```
#define CSSMERR_CSSM_LIB_REF_NOT_FOUND (CSSM_CSSM_BASE_CSSM_ERROR+9)
```

A reference to the loaded library cannot be obtained

```
#define CSSMERR_CSSM_INVALID_ADDIN_FUNCTION_TABLE \
    (CSSM_CSSM_BASE_CSSM_ERROR+10)
```

Addin function table registered with CSSM is invalid

```
#define CSSMERR_CSSM_EMM_AUTHENTICATE_FAILED \
    (CSSM_CSSM_BASE_CSSM_ERROR+11)
```

ModuleManager authentication failed

```
#define CSSMERR_CSSM_ADDIN_AUTHENTICATE_FAILED \
    (CSSM_CSSM_BASE_CSSM_ERROR+12)
```

Addin authenticate function failed

```
#define CSSMERR_CSSM_INVALID_SERVICE_MASK \
    (CSSM_CSSM_BASE_CSSM_ERROR+13)
```

Invalid service mask

```
#define CSSMERR_CSSM_MODULE_NOT_LOADED (CSSM_CSSM_BASE_CSSM_ERROR+14)
```

Module was not loaded

```
#define CSSMERR_CSSM_INVALID_SUBSERVICEID (CSSM_CSSM_BASE_CSSM_ERROR+15)
```

Invalid subservice Id was requested

```
#define CSSMERR_CSSM_BUFFER_TOO_SMALL (CSSM_CSSM_BASE_CSSM_ERROR+16)
```

Buffer size for the ModuleManagerGuids is less than the required size

```
#define CSSMERR_CSSM_INVALID_ATTRIBUTE (CSSM_CSSM_BASE_CSSM_ERROR+17)
```

Invalid attribute in context

```
#define CSSMERR_CSSM_ATTRIBUTE_NOT_IN_CONTEXT \
    (CSSM_CSSM_BASE_CSSM_ERROR+18)
```

Requested attribute is not in the context

```
#define CSSMERR_CSSM_MODULE_MANAGER_INITIALIZE_FAIL \
    (CSSM_CSSM_BASE_CSSM_ERROR+19)
```

ModuleManger initialize failed

```
#define CSSMERR_CSSM_MODULE_MANAGER_NOT_FOUND \
    (CSSM_CSSM_BASE_CSSM_ERROR+20)
```

ModuleManger to be notified is not loaded

```
#define CSSMERR_CSSM_EVENT_NOTIFICATION_CALLBACK_NOT_FOUND \
    (CSSM_CSSM_BASE_CSSM_ERROR+21)
```

Event Notification callback not found

6.6 Core Functions

The man-page definitions for CSSM Core functions are presented in this section.

NAME

CSSM_Init

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_Init(
    const CSSM_VERSION *Version,
    CSSM_PRIVILEGE_SCOPE Scope,
    const CSSM_GUID * CallerGuid,
    CSSM_KEY_HIERARCHY KeyHierarchy,
    CSSM_PVC_MODE *PvcPolicy,
    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 at least once by the application. It is an error to call any function of the CSSM API other than *CSSM_Init*() before a call to *CSSM_Init*() has returned successfully (that is, with CSSM_OK).

Implementations of CSSM may have platform specific characteristics associated with the implementation of *CSSM_SetPrivilege*() API. The privilege value may have thread specific scope or process specific scope. The application can specify the anticipated scope at *CSSM_Init*(). If the anticipated scope is not appropriate for the implementation, an error is returned. The scope can be configured only once. Subsequent attempts to configure scope are ignored.

The CSSM integrity model includes the ability to make and check assertions about trusted dynamically loaded libraries. Checking assertions happens while the program executes. It is known as Pointer Validation Checking (PVC). Pointer validation checking may be applied every time execution flow crosses the CSSM API or SPI interfaces.

Performing pointer validation checks has two purposes:

- It allows exportation of CSSM
- It aids in detering unanticipated run-time modification of the program

The CSSM can be configured to bypass pointer validation under some circumstances. Pointer validation cannot be bypassed when privileged operations are being performed.

The prerequisites for performing PVC on another module, be it service provider, CSSM, or other library, are:

- The module must have been signed and have an accompanying Signed Manifest
- The module must be loaded into process address space
- An entry-point into the module must be available

Typically, the entry-points are discovered when a module's functions are called by another module. The CSSM will perform pointer validation checks based on the configured checking policy. Checking policies are established by the manufacturers of CSSM and other libraries. The checking policy to be applied during execution is configured using the *CSSM_Init*() call. The policy can be configured once during the life of the process and occurs the first time *CSSM_Init*() is called.

PVC Policy Configuration Options

Pointer validation checking can be applied at the CSSM API interface, the CSSM SPI interface or both. The CSSM vendor can configure a default policy through instructions contained in the CSSM signed manifest. Manifest attributes pertaining to pointer validation checking are defined as follows:

Module	Tag	Value	Description
CSSM	CDSA_PVC_API	unspecified	CSSM will perform PVC checks at the API boundary
CSSM	CDSA_PVC_API	OFF	CSSM will not perform PVC checks at the API boundary
CSSM	CDSA_PVC_SPI	unspecified	CSSM will perform PVC checks at the SPI boundary
CSSM	CDSA_PVC_SPI	OFF	CSSM will not perform PVC checks at the SPI boundary
App	CDSA_PVC_API	EXEMPT	The calling module is allowed to override the CSSM policy for the API boundary
App	CDSA_PVC_API	unspecified	The calling module cannot weaken the CSSM API policy
App	CDSA_PVC_SPI	EXEMPT	The calling module is allowed to override the CSSM policy for the SPI boundary
App	CDSA_PVC_SPI	unspecified	The calling module cannot weaken the CSSM SPI policy

The *PvcPolicy* parameter to *CSSM_Init*() configures the run-time policy for the process. The *PvcPolicy* parameter is a bitmask allowing both API and SPI policies to be specified simultaneously. Unspecified policies default to the most conservative operational mode. The CSSM performs pointer validation checks unless explicitly disabled. Application modules may not override CSSM policy unless exemptions are explicitly granted. The following table shows the what policies may be configured for various manifest attribute values:

CSSM Manifest	Calling Module Manifest	Acceptable PvcPolicy Values
CDSA_PVC_API=<n/a>	CDSA_PVC_API=EXEMPT	API checks: off (0) or on (1)
CDSA_PVC_API=OFF	CDSA_PVC_API=EXEMPT	API checks: off (0) or on (1)
CDSA_PVC_API=<n/a>	CDSA_PVC_API=<n/a>	API checks: on (1)
CDSA_PVC_API=OFF	CDSA_PVC_API=<n/a>	API checks: off (0) or on (1)

The following table shows the *PvcPolicy* configuations available for the SPI:

SSM Manifest	Calling Module Manifest	Acceptable PvcPolicy Values
CDSA_PVC_SPI=<n/a>	CDSA_PVC_SPI=EXEMPT	SPI checks: off (0) or on (2)
CDSA_PVC_SPI =OFF	CDSA_PVC_SPI=EXEMPT	SPI checks: off (0) or on (2)
CDSA_PVC_SPI =<n/a>	CDSA_PVC_SPI=<n/a>	SPI checks: on (2)
CDSA_PVC_SPI =OFF	CDSA_PVC_SPI=<n/a>	SPI checks: off (0) or on (2)

If an application module does not have a manifest and CSSM requires the application module be subject to pointer validation checks, then pointer validation checks fail and CSSM will not operate with the anonymous module. All service provider modules are expected to have signed maniefests.

PARAMETERS

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

Scope (input)
The scope of the global privilege value. The scope may either process scope wide (CSSM_PRIVILEGE_SCOPE_PROCESS) or thread wide (CSSM_PRIVILEGE_SCOPE_THREAD). This parameter is ignored after the first call to *CSSM_Init*( ).

CallerGuid (input)
The GUID associated with the caller. This GUID is used to locate the caller's credentials when evaluating the request for privileges.

KeyHierarchy (input)
The CSSM_KEY_HIERARCHY flag directing CSSM what embedded key to use when verifying integrity of the named module.

PvcPolicy (input/output)
Configures the way in which pointer validation checks will be performed. If not the first call to *CSSM_Init*( ), the previously configured policy is returned in the *PvcPolicy* bitmask and the *CSSM_Init*( ) call continues processing. If successfully completed, the error code CSSMERR_CSSM_PVC_ALREADY_CONFIGURED is returned.

Value	Description
0	PVC validation is not performed
1	PVC validation is performed on application modules
2	PVC validation is performed on service provider modules
3	Both types of PVC validations are performed

Reserved (input)
A reserved input.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_SCOPE_NOT_SUPPORTED
CSSMERR_CSSM_PVC_ALREADY_CONFIGURED
CSSMERR_CSSM_INVALID_PVC

NAME

CSSM_Terminate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_Terminate ( )
```

DESCRIPTION

This function terminates the caller's use of CSSM. CSSM can cleanup all internal state associated with the calling application. This function must be called once by each application.

CSSM_Terminate( ) must be called one time for each time *CSSM_Init*( ) was previously called. CSSM services remain available to the program until the final call to *CSSM_Terminate*( ) completes. After that final call, all information introduced by the caller (including privileges, handles, contexts, introduced libraries, etc.) is lost, and it is an error to subsequently call any CSSM API function other than *CSSM_Init*( ).

PARAMETERS

None

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the General Error Codes and Common Error Codes and Values.

SEE ALSO

CSSM_Init( ).

6.7 Module Management Functions

The man-page definitions for Module Management functions are presented in this section.

NAME

CSSM_ModuleLoad

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleLoad
    (const CSSM_GUID *ModuleGuid,
    CSSM_KEY_HIERARCHY KeyHierarchy,
    CSSM_API_ModuleEventHandler AppNotifyCallback,
    void* AppNotifyCallbackCtx)
```

DESCRIPTION

This function initializes the security service service module. Initialization includes registering the application's module-event handler and enabling events with the security service service module. The application can choose to provide an event handler function to receive notification of insert, remove and fault events. The specified event handler is the single callback points for all attached sessions with the specified service module.

The function *CSSM_Init*( ) must be invoked prior to calling *CSSM_ModuleLoad*( ). The function *CSSM_ModuleAttach*( ) can be invoked multiple times per call to *CSSM_ModuleLoad*( ).

PARAMETERS

ModuleGuid (input)
The GUID of the module selected for loading.

KeyHierarchy (input)
The CSSM_KEY_HIERARCHY flag directing CSSM what embedded key to use when verifying integrity of the named module.

AppNotifyCallback (input/optional)
The event notification function provided by the caller. This defines the callback for event notifications from the loaded (and later attached) service module.

AppNotifyCallbackCtx (input/optional)
When the selected service module raises an event, this context is passed as an input to the event handler specified by *AppNotifyCallback*. CSSM does not interpret or modify the value of *AppNotifyCallbackCtx*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_GUID
CSSMERR_CSSM_ADDIN_LOAD_FAILED
CSSMERR_CSSM_EMM_LOAD_FAILED
CSSMERR_CSSM_INVALID_KEY_HIERARCHY

NAME

CSSM_ModuleUnload

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleUnload
    (const CSSM_GUID *ModuleGuid,
    CSSM_API_ModuleEventHandler AppNotifyCallback,
    void* AppNotifyCallbackCtx)
```

DESCRIPTION

The function deregisters event notification callbacks for the caller identified by *ModuleGuid*. *CSSM_ModuleLoad*( ) is the analog call to *CSSM_ModuleLoad*( ). If all registered callbacks registered with CSSM are removed, then CSSM unloads the service module that was loaded by calls to *CSSM_ModuleLoad*( ). Calls to *CSSM_ModuleUnload*( ) that are not matched with a previous call to *CSSM_ModuleLoad*( ) result in an error.

The CSSM uses the three input parameters *ModuleGuid*, *AppNotifyCallback,* and *AppNotifyCallbackCtx* to uniquely identify registered callbacks.

This function should be invoked after all necessary calls to *CSSM_ModuleDetach*( ) have been performed.

PARAMETERS

ModuleGuid (input)
The GUID of the module selected for unloading.

AppNotifyCallback (input/optional)
The event notification function to be de-registered. The function must have been provided by the caller in *CSSM_ModuleLoad*( ).

AppNotifyCallbackCtx (input/optional)
The event notification context that was provided in the corresponding call to *CSSM_ModuleLoad*( ).

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_ADDIN_UNLOAD_FAILED
CSSMERR_CSSM_EMM_UNLOAD_FAILED
CSSMERR_CSSM_EVENT_NOTIFICATION_CALLBACK_NOT_FOUND

NAME

CSSM_Introduce

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_Introduce
    (const CSSM_GUID *ModuleID,
    CSSM_KEY_HIERARCHY KeyHierarchy)
```

DESCRIPTION

The *CSSM_Introduce*( ) function identifies a dynamically loadable executable module (e.g. DLL) to the CSSM framework. CSSM uses the *ModuleID* information to locate the signed manifest and library on the host platform. The Module Directory Service (MDS) should be used to obtain the information. CSSM performs an integrity crosscheck on the module identified by *ModuleID* and caches the result in an internal structure. Integrity crosscheck uses the *KeyHierarchy* information to determine which classes of embedded public keys must serve as anchors when doing certificate path validation. If the export key hierarchy is specified, the set of export privileges contained in the manifest are retrieved from the manifest and saved with the integrity state information in the cache. Privileges granted to a module are accepted only if the manifest sections containing the privilege set have been signed by a principal in the export key hierarchy class and that hash of the module binary is part of the hash of the privilege attributes.

CSSM_Introduce( ) may be called at any time after *CSSM_Init*( ), by any module, on behalf of any module. Once a module is introduced into CSSM the load location of the module must not change. If the load location changes then the module must be re-introduced. Once introduced, the module load location, integrity and privilege information is held until *CSSM_Terminate*( ) is called or the process terminates. Initialization of internal data structures maintaining the table of introductions is performed when *CSSM_Init*( ) is called.

If *CSSM_Introduce*( ) is called on behalf of another module, then the caller needs to make sure that the other module is loaded into the process address space. If the library is already loaded into process address space, but a reference to the library cannot be obtained, a different error is returned (CSSMERR_CSSM_LIB_REF_NOT_FOUND).

PARAMETERS

ModuleID (input)

The CSSM_GUID of the calling library or other library that may call CDSA interfaces. The GUID is used to locate the signed manifest credentials of the named module to calculate module integrity information.

KeyHierarchy (input)

The CSSM_KEY_HIERARCHY flag directing CSSM what embedded key to use when verifying integrity of the named module.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_KEY_HIERARCHY
CSSMERR_CSSM_LIB_REF_NOT_FOUND

NAME

CSSM_Unintroduce

SYNOPSIS

```
CSSM_RETURN CSSM_Unintroduce
    (const CSSM_GUID *ModuleID)
```

DESCRIPTION

The *CSSM_Unintroduce*( ) function removes the module referenced by *ModuleID* from the list of module information maintained by the CSSM framework.

A caller may *un-introduce* modules other than itself if the caller has been previously introduced.

PARAMETERS

ModuleID (input)
The CSSM_GUID of the calling library or other library that may call CDSA interfaces. The GUID is used to locate the module integrity and privilege information. If the *ModuleID* is NULL then the caller will be "un-introduced".

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_GUID

NAME

CSSM_ModuleAttach

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleAttach
    (const CSSM_GUID *ModuleGuid,
    const CSSM_VERSION *Version,
    const CSSM_API_MEMORY_FUNCS *MemoryFuncs,
    uint32 SubserviceID,
    CSSM_SERVICE_TYPE SubServiceType,
    CSSM_ATTACH_FLAGS AttachFlags,
    CSSM_KEY_HIERARCHY KeyHierarchy,
    CSSM_FUNC_NAME_ADDR *FunctionTable,
    uint32 NumFunctionTable,
    const void *Reserved,
    CSSM_MODULE_HANDLE_PTR NewModuleHandle)
```

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.

If the sub-service is supported as part of the CSSM framework as well as by an EMM, *ModuleAttach* attaches the Service Provider to the CSSM framework. If the sub-service is only supported by an EMM, *ModuleAttach* loads the appropriate EMM. The Service Provider is given an indication of whether it is being attached to the CSSM framework or an EMM.

The caller may provide a function table containing function-names for the desired services. On output each function name is matched with an API function pointer. The caller can use the pointers to invoke service module operations through CSSM.

PARAMETERS

ModuleGuid (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 CDSA that the application is compatible with.

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

SubserviceID (input)
: A *SubServiceID* identifying a particular subservice within the module. Subservice IDs can be obtained from MDS or gleaned from insertion events reported through the *callback* function installed through *CSSM_ModuleLoad*(). Modules that provide only one service may use zero as their Subservice ID.

SubServiceType (input)
: A service mask describing the type of service the caller is requesting of the service provider module.

AttachFlags (input)
: A mask representing the caller's request for session-specific services.

KeyHierarchy (input)
The CSSM_KEY_HIERARCHY flag directing CSSM what embedded key to use when verifying integrity of the named module.

FunctionTable (input/output/optional)
A table of function-name and API function-pointer pairs. The caller provides the name of the functions as input. The corresponding API function pointers are returned on output. The function table allows dynamic linking of CDSA interfaces, including interfaces to Elective Module Managers, which are transparently loaded by CSSM during *CSSM_ModuleAttach*( ).

NumFunctionTable (input)
The number of entries in the *FunctionTable* parameter. If no *FunctionTable* is provided this value must be zero.

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

NewModuleHandle (output)
A new module handle that can be used to interact with the requested service provider. The value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_ADDIN_FUNCTION_TABLE
CSSMERR_CSSM_EMM_AUTHENTICATE_FAILED
CSSMERR_CSSM_ADDIN_AUTHENTICATE_FAILED
CSSMERR_CSSM_INVALID_SERVICE_MASK
CSSMERR_CSSM_MODULE_NOT_LOADED
CSSMERR_CSSM_INVALID_SUBSERVICEID
CSSMERR_CSSM_INVALID_KEY_HIERARCHY
CSSMERR_CSSM_INVALID_GUID

SEE ALSO

CSSM_ModuleDetach( )

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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See General Error Codes and Common Error Codes and Values.

SEE ALSO

CSSM_ModuleAttach()

NAME

CSSM_SetPrivilege

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetPrivilege
    (CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

The *CSSM_SetPrivilege*( ) function accepts as input a privilege value and stores it internal to the CSSM framework. The integrity credentials of the module calling *CSSM_SetPrivilege*( ) must be verified by CSSM before the privilege value is updated. Integrity credentials are established using *CSSM_Introduce*( ). CSSM will perform pointer validation check to ensure the caller has been previously introduced. *CSSM_SetPrivilege*( ) will fail if no integrity information can be found for the caller.

After pointer validation checks, CSSM verifies the requested privilege is authorized. This is done by comparing *Privilege* with the set of privileges contained in the caller manifest. If *Privilege* is not a member, the *CSSM_SetPrivilege*( ) call fails.

Subsequent calls to the framework that require privileges inherit the privilege value previously established by *CSSM_SetPrivilege*( ). The CSSM will perform pointer validation checks on the API caller before servicing the API call. If OK, then the *Privilege* value is supplied to the SPI function.

Internally, CSSM builds and maintains privilege information based on the chosen scope of the implementation. The scope may be dictated by the capabilities of the platform hosting the CSSM. If threading is available, the privilege value may be associated with the thread ID of the currently executing thread. In this scenario, CSSM may manage a table of tuples consisting of *threadID* and privilege value. If threading is not available, the privilege value may be global to the process.

Since the selected privilege value is shared, the application programmer should take precautions to reset the privilege value whenever program flow leaves the caller's module and again when control flow returns. In general, anytime there is a possibility for *CSSM_SetPrivilege*( ) to be called while within the context of the security critical section, *CSSM_SetPrivilege*( ) should be called again. Otherwise, the module receiving execution control could have called *CSSM_SetPrivilege*( ) resulting in the privilege value being reset.

Data structures used to maintain the global privilege value should be initialized in *CSSM_Init*( ). This includes lock initialization and preliminary resource allocation. *CSSM_Init*( ) is assumed to be idempotent with respect to shared structure initialization. This means *CSSM_Init*( ) will ensure a single thread initializes the shared structure and subsequent calls to *CSSM_Init*( ) will not re-initialize it. A reference count of calls to *CSSM_Init*( ) is needed to ensure matching calls to *CSSM_Terminate*( ) are handled.

Resource cleanup is performed at *CSSM_Terminate*( ), after the reference count falls to zero. The last call to *CSSM_Terminate*( ) results in shared resources being freed and lock structures being released.

PARAMETERS

Privilege (input)
The CSSM_PRIVILEGE value to be applied to subsequent calls to CSSM interfaces.

ERRORS

See the general error codes and common error codes and values section.

NAME

CSSM_GetPrivilege

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetPrivilege
    (CSSM_PRIVILEGE *Privilege);
```

DESCRIPTION

The *CSSM_GetPrivilege*() function returns the CSSM_PRIVILEGE value currently established in the framework.

PARAMETERS

Privilege (output)
The CSSM_PRIVILEGE value currently set.

ERRORS

See General Error Codes and Common Error Codes and Values sections.

NAME

CSSM_GetModuleGUIDFromHandle

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetModuleGUIDFromHandle
    (CSSM_MODULE_HANDLE ModuleHandle,
    CSSM_GUID_PTR ModuleGUID)
```

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.

ModuleGUID (output)
GUID of the module associated with *ModuleHandle.n*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See General Error Codes and Common Error Codes and Values sections.

SEE ALSO

CSSM_GetSubserviceUIDFromHandle( )

NAME

CSSM_GetSubserviceUIDFromHandle

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetSubserviceUIDFromHandle
    (CSSM_MODULE_HANDLE ModuleHandle,
    CSSM_SUBSERVICE_UID_PTR SubserviceUID)
```

DESCRIPTION

This function completes a structure containing the persistent unique identifier of the attached module subservice, as identified by the input handle.

PARAMETERS

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

SubserviceUID (output)
: Subservice UID value associated with *ModuleHandle*. The caller has to allocate the buffer.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See General Error Codes and Common Error Codes and Values sections.

SEE ALSO

CSSM_GetModuleGUIDFromHandle( )

6.8 EMM Module Management Functions

The man-page definition for the EMM Module Management function is presented in this section.

NAME

CSSM_ListAttachedModuleManagers

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ListAttachedModuleManagers
    (uint32 *NumberOfModuleManagers,
    CSSM_GUID_PTR ModuleManagerGuids)
```

DESCRIPTION

This function returns a list of GUIDs for the currently attached/active module managers in the CSSM environment.

PARAMETERS

NumberOfModuleManagers (input/output)
The number of GUIDs in the array. If the array is not large enough, then the actual number needed is returned and the error CSSMERR_CSSM_BUFFER_TOO_SMALL is returned. The caller should then allocate an appropriately-sized list and call the function again. If the supplied list is larger than needed, the number of module managers found is returned and no error is set.

ModuleManagerGuids (input/output)
A pointer to an array of CSSM_GUID structures, one per active module manager. The caller allocates this array.

RETURN VALUE

ERRORS

CSSMERR_CSSM_BUFFER_TOO_SMALL
CSSMERR_CSSM_INVALID_GUID

6.9 Utility Functions

The man-page definition for the Utility function is presented in this section.

NAME

CSSM_GetAPIMemoryFunctions

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetAPIMemoryFunctions
    (CSSM_MODULE_HANDLE AddInHandle,
    CSSM_API_MEMORY_FUNCS_PTR AppMemoryFuncs)
```

DESCRIPTION

This function retrieves the memory function table associated with the security service service module identified by the input handle.

PARAMETERS

AddInHandle (input)
The handle to the security service service module that is associated with the requested memory function table.

AppMemoryFuncs (output)
Pointer to an empty memory functions table. Upon function return, the table is filled with the memory function pointers associated with the specified attach handle. Caller has to allocate the buffer.

RETURN VALUE

CSSM_OK if the function was successful.

ERRORS

See General Error Codes and Common Error Codes and Values sections.

SEE ALSO

None.

Technical Standard

Part 3:

Cryptographic Service Providers (CSP)

The Open Group

Chapter 7

Cryptographic Services

7.1 Cryptographic Service Providers

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 (DL) 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 a *PassThrough* function.

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
- Secured storage of private keys

7.2 CDSA Add-In Modules

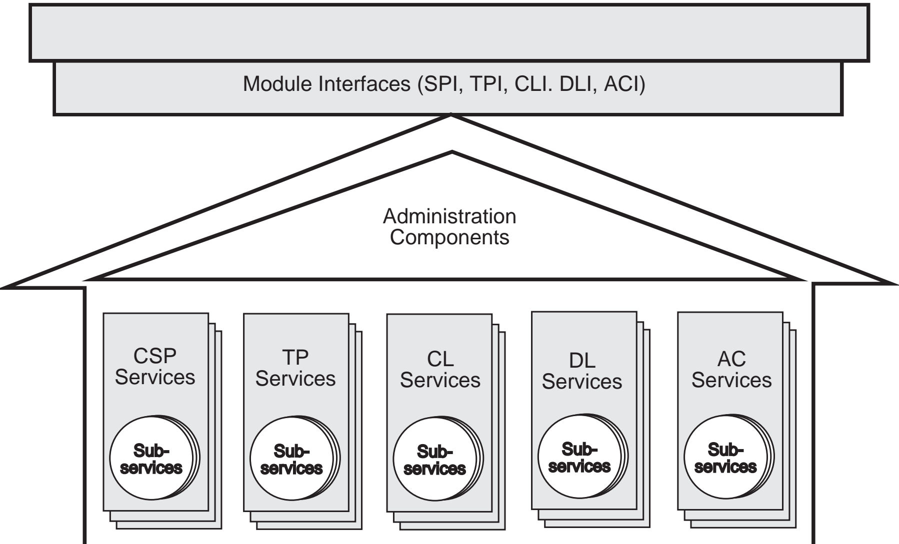


Figure 7-1 CDSA Add-In Module Structure

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. In this case it provides CSP Services.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions that 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 **Part 14** of this Technical Standard.

7.3 CDSA CSP Operation

7.3.1 CSSM Infrastructure

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:

- Support for dynamic module attach with integrity checking, and event notification handling
- Selection, initialization, and use of cryptographic operations, which are implemented by a CSP

In CDSA, all cryptographic services requested by applications are channeled to one of the CSPs via the CSSM. CSP vendors only need target their modules to CSSM for all security-conscious applications to gain access to their product.

Before an application calls a CSP to perform a cryptographic operation, the application uses the Module Directory Service (MDS) to determine what CSPs are available on the system, and what services they provide. Based on this information, the application then can determine which CSP to use for subsequent operations. An application establishes an *attach* session to select a particular CSP. Once attached, the application can initiate a cryptographic login session with the CSP. The application presents additional credentials, such as a passphrase or PIN, to gain access to specific keys and services managed by the CSP.

Within a module attach session or a cryptographic login session, an application creates, uses and discards cryptographic contexts. A cryptographic context carries the parameters required to perform a cryptographic service. The cryptographic context can be used for:

- A one-step cryptographic operation, in which only one call is needed to obtain the result.
- A cryptographic session of a multi-staged cryptographic service, in which there is an initialization call followed by one or more update calls, 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.

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. The cryptographic context contains most if not all of the input parameters required for an operation. Some of the cryptographic service functions accept input parameters in addition to the CSP handle and the context handle. These input parameters always take precedence over any duplicate or conflicting parameters in the cryptographic context.

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

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).

7.3.2 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 profile information stored in the Module Directory Services relation for CSPs, or return that information directly to applications upon request through the CSSM-defined API.

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.

7.3.3 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 higher-level interface, based on keys and keypairs associated with certificates. The Cryptographic Services Module Manager defines a 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 legacy 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 have a multi-threaded implementation.

7.3.4 CSP Registration

Each CSP registers a description of its basic functions and services with the Module Directory Services during CSP installation. Applications query this information to select appropriate CSPs for their use. CSPs with dynamic capabilities will not register all their information with MDS. CSPs can add information to MDS at runtime but CSP must also implement an API to query dynamic capabilities at runtime. 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 private keys and symmetric keys in tamper-resistant storage.

7.3.5 Cryptographic Services Operations

The security services API calls defined by the CSP Module Manager includes the following service categories:

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

Applications use these high-level concepts to provide authentication, data integrity, data and communication privacy, and non-repudiation of messages to 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

CSPs provide additional services:

- Unique identification number: hard-coded or random-generated
- Random number generator: attended and unattended
- Encrypted data: 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 a set of buffers)
- 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 *PassThrough* call. The *PassThrough* feature allows a highly knowledgeable application to call low-level CSP functions that are not available through the common Cryptographic API. The CSP Module Manager (CSPMM) first checks the authorization for the call, and if accepted the call is passed through to the specified 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.

7.3.6 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 (as required).

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 CSMM 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.

7.3.7 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-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 $512 \leq L \leq 1024$ 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.

A DSA wrapped, private key is represented as defined by the PKCS#8 specification. The PKCS#8 standard specifies the wrapped key format resulting from encoding an algorithm OID with an encoded private key.

7.3.8 Buffer Management for Cryptographic Services

Returning Buffers of Data

If data is returned in a buffer (CSSM_DATA), then the following behavior is defined:

1. If a data buffer is specified to receive the output value, then the value is written into the buffer if there is enough space and the length value is modified to equal the actual number of bytes written to the buffer. If there is not enough space to receive the output value, then the operation fails with the error code CSSM_CSP_ERR_OUTBUF_LENGTH. The length required for the output buffer can be determined by calling *CSSM_QuerySize*(). The state of the operation reverts to the state before the operation was attempted. The application can then allocate the correct number of bytes and retry the operation.
2. 2. If the Length field is set to 0 and the Data field is set to NULL, then the CSP will allocate the correct number of bytes and set the length accordingly.

Vector of Buffers

Many of the CSP APIs allow multiple input and output buffers to be manipulated simultaneously. The behavior of the CSP in these situations is as follows:

- The set of input buffers is treated as a single continuous logical buffer.
- If the input is transformed into a single output value at the end of the operation, then the value is returned as described for returning buffers of data.
- If the input buffers are transformed into a set of output buffers (i.e. encryption, decryption), then the result is returned in one of two ways:
  1. If output buffers are specified, then the buffers are filled to the given length with transformed data and the remaining data is placed in the next buffer. The lengths of the buffers are not modified. The total number of bytes is indicated by a separate return parameter that varies by API.
  2. If output buffers are not specified (an array of CSSM_DATA structures is passed with no buffer or length values), then the CSP may allocate as many output buffers of any length as it finds necessary up to the number of CSSM_DATA structures passed.
- Extra space in the output buffers is not "remembered" by the CSP for use in subsequent calls to an update API. New buffers must be supplied for each call.
- Data that does not fit into the output buffers or can not be returned immediately (that is, update calls to staged APIs) is returned using the *RemData* parameter. This value is treated as a single output value as described above.

7.4 Data Structures

7.4.1 CSSM_CC_HANDLE

```
typedef CSSM_LONG_HANDLE CSSM_CC_HANDLE; /* Cryptographic Context
                                        Handle */
```

7.4.2 CSSM_CSP_HANDLE

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

7.4.3 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.

7.4.4 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.

7.4.5 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.

7.4.6 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.

7.4.7 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 LogicalKeySizeInBits;   /* Logical key size in bits */
    uint32 EffectiveKeySizeInBits; /* Effective key size in bits */
} CSSM_KEY_SIZE, *CSSM_KEY_SIZE_PTR;
```

Definition

LogicalKeySizeInBits
The logical key size represents the actual size of the key in the case of symmetric algorithms (for example, DES, RC4, RC5) and the modulus size of the key in the case of asymmetric algorithms (for example, RSA, DSA).

EffectiveKeySizeInBits
The effective key size indicates the number of bits of keying material that will be used in the cryptographic operation. The following are instances where the effective key size is different from the logical key size for symmetric algorithms:

- Standard DES implementations take a 64-bit key (logical key size). However, one bit in each byte of the key is used for parity checking and hence the effective key size is 56 bits. Refer to FIPS 46-2 for details.

- Exportable DES implementations take a 64-bit key (logical key size). However, export regulations may mandate that the strength of the key be reduced (or dumbed down) to 40 bits; the effective key size in this case is 40 bits.

7.4.8 CSSM_KEYBLOB_TYPE

```
typedef uint32 CSSM_KEYBLOB_TYPE;
#define CSSM_KEYBLOB_RAW (0)          /* The blob is a clear, raw key */
#define CSSM_KEYBLOB_REFERENCE (1)    /* The blob is a reference to a key */
#define CSSM_KEYBLOB_WRAPPED (2)      /* The blob is a wrapped RAW key */
#define CSSM_KEYBLOB_OTHER (0xFFFFFFFF)
```

7.4.9 CSSM_KEYBLOB_FORMAT

```
typedef uint32 CSSM_KEYBLOB_FORMAT;

/* Raw Format */
#define CSSM_KEYBLOB_RAW_FORMAT_NONE (0)
                               /* No further conversion need to be done */
#define CSSM_KEYBLOB_RAW_FORMAT_PKCS1 (1) /* RSA PKCS1 V1.5 */
#define CSSM_KEYBLOB_RAW_FORMAT_PKCS3 (2) /* RSA PKCS3 V1.5 */
#define CSSM_KEYBLOB_RAW_FORMAT_MSCAPI (3) /* Microsoft CAPI V2.0 */
#define CSSM_KEYBLOB_RAW_FORMAT_PGP (4) /* PGP V */
#define CSSM_KEYBLOB_RAW_FORMAT_FIPS186 (5) /* US Gov. FIPS 186 - DSS V */
#define CSSM_KEYBLOB_RAW_FORMAT_BSAFE (6) /* RSA Bsafe V3.0 */
#define CSSM_KEYBLOB_RAW_FORMAT_CCA (9) /* CCA clear public key blob */
#define CSSM_KEYBLOB_RAW_FORMAT_PKCS8 (10) /* RSA PKCS8 V1.2 */
#define CSSM_KEYBLOB_RAW_FORMAT_SPKI (11) /* SPKI Specification */
#define CSSM_KEYBLOB_RAW_FORMAT_OCTET_STRING (12)
#define CSSM_KEYBLOB_RAW_FORMAT_OTHER (0xFFFFFFFF) /* Other, CSP defined */

/* Wrapped Format */
#define CSSM_KEYBLOB_WRAPPED_FORMAT_NONE (0)
                               /* No further conversion need to be done */
#define CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS8 (1) /* RSA PKCS8 V1.2 */
#define CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS7 (2)
> #define CSSM_KEYBLOB_WRAPPED_FORMAT_MSCAPI (3)
#define CSSM_KEYBLOB_WRAPPED_FORMAT_OTHER (0xFFFFFFFF) /* Other, CSP defined */

/* Reference Format */
#define CSSM_KEYBLOB_REF_FORMAT_INTEGER (0) /*Reference is a number or handle*/
#define CSSM_KEYBLOB_REF_FORMAT_STRING (1) /* Reference is a string or label */
#define CSSM_KEYBLOB_REF_FORMAT_SPKI (2) /* Reference is an SPKI S-expression*/
                                       /* to be evaluated to locate the key */
#define CSSM_KEYBLOB_REF_FORMAT_UNIQUE_ID (3)   /* A unique ID for the key */
                          /*relative to the data base in which it is created */
#define CSSM_KEYBLOB_REF_FORMAT_OTHER (0xFFFFFFFF)  /* Other, CSP defined */
```

7.4.10 CSSM_KEYCLASS

```
typedef uint32 CSSM_KEYCLASS;

#define CSSM_KEYCLASS_PUBLIC_KEY (0)   /* Key is public key */
#define CSSM_KEYCLASS_PRIVATE_KEY (1)  /* Key is private key */
#define CSSM_KEYCLASS_SESSION_KEY (2)  /* Key is session or symmetric key */
#define CSSM_KEYCLASS_SECRET_PART (3)  /* Key is part of secret key */

#define CSSM_KEYCLASS_OTHER (0xFFFFFFFF) /* Other */
```

7.4.11 CSSM_KEYATTR_FLAGS

```
typedef uint32 CSSM_KEYATTR_FLAGS;

/* Valid only during call to an API. Will never be valid when set
   in a key header */
#define CSSM_KEYATTR_RETURN_DEFAULT     (0x00000000)
#define CSSM_KEYATTR_RETURN_DATA        (0x10000000)
#define CSSM_KEYATTR_RETURN_REF         (0x20000000)
#define CSSM_KEYATTR_RETURN_NONE        (0x40000000)

/* Valid during an API call and in a key header */
#define CSSM_KEYATTR_PERMANENT          (0x00000001)
#define CSSM_KEYATTR_PRIVATE            (0x00000002)
#define CSSM_KEYATTR_MODIFIABLE         (0x00000004)
#define CSSM_KEYATTR_SENSITIVE          (0x00000008)
#define CSSM_KEYATTR_EXTRACTABLE        (0x00000020)

/* Valid only in a key header generated by a CSP, not valid during
   an API call */
#define CSSM_KEYATTR_ALWAYS_SENSITIVE   (0x00000010)
#define CSSM_KEYATTR_NEVER_EXTRACTABLE  (0x00000040)
```

7.4.12 CSSM_KEYUSE

```
typedef uint32 CSSM_KEYUSE;

#define CSSM_KEYUSE_ANY                 (0x80000000)
#define CSSM_KEYUSE_ENCRYPT             (0x00000001)
#define CSSM_KEYUSE_DECRYPT             (0x00000002)
#define CSSM_KEYUSE_SIGN                (0x00000004)
#define CSSM_KEYUSE_VERIFY              (0x00000008)
#define CSSM_KEYUSE_SIGN_RECOVER        (0x00000010)
#define CSSM_KEYUSE_VERIFY_RECOVER      (0x00000020)
#define CSSM_KEYUSE_WRAP                (0x00000040)
#define CSSM_KEYUSE_UNWRAP              (0x00000080)
#define CSSM_KEYUSE_DERIVE              (0x00000100)
```

7.4.13 CSSM_KEYHEADER

The key header contains meta-data about a key. It contains the GUID of the CSP that owns the data. The CSP initializes the values stored in the key header and returns the header to the application as part of the CSSM_KEY structure. The application can use this key structure as input to other CSP functions. It is highly recommended that applications do not directly alter the values stored in the key header. The cryptographic service function receiving the directly modified key header as an input parameter will typically fail and return an error indicating that the key is invalid.

The key header attributes describe both the CSP-stored copy of the key and the application's local ...

Note: Editor's note: this update to be completed/resolved.

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 */
    CSSM_KEYBLOB_TYPE BlobType; /* See BlobType #define's */
    CSSM_KEYBLOB_FORMAT Format; /* Raw or Reference format */
    CSSM_ALGORITHMS AlgorithmId; /* Algorithm ID of key */
    CSSM_KEYCLASS KeyClass; /* Public/Private/Secret, etc. */
    uint32 LogicalKeySizeInBits; /* Logical key size in bits */
    CSSM_KEYATTR_FLAGS KeyAttr; /* Attribute flags */
    CSSM_KEYUSE KeyUsage; /* Key use flags */
    CSSM_DATE StartDate; /* Effective date of key */
    CSSM_DATE EndDate; /* Expiration date of key */
    CSSM_ALGORITHMS WrapAlgorithmId; /* == CSSM_ALGID_NONE if clear key */
    CSSM_ENCRYPT_MODE 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 key
CSSM_KEYBLOB_OTHER	The blob is CSP specific

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. When a CSSM_KEYBLOB_RAW_FORMAT identifier is used, the key bits are present (in the specified representation) in the *KeyData* field of the CSSM_KEY structure. When a CSSM_KEYBLOB_REF_FORMAT identifier is used, the KeyData field of the CSSM_KEY structure contains the reference associated with the key bits. This reference can be a

number, handle, string, label, or CSP-specific format. Any of the following values are valid as format identifiers.

Keyblob Format Identifier	Description
CSSM_KEYBLOB_RAW_FORMAT_NONE	Raw format is none
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_CCA	CCA clear public key blob
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_SPKI	SPKI Specification *http://www.pobox.com/~cme/theory.txt*
CSSM_KEYBLOB_RAW_FORMAT_OTHER	Other, CSP defined
CSSM_KEYBLOB_WRAPPED_FORMAT_NONE	No further conversion needs to be performed
CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS8	PKCS8 V1.2: See "Private-Key Information Syntax Standard", an RSA Laboratories publication http://www.rsa.com/rsalabs/pubs/PKCS/
CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS5	PKCS5 V1.5 PBE scheme
CSSM_KEYBLOB_WRAPPED_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 label
CSSM_KEYBLOB_REF_FORMAT_SPKI	Reference is an SPKI S-expression to be evaluated to locate the key
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

LogicalKeySizeInBits

The logical key size represents the actual size of the key in the case of symmetric algorithms (for example, DES, RC4, RC5), and the modulus size of the key in the case of asymmetric algorithms (for example, RSA, DSA).

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_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_PRIVATE_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
LogicalKeySizeInBits	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	

7.4.14 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.

7.4.15 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;
```

7.4.16 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;
```

7.4.17 CSSM_CONTEXT_TYPE

This type defines a set of constants used different classes of cryptographic algorithms. Each algorithm class corresponds to a context type. See the definition of CSSM_CONTEXT for a description of each algorithm class value.

```
typedef uint32 CSSM_CONTEXT_TYPE;

#define CSSM_ALGCLASS_NONE      (0)
#define CSSM_ALGCLASS_CUSTOM    (CSSM_ALGCLASS_NONE+1)
#define CSSM_ALGCLASS_SIGNATURE (CSSM_ALGCLASS_NONE+2)
#define CSSM_ALGCLASS_SYMMETRIC (CSSM_ALGCLASS_NONE+3)
#define CSSM_ALGCLASS_DIGEST    (CSSM_ALGCLASS_NONE+4)
#define CSSM_ALGCLASS_RANDOMGEN (CSSM_ALGCLASS_NONE+5)
#define CSSM_ALGCLASS_UNIQUEGEN (CSSM_ALGCLASS_NONE+6)
#define CSSM_ALGCLASS_MAC       (CSSM_ALGCLASS_NONE+7)
#define CSSM_ALGCLASS_ASYMMETRIC(CSSM_ALGCLASS_NONE+8)
#define CSSM_ALGCLASS_KEYGEN    (CSSM_ALGCLASS_NONE+9)
#define CSSM_ALGCLASS_DERIVEKEY (CSSM_ALGCLASS_NONE+10)
```

7.4.18 CSSM Algorithms

This type defines a set of constants used to identify cryptographic algorithms. See the definition of CSSM_CONTEXT for a description of each algorithm value.

```
typedef uint32 CSSM_ALGORITHMS;

#define CSSM_ALGID_NONE        (0)
#define CSSM_ALGID_CUSTOM    (CSSM_ALGID_NONE+1)
#define CSSM_ALGID_DH        (CSSM_ALGID_NONE+2)
#define CSSM_ALGID_PH        (CSSM_ALGID_NONE+3)
#define CSSM_ALGID_KEA       (CSSM_ALGID_NONE+4)
#define CSSM_ALGID_MD2       (CSSM_ALGID_NONE+5)
#define CSSM_ALGID_MD4       (CSSM_ALGID_NONE+6)
#define CSSM_ALGID_MD5       (CSSM_ALGID_NONE+7)
#define CSSM_ALGID_SHA1      (CSSM_ALGID_NONE+8)
#define CSSM_ALGID_NHASH     (CSSM_ALGID_NONE+9)
#define CSSM_ALGID_HAVAL     (CSSM_ALGID_NONE+10)
#define CSSM_ALGID_RIPEMD    (CSSM_ALGID_NONE+11)
#define CSSM_ALGID_IBCHASH   (CSSM_ALGID_NONE+12)
#define CSSM_ALGID_RIPEMAC   (CSSM_ALGID_NONE+13)
#define CSSM_ALGID_DES       (CSSM_ALGID_NONE+14)
#define CSSM_ALGID_DESX      (CSSM_ALGID_NONE+15)
#define CSSM_ALGID_RDES      (CSSM_ALGID_NONE+16)
#define CSSM_ALGID_3DES_3KEY_EDE (CSSM_ALGID_NONE+17)
#define CSSM_ALGID_3DES_2KEY_EDE (CSSM_ALGID_NONE+18)
#define CSSM_ALGID_3DES_1KEY_EEE (CSSM_ALGID_NONE+19)
#define CSSM_ALGID_3DES_3KEY  CSSM_ALGID_3DES_3KEY_EDE
#define CSSM_ALGID_3DES_3KEY_EEE (CSSM_ALGID_NONE+20)
#define CSSM_ALGID_3DES_2KEY  CSSM_ALGID_3DES_2KEY_EDE
#define CSSM_ALGID_3DES_2KEY_EEE (CSSM_ALGID_NONE+21)
#define CSSM_ALGID_IDEA      (CSSM_ALGID_NONE+22)
#define CSSM_ALGID_RC2       (CSSM_ALGID_NONE+23)
#define CSSM_ALGID_RC5       (CSSM_ALGID_NONE+24)
#define CSSM_ALGID_RC4       (CSSM_ALGID_NONE+25)
#define CSSM_ALGID_SEAL      (CSSM_ALGID_NONE+26)
#define CSSM_ALGID_CAST      (CSSM_ALGID_NONE+27)
#define CSSM_ALGID_BLOWFISH  (CSSM_ALGID_NONE+28)
#define CSSM_ALGID_SKIPJACK  (CSSM_ALGID_NONE+29)
#define CSSM_ALGID_LUCIFER   (CSSM_ALGID_NONE+30)
```

```
#define CSSM_ALGID_MADRYGA   (CSSM_ALGID_NONE+31)
#define CSSM_ALGID_FEAL      (CSSM_ALGID_NONE+32)
#define CSSM_ALGID_REDOC     (CSSM_ALGID_NONE+33)
#define CSSM_ALGID_REDOC3    (CSSM_ALGID_NONE+34)
#define CSSM_ALGID_LOKI      (CSSM_ALGID_NONE+35)
#define CSSM_ALGID_KHUFU     (CSSM_ALGID_NONE+36)
#define CSSM_ALGID_KHAFRE    (CSSM_ALGID_NONE+37)
#define CSSM_ALGID_MMB       (CSSM_ALGID_NONE+38)
#define CSSM_ALGID_GOST      (CSSM_ALGID_NONE+39)
#define CSSM_ALGID_SAFER     (CSSM_ALGID_NONE+40)
#define CSSM_ALGID_CRAB      (CSSM_ALGID_NONE+41)
#define CSSM_ALGID_RSA       (CSSM_ALGID_NONE+42)
#define CSSM_ALGID_DSA       (CSSM_ALGID_NONE+43)
#define CSSM_ALGID_MD5WithRSA(CSSM_ALGID_NONE+44)
#define CSSM_ALGID_MD2WithRSA(CSSM_ALGID_NONE+45)
#define CSSM_ALGID_ElGamal   (CSSM_ALGID_NONE+46)
#define CSSM_ALGID_MD2Random (CSSM_ALGID_NONE+47)
#define CSSM_ALGID_MD5Random (CSSM_ALGID_NONE+48)
#define CSSM_ALGID_SHARandom (CSSM_ALGID_NONE+49)
#define CSSM_ALGID_DESRandom (CSSM_ALGID_NONE+50)
#define CSSM_ALGID_SHA1WithRSA (CSSM_ALGID_NONE+51)
#define CSSM_ALGID_CDMF      (CSSM_ALGID_NONE+52)
#define CSSM_ALGID_CAST3     (CSSM_ALGID_NONE+53)
#define CSSM_ALGID_CAST5     (CSSM_ALGID_NONE+54)
#define CSSM_ALGID_GenericSecret (CSSM_ALGID_NONE+55)
#define CSSM_ALGID_ConcatBaseAndKey (CSSM_ALGID_NONE+56)
#define CSSM_ALGID_ConcatKeyAndBase (CSSM_ALGID_NONE+57)
#define CSSM_ALGID_ConcatBaseAndData (CSSM_ALGID_NONE+58)
#define CSSM_ALGID_ConcatDataAndBase (CSSM_ALGID_NONE+59)
#define CSSM_ALGID_XORBaseAndData (CSSM_ALGID_NONE+60)
#define CSSM_ALGID_ExtractFromKey (CSSM_ALGID_NONE+61)
#define CSSM_ALGID_SSL3PreMasterGen (CSSM_ALGID_NONE+62)
#define CSSM_ALGID_SSL3MasterDerive (CSSM_ALGID_NONE+63)
#define CSSM_ALGID_SSL3KeyAndMacDerive (CSSM_ALGID_NONE+64)
#define CSSM_ALGID_SSL3MD5_MAC (CSSM_ALGID_NONE+65)
#define CSSM_ALGID_SSL3SHA1_MAC (CSSM_ALGID_NONE+66)
#define CSSM_ALGID_PKCS5_PBKDF1_MD5 (CSSM_ALGID_NONE+67)
#define CSSM_ALGID_PKCS5_PBKDF1_MD2 (CSSM_ALGID_NONE+68)
#define CSSM_ALGID_PKCS5_PBKDF1_SHA1 (CSSM_ALGID_NONE+69)
#define CSSM_ALGID_WrapLynks (CSSM_ALGID_NONE+70)
#define CSSM_ALGID_WrapSET_OAEP (CSSM_ALGID_NONE+71)
#define CSSM_ALGID_BATON     (CSSM_ALGID_NONE+72)
#define CSSM_ALGID_ECDSA     (CSSM_ALGID_NONE+73)
#define CSSM_ALGID_MAYFLY    (CSSM_ALGID_NONE+74)
#define CSSM_ALGID_JUNIPER   (CSSM_ALGID_NONE+75)
#define CSSM_ALGID_FASTHASH  (CSSM_ALGID_NONE+76)
#define CSSM_ALGID_3DES      (CSSM_ALGID_NONE+77)
#define CSSM_ALGID_SSL3MD5   (CSSM_ALGID_NONE+78)
#define CSSM_ALGID_SSL3SHA1  (CSSM_ALGID_NONE+79)
#define CSSM_ALGID_FortezzaTimestamp (CSSM_ALGID_NONE+80)
#define CSSM_ALGID_SHA1WithDSA (CSSM_ALGID_NONE+81)
#define CSSM_ALGID_SHA1WithECDSA (CSSM_ALGID_NONE+82)
#define CSSM_ALGID_DSA_BSAFE (CSSM_ALGID_NONE+83)
#define CSSM_ALGID_ECDH      (CSSM_ALGID_NONE+84)
#define CSSM_ALGID_ECMQV     (CSSM_ALGID_NONE+85)
#define CSSM_ALGID_PKCS12_SHA1_PBE (CSSM_ALGID_NONE+86)
#define CSSM_ALGID_ECNRA     (CSSM_ALGID_NONE+87)
#define CSSM_ALGID_SHA1WithECNRA (CSSM_ALGID_NONE+88)
#define CSSM_ALGID_ECES      (CSSM_ALGID_NONE+89)
```

```
#define CSSM_ALGID_ECAES     (CSSM_ALGID_NONE+90)
#define CSSM_ALGID_SHA1HMAC  (CSSM_ALGID_NONE+91)
#define CSSM_ALGID_FIPS186Random (CSSM_ALGID_NONE+92)
#define CSSM_ALGID_ECC       (CSSM_ALGID_NONE+93)
#define CSSM_ALGID_MQV       (CSSM_ALGID_NONE+94)
#define CSSM_ALGID_NRA       (CSSM_ALGID_NONE+95)
#define CSSM_ALGID_IntelPlatformRandom (CSSM_ALGID_NONE+96)
#define CSSM_ALGID_UTC       (CSSM_ALGID_NONE+97)
#define CSSM_ALGID_HAVAL3    (CSSM_ALGID_NONE+98)
#define CSSM_ALGID_HAVAL4    (CSSM_ALGID_NONE+99)
#define CSSM_ALGID_HAVAL5    (CSSM_ALGID_NONE+100)
#define CSSM_ALGID_TIGER     (CSSM_ALGID_NONE+101)
#define CSSM_ALGID_MD5HMAC   (CSSM_ALGID_NONE+102)
#define CSSM_ALGID_PKCS5_PBKDF2 (CSSM_ALGID_NONE+103)
#define CSSM_ALGID_RUNNING_COUNTER (CSSM_ALGID_NONE+104)
#define CSSM_ALGID_LAST      (0x7FFFFFFF)

/*
 * All algorithms IDs that are vendor specific, and not
 * part of the CSSM specification should be defined relative
 * to CSSM_ALGID_VENDOR_DEFINED.
 */
#define CSSM_ALGID_VENDOR_DEFINED (CSSM_ALGID_NONE+0x80000000)
```

7.4.19 CSSM_ATTRIBUTE_TYPE

This type defines a set of constants used to identify the types of attributes store in a cryptographic context.

```
/* Attribute data type tags */
#define CSSM_ATTRIBUTE_DATA_NONE                  (0x00000000)
#define CSSM_ATTRIBUTE_DATA_UINT32                (0x10000000)
#define CSSM_ATTRIBUTE_DATA_CSSM_DATA             (0x20000000)
#define CSSM_ATTRIBUTE_DATA_CRYPTO_DATA           (0x30000000)
#define CSSM_ATTRIBUTE_DATA_KEY                   (0x40000000)
#define CSSM_ATTRIBUTE_DATA_STRING                (0x50000000)
#define CSSM_ATTRIBUTE_DATA_DATE                  (0x60000000)
#define CSSM_ATTRIBUTE_DATA_RANGE                 (0x70000000)
#define CSSM_ATTRIBUTE_DATA_ACCESS_CREDENTIALS (0x80000000)
#define CSSM_ATTRIBUTE_DATA_VERSION               (0x01000000)
#define CSSM_ATTRIBUTE_DATA_DL_DB_HANDLE          (0x02000000)

#define CSSM_ATTRIBUTE_TYPE_MASK                  (0xFF000000)

typedef enum cssm_attribute_type {
    CSSM_ATTRIBUTE_NONE         = 0,
    CSSM_ATTRIBUTE_CUSTOM       = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 1,
    CSSM_ATTRIBUTE_DESCRIPTION  = CSSM_ATTRIBUTE_DATA_STRING | 2,
    CSSM_ATTRIBUTE_KEY          = CSSM_ATTRIBUTE_DATA_KEY | 3,
    CSSM_ATTRIBUTE_INIT_VECTOR  = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 4,
    CSSM_ATTRIBUTE_SALT         = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 5,
    CSSM_ATTRIBUTE_PADDING      = CSSM_ATTRIBUTE_DATA_UINT32 | 6,
    CSSM_ATTRIBUTE_RANDOM       = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 7,
    CSSM_ATTRIBUTE_SEED         = CSSM_ATTRIBUTE_DATA_CRYPTO_DATA | 8,
    CSSM_ATTRIBUTE_PASSPHRASE   = CSSM_ATTRIBUTE_DATA_CRYPTO_DATA | 9,
    CSSM_ATTRIBUTE_KEY_LENGTH   = CSSM_ATTRIBUTE_DATA_UINT32 | 10,
    CSSM_ATTRIBUTE_KEY_LENGTH_RANGE = CSSM_ATTRIBUTE_DATA_RANGE | 11,
    CSSM_ATTRIBUTE_BLOCK_SIZE   = CSSM_ATTRIBUTE_DATA_UINT32 | 12,
    CSSM_ATTRIBUTE_OUTPUT_SIZE  = CSSM_ATTRIBUTE_DATA_UINT32 | 13,
    CSSM_ATTRIBUTE_ROUNDS       = CSSM_ATTRIBUTE_DATA_UINT32 | 14,
    CSSM_ATTRIBUTE_IV_SIZE      = CSSM_ATTRIBUTE_DATA_UINT32 | 15,
    CSSM_ATTRIBUTE_ALG_PARAMS   = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 16,
    CSSM_ATTRIBUTE_LABEL        = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 17,
    CSSM_ATTRIBUTE_KEY_TYPE     = CSSM_ATTRIBUTE_DATA_UINT32 | 18,
    CSSM_ATTRIBUTE_MODE         = CSSM_ATTRIBUTE_DATA_UINT32 | 19,
    CSSM_ATTRIBUTE_EFFECTIVE_BITS = CSSM_ATTRIBUTE_DATA_UINT32 | 20,
    CSSM_ATTRIBUTE_START_DATE   = CSSM_ATTRIBUTE_DATA_DATE | 21,
    CSSM_ATTRIBUTE_END_DATE     = CSSM_ATTRIBUTE_DATA_DATE | 22,
    CSSM_ATTRIBUTE_KEYUSAGE     = CSSM_ATTRIBUTE_DATA_UINT32 | 23,
    CSSM_ATTRIBUTE_KEYATTR      = CSSM_ATTRIBUTE_DATA_UINT32 | 24,
    CSSM_ATTRIBUTE_VERSION      = CSSM_ATTRIBUTE_DATA_VERSION | 25,
    CSSM_ATTRIBUTE_PRIME        = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 26,
    CSSM_ATTRIBUTE_BASE         = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 27,
    CSSM_ATTRIBUTE_SUBPRIME     = CSSM_ATTRIBUTE_DATA_CSSM_DATA | 28,
    CSSM_ATTRIBUTE_ALG_ID       = CSSM_ATTRIBUTE_DATA_UINT32 | 29,
    CSSM_ATTRIBUTE_ITERATION_COUNT = CSSM_ATTRIBUTE_DATA_UINT32 | 30,
    CSSM_ATTRIBUTE_ROUNDS_RANGE = CSSM_ATTRIBUTE_DATA_RANGE | 31,
    CSSM_ATTRIBUTE_CSP_HANDLE   = CSSM_ATTRIBUTE_DATA_UINT32 | 34,
    CSSM_ATTRIBUTE_DL_DB_HANDLE = CSSM_ATTRIBUTE_DATA_DL_DB_HANDLE | 35,
    CSSM_ATTRIBUTE_ACCESS_CREDENTIALS =
                                  CSSM_ATTRIBUTE_DATA_ACCESS_CREDENTIALS | 36,
    CSSM_ATTRIBUTE_PUBLIC_KEY_FORMAT = CSSM_ATTRIBUTE_DATA_UINT32 | 37,
    CSSM_ATTRIBUTE_PRIVATE_KEY_FORMAT = CSSM_ATTRIBUTE_DATA_UINT32 | 38,
```

```
    CSSM_ATTRIBUTE_SYMMETRIC_KEY_FORMAT = CSSM_ATTRIBUTE_DATA_UINT32 | 39,
    CSSM_ATTRIBUTE_WRAPPED_KEY_FORMAT = CSSM_ATTRIBUTE_DATA_UINT32 | 40,
} CSSM_ATTRIBUTE_TYPE;
```

7.4.20 CSSM_ENCRYPT_MODE

This type defines a set of constants used to identify encryption modes used by different cryptographic algorithms. See the definition of CSSM_CONTEXT for a description of each encryption mode.

```
typedef uint32 CSSM_ENCRYPT_MODE

#define CSSM_ALGMODE_NONE            (0)
#define CSSM_ALGMODE_CUSTOM          (CSSM_ALGMODE_NONE+1)
#define CSSM_ALGMODE_ECB             (CSSM_ALGMODE_NONE+2)
#define CSSM_ALGMODE_ECBPad          (CSSM_ALGMODE_NONE+3)
#define CSSM_ALGMODE_CBC             (CSSM_ALGMODE_NONE+4)
#define CSSM_ALGMODE_CBC_IV8         (CSSM_ALGMODE_NONE+5)
#define CSSM_ALGMODE_CBCPadIV8       (CSSM_ALGMODE_NONE+6)
#define CSSM_ALGMODE_CFB             (CSSM_ALGMODE_NONE+7)
#define CSSM_ALGMODE_CFB_IV8         (CSSM_ALGMODE_NONE+8)
#define CSSM_ALGMODE_CFBPadIV8       (CSSM_ALGMODE_NONE+9)
#define CSSM_ALGMODE_OFB             (CSSM_ALGMODE_NONE+10)
#define CSSM_ALGMODE_OFB_IV8         (CSSM_ALGMODE_NONE+11)
#define CSSM_ALGMODE_OFBPadIV8       (CSSM_ALGMODE_NONE+12)
#define CSSM_ALGMODE_COUNTER         (CSSM_ALGMODE_NONE+13)
#define CSSM_ALGMODE_BC              (CSSM_ALGMODE_NONE+14)
#define CSSM_ALGMODE_PCBC            (CSSM_ALGMODE_NONE+15)
#define CSSM_ALGMODE_CBCC            (CSSM_ALGMODE_NONE+16)
#define CSSM_ALGMODE_OFBNLF          (CSSM_ALGMODE_NONE+17)
#define CSSM_ALGMODE_PBC             (CSSM_ALGMODE_NONE+18)
#define CSSM_ALGMODE_PFB             (CSSM_ALGMODE_NONE+19)
#define CSSM_ALGMODE_CBCPD           (CSSM_ALGMODE_NONE+20)
#define CSSM_ALGMODE_PUBLIC_KEY      (CSSM_ALGMODE_NONE+21)
#define CSSM_ALGMODE_PRIVATE_KEY     (CSSM_ALGMODE_NONE+22)
#define CSSM_ALGMODE_SHUFFLE         (CSSM_ALGMODE_NONE+23)
#define CSSM_ALGMODE_ECB64           (CSSM_ALGMODE_NONE+24)
#define CSSM_ALGMODE_CBC64           (CSSM_ALGMODE_NONE+25)
#define CSSM_ALGMODE_OFB64           (CSSM_ALGMODE_NONE+26)
#define CSSM_ALGMODE_CFB32           (CSSM_ALGMODE_NONE+28)
#define CSSM_ALGMODE_CFB16           (CSSM_ALGMODE_NONE+29)
#define CSSM_ALGMODE_CFB8            (CSSM_ALGMODE_NONE+30)
#define CSSM_ALGMODE_WRAP            (CSSM_ALGMODE_NONE+31)
#define CSSM_ALGMODE_PRIVATE_WRAP    (CSSM_ALGMODE_NONE+32)
#define CSSM_ALGMODE_RELAYX          (CSSM_ALGMODE_NONE+33)
#define CSSM_ALGMODE_ECB128          (CSSM_ALGMODE_NONE+34)
#define CSSM_ALGMODE_ECB96           (CSSM_ALGMODE_NONE+35)
#define CSSM_ALGMODE_CBC128          (CSSM_ALGMODE_NONE+36)
#define CSSM_ALGMODE_OAEP_HASH       (CSSM_ALGMODE_NONE+37)
#define CSSM_ALGMODE_PKCS1_EME_V15   (CSSM_ALGMODE_NONE+38)
#define CSSM_ALGMODE_PKCS1_EME_OAEP  (CSSM_ALGMODE_NONE+39)
#define CSSM_ALGMODE_PKCS1_EMSA_V15  (CSSM_ALGMODE_NONE+40)
#define CSSM_ALGMODE_ISO_9796        (CSSM_ALGMODE_NONE+41)
#define CSSM_ALGMODE_X9_31           (CSSM_ALGMODE_NONE+42)
#define CSSM_ALGMODE_LAST            (0x7FFFFFFF)

/*
 * All algorithms modes that are vendor specific, and
 * not part of the CSSM specification should be defined
```

```
 * relative to CSSM_ALGMODE_VENDOR_DEFINED.
 */
#define CSSM_ALGMODE_VENDOR_DEFINED(CSSM_ALGMODE_NONE+0x80000000)
```

7.4.21 CSSM_PADDING

This type defines a set of constants used to identify padding methods used by different encryption algorithms.

```
typedef uint32 CSSM_PADDING

#define CSSM_PADDING_NONE           (0)
#define CSSM_PADDING_CUSTOM         (CSSM_PADDING_NONE+1)
#define CSSM_PADDING_ZERO           (CSSM_PADDING_NONE+2)
#define CSSM_PADDING_ONE            (CSSM_PADDING_NONE+3)
#define CSSM_PADDING_ALTERNATE      (CSSM_PADDING_NONE+4)
#define CSSM_PADDING_FF             (CSSM_PADDING_NONE+5)
#define CSSM_PADDING_PKCS5          (CSSM_PADDING_NONE+6)
#define CSSM_PADDING_PKCS7          (CSSM_PADDING_NONE+7)
#define CSSM_PADDING_CIPHERSTEALING (CSSM_PADDING_NONE+8)
#define CSSM_PADDING_RANDOM         (CSSM_PADDING_NONE+9)
#define CSSM_PADDING_PKCS1          (CSSM_PADDING_NONE+10)

/*
 * All padding types that are vendor specific, and not
 * part of the CSSM specification should be defined
 * relative to CSSM_PADDING_VENDOR_DEFINED.
 */
#define CSSM_PADDING_VENDOR_DEFINED(CSSM_PADDING_NONE+0x80000000)
```

7.4.22 CSSM_KEY_TYPE

```
typedef CSSM_ALGORITHMS CSSM_KEY_TYPE;
```

Definition

The cryptographic key type is represented by the cryptographic algorithm where the key will be used. The Cryptographic Service Provider must interpret the algorithm type and deduce the key type required for that algorithm.

7.4.23 CSSM_CONTEXT_ATTRIBUTE

```
typedef struct cssm_context_attribute{
    CSSM_ATTRIBUTE_TYPE AttributeType;
    uint32 AttributeLength;
    union cssm_context_attribute_value{
        char *String;
        uint32 Uint32;
        CSSM_ACCESS_CREDENTIALS_PTR AccessCredentials;
        CSSM_KEY_PTR Key;
        CSSM_DATA_PTR Data;
        CSSM_PADDING Padding;
        CSSM_DATE_PTR Date;
        CSSM_RANGE_PTR Range;
        CSSM_CRYPTO_DATA_PTR CryptoData;
        CSSM_VERSION_PTR Version;
        CSSM_DL_DB_HANDLE_PTR DLDBHandle;
```

```
        CSSM_KR_PROFILE_PTR KRProfile;
    } Attribute;
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

Definition

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

Identifier	Description	Data Type
CSSM_ATTRIBUTE_NONE	No attribute	None
CSSM_ATTRIBUTE_CUSTOM	Custom data	Void pointer
CSSM_ATTRIBUTE_DESCRIPTION	Description of attribute	Null-terminated 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	uint32
CSSM_ATTRIBUTE_RANDOM	Random data	CSSM_DATA
CSSM_ATTRIBUTE_SEED	Seed	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_PASSPHRASE	Passphrase data	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_DATA_ACCESS_ CREDENTIALS	One or more credentials and samples required as input to use a private key or a secret key	CSSM_ACCESS_CREDENTIALS
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	Effective key size (in bits)	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_CSP_HANDLE	CSP Handle	Uint32
CSSM_ATTRIBUTE_DL_DB_ HANDLE	DL DB Handle	
CSSM_ATTRIBUTE_ALG_ID	Algorithm identifier	uint32
CSSM_ATTRIBUTE_ITERATION_ COUNT	Algorithm iterations	uint32
CSSM_ATTRIBUTE_ROUNDS_RANGE		
T}	Range of number of rounds possible	CSSM_RANGE
CSSM_KR_PROFILE_PTR	Pointer to the key recovery profile structure that defines the user parameters with respect to the key recovery process. See Section 25.3.3 on page 661.	Pointer

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

7.4.24 CSSM_CONTEXT

```
typedef struct cssm_context {
    CSSM_CONTEXT_TYPE ContextType;
    CSSM_ALGORITHMS AlgorithmType;
    uint32 NumberOfAttributes;
    CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes;
    CSSM_CSP_HANDLE CSPHandle;
    CSSM_BOOL Privileged;
    CSSM_KR_POLICY_FLAGS EncryptionProhibited;
    uint32 WorkFactor;
    uint32 Reserved;
} 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_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 indicating the cryptographic algorithm to be used.

Identifier	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_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_EDE	Triple-DES with 3 keys applied encrypt, decrypt, encrypt (EDE).
CSSM_ALGID_3DES_2KEY_EDE	Triple-DES with 2 keys applied encrypt, decrypt, encrypt (EDE), with the first key used for the first and last operation.
CSSM_ALGID_3DES_1KEY_EEE	Triple-DES with 1 keys applied encrypt, encrypt, encrypt (EEE), with the first key used for all operations.
CSSM_ALGID_3DES_3KEY	Triple-DES with 3 keys. Alias for CSSM_ALGID_3DES_3KEY_EDE.
CSSM_ALGID_3DES_3KEY_EEE	Triple-DES with 3 keys applied encrypt, encrypt, encrypt (EEE).

CSSM_ALGID_3DES_2KEY	Triple-DES with 2 keys. Alias for CSSM_ALGID_3DES_2KEY_EDE.
CSSM_ALGID_3DES_2KEY_EEE	Triple-DES with 2 keys applied encrypt, encrypt, encrypt (EEE), with the first key used for the first and last operation.
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_RSA	RSA public key cipher
CSSM_ALGID_DSA	Digital Signature Algorithm
CSSM_ALGID_MD5WithRSA	MD5/RSA signature algorithm
CSSM_ALGID_MD2WithRSA	MD2/RSA signature algorithm
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_SHA1WithRSA	SHA-1/RSA signature algorithm
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_PKCS5_PBKDF1_MD5	PKCS #5 key derivation using the PBKDF1 algorithm with MD5.
CSSM_ALGID_PKCS5_PBKDF1_MD2	PKCS #5 key derivation using the PBKDF1 algorithm with MD2.
CSSM_ALGID_PKCS5_PBKDF1_SHA1	PKCS #5 key derivation using the PBKDF1 algorithm with SHA-1.
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_ECDH	Elliptic Curve DiffieHellman Key Exchange algorithm
CSSM_ALGID_ECMQV	Elliptic Curve MQV key exchange algorithm
CSSM_ALGID_PKCS12_SHA1_PBE	PKCS12 SHA-1 PBE key derivation algorithm
CSSM_ALGID_ECNRA	Elliptic Curve Nyberg-Rueppel
CSSM_ALGID_SHA1WithECNRA	SHA-1 with Elliptic Curve Nyberg-Rueppel
CSSM_ALGID_ECES	Elliptic Curve Encryption Scheme
CSSM_ALGID_ECAES	Elliptic Curve Authenticate Encryption Scheme
CSSM_ALGID_SHA1HMAC	SHA1-MAC
CSSM_ALGID_FIPS186Random	FIPS186Random
CSSM_ALGID_ECC	ECC
CSSM_ALGID_MQV	Discrete-Log MQV key exchange algorithm@
CSSM_ALGID_NRA	Discrete-Log Nyberg-Rueppel Signature scheme
CSSM_ALGID_IntelPlatformRandom	Random data obtained by querying the Intel Platform Random Number Generator
CSSM_ALGID_UTC	Time value in the form YYYYMMDDhhmmss
CSSM_ALGID_HAVAL3	HAVAL3 Digest
CSSM_ALGID_HAVAL4	HAVAL4 Digest

CSSM_ALGID_HAVAL5	HAVAL5 Digest
CSSM_ALGID_TIGER	TIGER Digest
CSSM_ALGID_MD5HMAC	HMAC-MD5
CSSM_ALGID_PKCS5_PBKDF2	PKCS #5 key derivation using the PBKDF2 algorithm.
CSSM_ALGID_RUNNING_COUNTER	Value of a running hardware counter that operates while the device is in operation.
CSSM_ALGID_VENDOR_DEFINED	All algorithms IDs that are vendor specific, and not part of the CSSM specification should be defined relative to CSSM_ALGID_VENDOR_DEFINED.

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:

Identifier	**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@T{ CBC with padding and Initialization Vector of 8 bytes
CSSM_ALGMODE_CFB	Cipher FeedBack This mode should be used only for sizes that are not covered by CSSM_ALGMODE_CFB8, CSSM_ALGMODE_CFB16, CSSM_ALGMODE_CFB32, or CSSM_ALGMODE_CFB64. The arbitrary size is specified using the context attribute CSSM_ATTRIBUTE_OUTPUT_SIZE.
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
CSSM_ALGMODE_PKCS1_EME_OAEP	PKCS #1 version 2.0, requires that CSSM_PKCS1_OAEP_PARAMS be included as a context attribute of type CSSM_ATTRIBUTE_ALG_PARAMS
CSSM_ALGMODE_PKCS1_EME_V15	PKCS #1 version 1.5: this is the default if no algorithm mode is specified.

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.

CSPHandle
The handle of the CSP associated with this context.

Privileged
When this flag is CSSM_TRUE, the context can perform crypto operations without being forced to follow the key recovery policy.

EncryptionProhibited
If 0, then encryption is allowed. Otherwise, the flag indicates which policy disallowed encryption (see Section 25.3.5 on page 663).

WorkFactor
WorkFactor is the maximum number of key bits that can be left out of key recovery fields when they are generated.

Reserved
An unsigned integer field reserved for future use.

7.4.25 CSSM_SC_FLAGS

A bit mask containing information about a subservice capabilities. This type is used to interpret the *ScFlags* fields of the "CSP SmartcardInfo Relation" in the MDS.

```
typedef uint32 CSSM_SC_FLAGS;

#define CSSM_CSP_TOK_RNG (0x00000001)
#define CSSM_CSP_TOK_CLOCK_EXISTS (0x00000040)
```

Definition

CSSM_CSP_TOK_RNG
Subservice has a hardware RNG

CSSM_CSP_TOK_CLOCK_EXISTS
Subservice has a hardware clock

7.4.26 CSSM_CSP_READER_FLAGS

A bit mask containing information about the state of a reader.

```
typedef uint32 CSSM_CSP_READER_FLAGS;

#define CSSM_CSP_RDR_TOKENPRESENT (0x00000001)
                                   /* Token is present in reader/slot */
#define CSSM_CSP_RDR_EXISTS       (0x00000002)
                         /* Device is a reader with a removable token */
#define CSSM_CSP_RDR_HW           (0x00000004)
                                           /* Slot is a hardware slot */
```

7.4.27 CSSM_CSP_FLAGS

A bit mask containing information about the state of a service provider.

```
typedef uint32 CSSM_CSP_FLAGS;

#define CSSM_CSP_TOK_WRITE_PROTECTED       (0x00000002)
#define CSSM_CSP_TOK_LOGIN_REQUIRED        (0x00000004)
#define CSSM_CSP_TOK_USER_PIN_INITIALIZED  (0x00000008)
#define CSSM_CSP_TOK_PROT_AUTHENTICATION   (0x00000100)

#define CSSM_CSP_TOK_USER_PIN_EXPIRED      (0x00100000)
#define CSSM_CSP_TOK_SESSION_KEY_PASSWORD  (0x00200000)
#define CSSM_CSP_TOK_PRIVATE_KEY_PASSWORD  (0x00400000)
#define CSSM_CSP_STORES_PRIVATE_KEYS       (0x01000000)
#define CSSM_CSP_STORES_PUBLIC_KEYS        (0x02000000)
#define CSSM_CSP_STORES_SESSION_KEYS       (0x04000000)
#define CSSM_CSP_STORES_CERTIFICATES       (0x08000000)
#define CSSM_CSP_STORES_GENERIC            (0x10000000)
```

Description

CSSM_CSP_TOK_WRITE_PROTECTED
Service provider 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_PROT_AUTHENTICATION
Service provider has protected authentication path for entering a user PIN. No password should be supplied to the CSSM_CSP_Login API.

CSSM_CSP_TOK_USER_PIN_EXPIRED
The user PIN must be changed before the service provider can be used.

CSSM_CSP_TOK_SESSION_KEY_PASSWORD
Session keys held by the CSP require individual passwords, possibly in addition to a login password.

CSSM_CSP_TOK_PRIVATE_KEY_PASSWORD
Private keys held by the CSP require individual passwords, possibly in addition to a login password

CSSM_CSP_STORES_PRIVATE_KEYS
CSP can store private keys.

CSSM_CSP_STORES_PUBLIC_KEYS
CSP can store public keys.

CSSM_CSP_STORES_SESSION_KEYS
CSP can store session/secret keys

CSSM_CSP_STORES_CERTIFICATES
Service provider can store certs using DL APIs.

CSSM_CSP_STORES_GENERIC
Service provider can store generic objects using DL APIs.

7.4.28 CSSM_PKCS_OAEP

```
typedef uint32 CSSM_PKCS_OAEP_MGF;

typedef uint32 CSSM_PKCS_OAEP_PSOURCE;

#define CSSM_PKCS_OAEP_MGF_NONE   (0)
#define CSSM_PKCS_OAEP_MGF1_SHA1 (CSSM_PKCS_OAEP_MGF_NONE+1)

#define CSSM_PKCS_OAEP_MGF1_MD5 (CSSM_PKCS_OAEP_MGF_NONE+2)

#define CSSM_PKCS_OAEP_PSOURCE_NONE (0)
#define CSSM_PKCS_OAEP_PSOURCE_Pspecified (CSSM_PKCS_OAEP_PSOURCE_NONE+1)
```

7.4.29 CSSM_PKCS_OAEP_PARAMS

When using PKCS #1 RSA with Optimal Asymmetric Encryption Padding (OAEP) encoding, this structure must be added to the asymmetric cryptography context for that operation. The parameter is added as attribute type CSSM_ATTRIBUTE_ALG_PARAMS along with an attribute of type CSSM_ATTRIBUTE_MODE, which specifies OAEP.

```
typedef struct cssm_pkcs1_oaep_params {
    uint32 HashAlgorithm;
    CSSM_DATA HashParams;
    CSSM_PKCS_OAEP_MGF MGF;
    CSSM_DATA MGFParams;
    CSSM_PKCS_OAEP_PSOURCE PSource;
    CSSM_DATA PSourceParams;
} CSSM_PKCS1_OAEP_PARAMS, *CSSM_PKCS1_OAEP_PARAMS_PTR;
```

Definition

HashAlgorithm
CSSM algorithm identifier specifying the hash algorithm used during the OAEP encoding.

HashParams
Extra parameters required by a hashing algorithm. For most hash functions this parameter will be empty.

MGF
One of the Mask Generation Function (MGF) identifiers defined in PKCS #1. These values match the functions specified in PKCS #1, but custom MGF functions may be used if supported by the CSP.

MGFParams
Parameter data required by the MGF. The MGF (MGF1 w/SHA-1) function specified in PKCS #1 does not require a parameter.

Psource
One of the source identifiers defined in PKCS #1 for extra data encrypted with the data block. The "specified" mode is the only mode currently made available by PKCS #1.

PsourceParams
Varies depending on the source specified in *Psource*.

7.4.30 CSSM_CSP_OPERATIONAL_STATISTICS

```
typedef struct cssm_csp_operational_statistics
{
    CSSM_BOOL UserAuthenticated;
    /* CSSM_TRUE if the user is logged in to the token,
       CSSM_FALSE otherwise. */
    CSSM_CSP_FLAGS DeviceFlags;
    uint32 TokenMaxSessionCount; /* Exported by Cryptoki modules. */
    uint32 TokenOpenedSessionCount;
    uint32 TokenMaxRWSessionCount;
    uint32 TokenOpenedRWSessionCount;
    uint32 TokenTotalPublicMem; /* Storage space statistics. */
    uint32 TokenFreePublicMem;
    uint32 TokenTotalPrivateMem;
```

```
    uint32 TokenFreePrivateMem;
} CSSM_CSP_OPERATIONAL_STATISTICS *CSSM_CSP_OPERATIONAL_STATISTICS_PTR;

#define CSSM_VALUE_NOT_AVAILABLE ((uint32)(~0))
```

Indicates that the statistical value can not be revealed or is not relevant for a CSP

Definition

UserAuthenticated
CSSM_TRUE if the user is logged in to the token, CSSM_FALSE otherwise

DeviceFlags
Device status flags.

TokenMaxSessionCount
Maximum number of CSP handles referencing the token that may exist simultaneously.

TokenOpenedSessionCount
Number of existing CSP handles referencing the token.

TokenMaxRWSessionCount
Maximum number of CSP handles that can reference the token simultaneously in read-write mode.

TokenOpenedRWSessionCount
Number of existing CSP handles referencing the token in read-write mode.

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.

7.4.31 CSSM_PKCS5_PBKDF1_PARAMS

This structure is used to provide input parameters to key derivation algorithms based on password based encryption.

```
typedef struct cssm_pkcs5_pbkdf1_params {
    CSSM_DATA Passphrase;
    CSSM_DATA InitVector;
} CSSM_PKCS5_PBKDF1_PARAMS, *CSSM_PKCS5_PBKDF1_PARAMS_PTR;
```

Definition

Passphrase
The passphrase used as the basis for key derivation.

InitVector
The initialization vector returned as an additional result of the key derivation procedure. If the returned derived key is to be used for CBC mode encryption, *InitVector* should be used as the initialization vector for the encryption function.

7.4.32 CSSM_PKCS5_PBKDF2_PRF

This type indicates the underlying pseudo-random function (PRF) used by the PKCS #5 v2.0 PBKDF2 key derivation function, CSSM_ALGID_PKCS5_PBKDF2.

```
typedef uint32 CSSM_PKCS5_PBKDF2_PRF;

#define CSSM_PKCS5_PBKDF2_PRF_HMAC_SHA1 (0)
```

7.4.33 CSSM_PKCS5_PBKDF2_PARAMS

This structure is used to provide input parameters to key derivation algorithms based on PKCS #5 v2.0 password based encryption.

```
typedef struct cssm_pkcs5_pbkdf2_params {
    CSSM_DATA Passphrase;
    CSSM_PKCS5_PBKDF2_PRF PseudoRandomFunction;
} CSSM_PKCS5_PBKDF2_PARAMS; *CSSM_PKCS5_PBKDF2_PARAMS_PTR;
```

Definition

Passphrase
The passphrase used as the basis for key derivation.

PseudoRandomFunction
Pseudo-random function to use for the key derivation process.

7.4.34 CSSM_KEA_DERIVE_PARAMS

This structure is used during phase 2 of the Key Exchange Algorithm (KEA).

```
typedef struct cssm_kea_derive_params {
    CSSM_DATA Rb;
    CSSM_DATA Yb;
} CSSM_KEA_DERIVE_PARAMS, *CSSM_KEA_DERIVE_PARAMS_PTR;
```

Definition

Rb References a buffer containing the Ra value received from the remote party.

Yb References a buffer containing the public value from the remote party.

7.5 Error Codes and Error Values

This section defines Error Values that can be returned by CSP operations.

Each CSP function may return any Error Value derived from the Common Error Codes defined in Appendix A on page 925, if it satisfies the conditions defined for that Error Code. In addition, a number of common sets of Error Values are defined specifically for CSP functions:

1. A general set that can be returned by any CSP function
2. A set that can be returned by key operations
3. A set that can be returned by operations accepting vectors of buffers
4. A set that can be returned by operations that take a cryptographic context handle
5. A set that can be returned by staged operations

Lastly, there is an unclassified set that is specific to certain operations.

Each CSP function will only list the Error Values from the unclassified set that it returns, plus certain CSSM Error Values that relate to invalid contexts.

7.5.1 CSP Error Values Derived from Common Error Codes

See Appendix A on page 925.

Common Error Values for All Module Types

```
#define CSSMERR_CSP_INTERNAL_ERROR \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)
#define CSSMERR_CSP_MEMORY_ERROR \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)
#define CSSMERR_CSP_MDS_ERROR \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)
#define CSSMERR_CSP_INVALID_POINTER \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)
#define CSSMERR_CSP_INVALID_INPUT_POINTER \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)
#define CSSMERR_CSP_INVALID_OUTPUT_POINTER \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)
#define CSSMERR_CSP_FUNCTION_NOT_IMPLEMENTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)
#define CSSMERR_CSP_SELF_CHECK_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)
#define CSSMERR_CSP_OS_ACCESS_DENIED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)
#define CSSMERR_CSP_FUNCTION_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)
```

Common ACL Error Values

```
#define CSSMERR_CSP_OPERATION_AUTH_DENIED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OPERATION_AUTH_DENIED)
#define CSSMERR_CSP_OBJECT_USE_AUTH_DENIED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OBJECT_USE_AUTH_DENIED)
#define CSSMERR_CSP_OBJECT_MANIP_AUTH_DENIED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OBJECT_MANIP_AUTH_DENIED)
#define CSSMERR_CSP_OBJECT_ACL_NOT_SUPPORTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OBJECT_ACL_NOT_SUPPORTED)
#define CSSMERR_CSP_OBJECT_ACL_REQUIRED \
```

```
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_OBJECT_ACL_REQUIRED)
#define CSSMERR_CSP_INVALID_ACCESS_CREDENTIALS \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACCESS_CREDENTIALS)
#define CSSMERR_CSP_INVALID_ACL_BASE_CERTS \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_BASE_CERTS)
#define CSSMERR_CSP_ACL_BASE_CERTS_NOT_SUPPORTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_BASE_CERTS_NOT_SUPPORTED)
#define CSSMERR_CSP_INVALID_SAMPLE_VALUE \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_SAMPLE_VALUE)
#define CSSMERR_CSP_SAMPLE_VALUE_NOT_SUPPORTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_SAMPLE_VALUE_NOT_SUPPORTED)
#define CSSMERR_CSP_INVALID_ACL_SUBJECT_VALUE \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_SUBJECT_VALUE)
#define CSSMERR_CSP_ACL_SUBJECT_TYPE_NOT_SUPPORTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_SUBJECT_TYPE_NOT_SUPPORTED)
#define CSSMERR_CSP_INVALID_ACL_CHALLENGE_CALLBACK \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_CHALLENGE_CALLBACK)
#define CSSMERR_CSP_ACL_CHALLENGE_CALLBACK_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_CHALLENGE_CALLBACK_FAILED)
#define CSSMERR_CSP_INVALID_ACL_ENTRY_TAG \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_ENTRY_TAG)
#define CSSMERR_CSP_ACL_ENTRY_TAG_NOT_FOUND \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_ENTRY_TAG_NOT_FOUND)
#define CSSMERR_CSP_INVALID_ACL_EDIT_MODE \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_EDIT_MODE)
#define CSSMERR_CSP_ACL_CHANGE_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_CHANGE_FAILED)
#define CSSMERR_CSP_INVALID_NEW_ACL_ENTRY \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_NEW_ACL_ENTRY)
#define CSSMERR_CSP_INVALID_NEW_ACL_OWNER \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_NEW_ACL_OWNER)
#define CSSMERR_CSP_ACL_DELETE_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_DELETE_FAILED)
#define CSSMERR_CSP_ACL_REPLACE_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_REPLACE_FAILED)
#define CSSMERR_CSP_ACL_ADD_FAILED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_ACL_ADD_FAILED)
```

Common Error Values for Specific Data Types

```
#define CSSMERR_CSP_INVALID_CONTEXT_HANDLE \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_CONTEXT_HANDLE)
#define CSSMERR_CSP_PRIVILEGE_NOT_GRANTED \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_PRIVILEGE_NOT_GRANTED)
#define CSSMERR_CSP_INVALID_DATA \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_DATA)
#define CSSMERR_CSP_INVALID_PASSTHROUGH_ID \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_PASSTHROUGH_ID)
#define CSSMERR_CSP_INVALID_CRYPTO_DATA \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRCODE_INVALID_CRYPTO_DATA)
```

7.5.2 General CSP Error Values

These error values can be returned from any CSP function.

```
#define CSSM_CSP_BASE_CSP_ERROR \
    (CSSM_CSP_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT)
```

```
#define CSSMERR_CSP_INPUT_LENGTH_ERROR (CSSM_CSP_BASE_CSP_ERROR+1)
```

An input buffer does not have the expected length

```
#define CSSMERR_CSP_OUTPUT_LENGTH_ERROR (CSSM_CSP_BASE_CSP_ERROR+2)
```

An output buffer was supplied, but it was too small to hold the output data

```
#define CSSMERR_CSP_PRIVILEGE_NOT_SUPPORTED (CSSM_CSP_BASE_CSP_ERROR+3)
```

The CSP does not support the requested privilege level

```
#define CSSMERR_CSP_DEVICE_ERROR (CSSM_CSP_BASE_CSP_ERROR+4)
```

General device error; Indicates that a hardware subsystem has failed in some way

```
#define CSSMERR_CSP_DEVICE_MEMORY_ERROR (CSSM_CSP_BASE_CSP_ERROR+5)
```

General device error; Indicates that a hardware subsystem has run out of memory

```
#define CSSMERR_CSP_ATTACH_HANDLE_BUSY (CSSM_CSP_BASE_CSP_ERROR+6)
```

The attach handle used to attempt an operation currently has an operation in progress that will not allow other operations to begin until it completes

```
#define CSSMERR_CSP_NOT_LOGGED_IN (CSSM_CSP_BASE_CSP_ERROR+7)
```

The operation can not be performed without authenticating using CSSM_CSP_Login

7.5.3 CSP Key Error Values

These error values can be returned from CSP functions that use a key. The key may be passed directly into the function or specified as an attribute through the cryptographic context.

```
#define CSSMERR_CSP_INVALID_KEY (CSSM_CSP_BASE_CSP_ERROR+16)
```

The supplied key is invalid or incompatible with the operation

```
#define CSSMERR_CSP_INVALID_KEY_REFERENCE (CSSM_CSP_BASE_CSP_ERROR+17)
```

The CSSM_KEY contains a reference that does not indicate a key in the CSP

```
#define CSSMERR_CSP_INVALID_KEY_CLASS (CSSM_CSP_BASE_CSP_ERROR+18)
```

The supplied key is not the proper class (i.e. Public key is supplied for a private key operation)

```
#define CSSMERR_CSP_ALGID_MISMATCH (CSSM_CSP_BASE_CSP_ERROR+19)
```

The algorithm ID in the key header does not match the algorithm to be performed

```
#define CSSMERR_CSP_KEY_USAGE_INCORRECT (CSSM_CSP_BASE_CSP_ERROR+20)
```

The key does not have the proper usage flags to perform the operation

```
#define CSSMERR_CSP_KEY_BLOB_TYPE_INCORRECT (CSSM_CSP_BASE_CSP_ERROR+21)
```

The key data blob type is not the correct type (i.e. The key is wrapped when a raw key or reference is expected)

```
#define CSSMERR_CSP_KEY_HEADER_INCONSISTENT (CSSM_CSP_BASE_CSP_ERROR+22)
```

The key header information is corrupt, or does not match the key data

```
#define CSSMERR_CSP_UNSUPPORTED_KEY_FORMAT (CSSM_CSP_BASE_CSP_ERROR+23)
```

The key format is not supported by the CSP

```
#define CSSMERR_CSP_UNSUPPORTED_KEY_SIZE (CSSM_CSP_BASE_CSP_ERROR+24)
```

The key size is not supported or not allowed by the current privilege or supported by the CSP

```
#define CSSMERR_CSP_INVALID_KEY_POINTER (CSSM_CSP_BASE_CSP_ERROR+25)
```

The pointer to a CSSM_KEY structure is invalid

```
#define CSSMERR_CSP_INVALID_KEYUSAGE_MASK (CSSM_CSP_BASE_CSP_ERROR+26)
```

A requested key usage is not valid for the key type, or two of the uses are not compatible

```
#define CSSMERR_CSP_UNSUPPORTED_KEYUSAGE_MASK (CSSM_CSP_BASE_CSP_ERROR+27)
```

The key usage mask is valid, but not supported by the CSP

```
#define CSSMERR_CSP_INVALID_KEYATTR_MASK (CSSM_CSP_BASE_CSP_ERROR+28)
```

A requested key attribute is not valid for the key type, or two of the attributes are not compatible

```
#define CSSMERR_CSP_UNSUPPORTED_KEYATTR_MASK (CSSM_CSP_BASE_CSP_ERROR+29)
```

The key attribute mask is valid, but not supported by the CSP

```
#define CSSMERR_CSP_INVALID_KEY_LABEL (CSSM_CSP_BASE_CSP_ERROR+30)
```

The label specified for a key is invalid

```
#define CSSMERR_CSP_UNSUPPORTED_KEY_LABEL (CSSM_CSP_BASE_CSP_ERROR+31)
```

The CSP does not support the use of labels for the key.

```
#define CSSMERR_CSP_INVALID_KEY_FORMAT (CSSM_CSP_BASE_CSP_ERROR+32)
```

Invalid key format

7.5.4 CSP Vector of Buffers Error Values

These error values can be returned by APIs that accept a vector of buffers as input or output.

```
#define CSSMERR_CSP_INVALID_DATA_COUNT (CSSM_CSP_BASE_CSP_ERROR+40)
```

Input vector length is invalid; buffer count can not be zero.

```
#define CSSMERR_CSP_VECTOR_OF_BUFS_UNSUPPORTED (CSSM_CSP_BASE_CSP_ERROR+41)
```

The CSP only supports input of a single buffer per API call

```
#define CSSMERR_CSP_INVALID_INPUT_VECTOR (CSSM_CSP_BASE_CSP_ERROR+42)
```

A vector of buffers for input does not contain valid information

```
#define CSSMERR_CSP_INVALID_OUTPUT_VECTOR (CSSM_CSP_BASE_CSP_ERROR+43)
```

A vector of buffers for output does not contain valid information

7.5.5 CSP Cryptographic Context Error Values

These error values can be returned by CSP APIs that take A cryptographic context handle as input.

```
#define CSSMERR_CSP_INVALID_CONTEXT (CSSM_CSP_BASE_CSP_ERROR+48)
```

The cryptographic context and operation types are not compatible

```
#define CSSMERR_CSP_INVALID_ALGORITHM (CSSM_CSP_BASE_CSP_ERROR+49)
```

Algorithm is not supported by the CSP

```
#define CSSMERR_CSP_INVALID_ATTR_KEY  (CSSM_CSP_BASE_CSP_ERROR + 54)
#define CSSMERR_CSP_MISSING_ATTR_KEY  (CSSM_CSP_BASE_CSP_ERROR + 55)
```

The cryptographic key is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_INIT_VECTOR  (CSSM_CSP_BASE_CSP_ERROR + 56)
#define CSSMERR_CSP_MISSING_ATTR_INIT_VECTOR  (CSSM_CSP_BASE_CSP_ERROR + 57)
```

The algorithm mode required an initialization vector and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_SALT  (CSSM_CSP_BASE_CSP_ERROR + 58)
#define CSSMERR_CSP_MISSING_ATTR_SALT  (CSSM_CSP_BASE_CSP_ERROR + 59)
```

The operation requires salt and it is is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_PADDING  (CSSM_CSP_BASE_CSP_ERROR + 60)
#define CSSMERR_CSP_MISSING_ATTR_PADDING  (CSSM_CSP_BASE_CSP_ERROR + 61)
```

An algorithm mode with padding is specified and the padding type is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_RANDOM  (CSSM_CSP_BASE_CSP_ERROR + 62)
#define CSSMERR_CSP_MISSING_ATTR_RANDOM  (CSSM_CSP_BASE_CSP_ERROR + 63)
```

The operation required random data and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_SEED  (CSSM_CSP_BASE_CSP_ERROR + 64)
#define CSSMERR_CSP_MISSING_ATTR_SEED  (CSSM_CSP_BASE_CSP_ERROR + 65)
```

The algorithm requires a seed value and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_PASSPHRASE  (CSSM_CSP_BASE_CSP_ERROR + 66)
#define CSSMERR_CSP_MISSING_ATTR_PASSPHRASE  (CSSM_CSP_BASE_CSP_ERROR + 67)
```

The operation requires a passphrase value and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_KEY_LENGTH  (CSSM_CSP_BASE_CSP_ERROR + 68)
#define CSSMERR_CSP_MISSING_ATTR_KEY_LENGTH  (CSSM_CSP_BASE_CSP_ERROR + 69)
```

The operation requires a key length and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_BLOCK_SIZE  (CSSM_CSP_BASE_CSP_ERROR + 70)
#define CSSMERR_CSP_MISSING_ATTR_BLOCK_SIZE  (CSSM_CSP_BASE_CSP_ERROR + 71)
```

The algorithm has a configurable block size and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_OUTPUT_SIZE  (CSSM_CSP_BASE_CSP_ERROR + 100)
#define CSSMERR_CSP_MISSING_ATTR_OUTPUT_SIZE  (CSSM_CSP_BASE_CSP_ERROR + 101)
```

The algorithm has a configurable output size and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_ROUNDS  (CSSM_CSP_BASE_CSP_ERROR + 102)
#define CSSMERR_CSP_MISSING_ATTR_ROUNDS  (CSSM_CSP_BASE_CSP_ERROR + 103)
```

The algorithm has a configurable number of rounds and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_ALG_PARAMS  (CSSM_CSP_BASE_CSP_ERROR + 104)
#define CSSMERR_CSP_MISSING_ATTR_ALG_PARAMS  (CSSM_CSP_BASE_CSP_ERROR + 105)
```

The algorithm required a set of parameters and they are missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_LABEL  (CSSM_CSP_BASE_CSP_ERROR + 106)
#define CSSMERR_CSP_MISSING_ATTR_LABEL  (CSSM_CSP_BASE_CSP_ERROR + 107)
```

The operation creates an object that requires a label and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_KEY_TYPE  (CSSM_CSP_BASE_CSP_ERROR + 108)
#define CSSMERR_CSP_MISSING_ATTR_KEY_TYPE  (CSSM_CSP_BASE_CSP_ERROR + 109)
```

The operation requires a key type and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_MODE  (CSSM_CSP_BASE_CSP_ERROR + 110)
#define CSSMERR_CSP_MISSING_ATTR_MODE  (CSSM_CSP_BASE_CSP_ERROR + 111)
```

The algorithm requires a mode to be specified and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_EFFECTIVE_BITS
    (CSSM_CSP_BASE_CSP_ERROR + 112)
#define CSSMERR_CSP_MISSING_ATTR_EFFECTIVE_BITS
    (CSSM_CSP_BASE_CSP_ERROR + 113)
```

The algorithm has a configurable number of effective bits and the value it missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_START_DATE  (CSSM_CSP_BASE_CSP_ERROR + 114)
#define CSSMERR_CSP_MISSING_ATTR_START_DATE  (CSSM_CSP_BASE_CSP_ERROR + 115)
```

The operation creates an object with a validity date and the starting date is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_END_DATE  (CSSM_CSP_BASE_CSP_ERROR + 116)
#define CSSMERR_CSP_MISSING_ATTR_END_DATE  (CSSM_CSP_BASE_CSP_ERROR + 117)
```

The operation creates an object with a validity date and the ending date is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_VERSION  (CSSM_CSP_BASE_CSP_ERROR + 118)
#define CSSMERR_CSP_MISSING_ATTR_VERSION  (CSSM_CSP_BASE_CSP_ERROR + 119)
```

The operation requires a version number and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_PRIME  (CSSM_CSP_BASE_CSP_ERROR + 120)
#define CSSMERR_CSP_MISSING_ATTR_PRIME  (CSSM_CSP_BASE_CSP_ERROR + 121)
```

The operation requires a prime value and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_BASE  (CSSM_CSP_BASE_CSP_ERROR + 122)
#define CSSMERR_CSP_MISSING_ATTR_BASE  (CSSM_CSP_BASE_CSP_ERROR + 123)
```

The operation requires a base value and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_SUBPRIME  (CSSM_CSP_BASE_CSP_ERROR + 124)
#define CSSMERR_CSP_MISSING_ATTR_SUBPRIME  (CSSM_CSP_BASE_CSP_ERROR + 125)
```

The operation requires a sub-prime value and it is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_ITERATION_COUNT
    (CSSM_CSP_BASE_CSP_ERROR + 126)
#define CSSMERR_CSP_MISSING_ATTR_ITERATION_COUNT
    (CSSM_CSP_BASE_CSP_ERROR + 127)
```

The operation has a configurable iteration count and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_DL_DB_HANDLE
    (CSSM_CSP_BASE_CSP_ERROR + 128)
#define CSSMERR_CSP_MISSING_ATTR_DL_DB_HANDLE
    (CSSM_CSP_BASE_CSP_ERROR + 129)
```

The operation can store new objects in a specific location and the value is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_ACCESS_CREDENTIALS
    (CSSM_CSP_BASE_CSP_ERROR + 130)
#define CSSMERR_CSP_MISSING_ATTR_ACCESS_CREDENTIALS
    (CSSM_CSP_BASE_CSP_ERROR + 131)
```

The operation requires access credentials and they are missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_PUBLIC_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 132)
#define CSSMERR_CSP_MISSING_ATTR_PUBLIC_KEY_FORMAT
```

```
    (CSSM_CSP_BASE_CSP_ERROR + 133)
```

The resulting public key format is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_PRIVATE_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 134)
#define CSSMERR_CSP_MISSING_ATTR_PRIVATE_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 135)
```

The resulting private key format is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_SYMMETRIC_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 136)
#define CSSMERR_CSP_MISSING_ATTR_SYMMETRIC_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 137)
```

The resulting symmetric key format is missing or invalid.

```
#define CSSMERR_CSP_INVALID_ATTR_WRAPPED_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 138)
#define CSSMERR_CSP_MISSING_ATTR_WRAPPED_KEY_FORMAT
    (CSSM_CSP_BASE_CSP_ERROR + 139)
```

The resulting wrapped key format is missing or invalid.

7.5.6 CSP Staged Cryptographic API Error Values

These error values can be returned by staged cryptographic APIs. The names of the staged cryptographic APIs end in "Init", "Update", "Final" and "InitP".

```
#define CSSMERR_CSP_STAGED_OPERATION_IN_PROGRESS \
    (CSSM_CSP_BASE_CSP_ERROR+72)
```

The application has already started a staged operation using the specified CC

```
#define CSSMERR_CSP_STAGED_OPERATION_NOT_STARTED \
    (CSSM_CSP_BASE_CSP_ERROR+73)
```

An "Update" or "Final" API has been called without calling the corresponding "Init"

7.5.7 Other CSP Error Values

```
#define CSSMERR_CSP_VERIFY_FAILED (CSSM_CSP_BASE_CSP_ERROR+74)
```

The signature is not valid

```
#define CSSMERR_CSP_INVALID_SIGNATURE (CSSM_CSP_BASE_CSP_ERROR+75)
```

The signature data is not in the proper format

```
#define CSSMERR_CSP_QUERY_SIZE_UNKNOWN (CSSM_CSP_BASE_CSP_ERROR+76)
```

The size of the output data can not be determined

```
#define CSSMERR_CSP_BLOCK_SIZE_MISMATCH (CSSM_CSP_BASE_CSP_ERROR+77)
```

The size of the input is not equal to a multiple of the algorithm block size; valid for symmetric block ciphers in an unpadded mode

```
#define CSSMERR_CSP_PRIVATE_KEY_NOT_FOUND (CSSM_CSP_BASE_CSP_ERROR+78)
```

The private key matching the public key was not found

```
#define CSSMERR_CSP_PUBLIC_KEY_INCONSISTENT (CSSM_CSP_BASE_CSP_ERROR+79)
```

The public key specified does not match the private key being unwrapped

```
#define CSSMERR_CSP_DEVICE_VERIFY_FAILED (CSSM_CSP_BASE_CSP_ERROR+80)
```

The logical device could not be verified by the service provider

```
#define CSSMERR_CSP_INVALID_LOGIN_NAME (CSSM_CSP_BASE_CSP_ERROR+81)
```

The login name is not recognized by the CSP

```
#define CSSMERR_CSP_ALREADY_LOGGED_IN (CSSM_CSP_BASE_CSP_ERROR+82)
```

The device is already logged in and can not be reauthenticated

```
#define CSSMERR_CSP_PRIVATE_KEY_ALREADY_EXISTS \
    (CSSM_CSP_BASE_CSP_ERROR+83)
```

The private key already exists in the CSP.

```
#define CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS (CSSM_CSP_BASE_CSP_ERROR+84)
```

Key label already exists in the CSP.

```
#define CSSMERR_CSP_INVALID_DIGEST_ALGORITHM (CSSM_CSP_BASE_CSP_ERROR+85)
```

The digest algorithm passed in to the Sign/Verify operation is invalid.

```
#define CSSMERR_CSP_CRYPTO_DATA_CALLBACK_FAILED
    (CSSM_CSP_BASE_CSP_ERROR+86)
```

The crypto data callback failed

7.6 Cryptographic Context Operations

The man-page definitions for Cryptographic Context operations are presented in this section.

NAME

CSSM_CSP_CreateSignatureContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_CreateSignatureContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    CSSM_CC_HANDLE *NewContextHandle)
```

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 an *AccessCredentials* structure. The *AccessCredentials* 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.

AccessCred (input/optional)
: A pointer to the set of one or more credentials required to unlock the private key. The credentials structure can contain an immediate value for the credential, such as a passphrase, or the caller can specify a callback function the CSP can use to obtain one or more credentials. Credentials are required for signature operations, not for verify operations.

Key (input)
: The key used to sign/verify. The caller passes in a pointer to a CSSM_KEY structure containing the key and the key length.

NewContextHandle (output)
: Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_CSP_CreateSymmetricContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    CSSM_ENCRYPT_MODE Mode,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    const CSSM_DATA *InitVector,
    CSSM_PADDING Padding,
    void *Reserved,
    CSSM_CC_HANDLE *NewContextHandle)
```

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. Algorithm-specific attributes must be added to the context after the initial creation using the *CSSM_UpdateContextAttributes*() function. The cryptographic context handle is returned. The cryptographic context handle can be used to call symmetric encryption functions and the cryptographic wrap/unwrap functions.

Additional attributes can be added to the newly created context using the function *CSSM_UpdateContextAttributes*(). Incremental attributes of interest when using this context to unwrap a key include a handle-pair identifying a Data Storage Library service module and an open data store for CSPs that manage multiple persistent key stores. If a CSP does not support multiple key stores, the CSP ignores the presence or absence of this attribute.

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.

AccessCred (input/optional)
: A pointer to the set of one or more credentials required to unlock the secret key. The credentials structure can contain an immediate value for the credential, such as a passphrase, or the caller can specify a *callback* function the CSP can use to obtain one or more credentials. Credentials may be required for encryption, decryption, and wrapping operations.

Key (input)
: The key used for symmetric encryption. The caller passes in a pointer to a CSSM_KEY structure containing the key.

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.

Reserved (input)
Reserved for future use.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_CSP_CreateDigestContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    CSSM_CC_HANDLE *NewContextHandle)
```

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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_CSP_CreateMacContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    const CSSM_KEY *Key,
    CSSM_CC_HANDLE *NewContextHandle)
```

DESCRIPTION

This function creates a message authentication code cryptographic context, given a handle of a CSP, algorithm identification number, and a key. 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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_CSP_CreateRandomGenContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    const CSSM_CRYPTO_DATA *Seed,
    uint32 Length,
    CSSM_CC_HANDLE *NewContextHandle)
```

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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_GenerateRandom()
CSSM_GetContext()
CSSM_SetContext()
CSSM_DeleteContext()
CSSM_GetContextAttribute()
CSSM_UpdateContextAttributes()

NAME

CSSM_CSP_CreateAsymmetricContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_CreateAsymmetricContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    CSSM_PADDING Padding,
    CSSM_CC_HANDLE *NewContextHandle)
```

DESCRIPTION

This function creates an asymmetric encryption cryptographic context, given a handle of a CSP, an algorithm identification number, a key, and padding. 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.

AccessCred (input)
A pointer to the set of one or more credentials required to unlock the private key. The credentials structure can contain an immediate value for the credential, such as a passphrase, or the caller can specify a *callback* function the CSP can use to obtain one or more credentials. Credentials can be required for encryption and decryption operations.

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, *AccessCredentials* 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, *AccessCredentials* 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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_CSP_CreateDeriveKeyContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    CSSM_KEY_TYPE DeriveKeyType,
    uint32 DeriveKeyLengthInBits,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *BaseKey,
    uint32 IterationCount,
    const CSSM_DATA *Salt,
    const CSSM_CRYPTO_DATA *Seed,
    CSSM_CC_HANDLE *NewContextHandle)
```

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 *AccessCredentials* 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 logical length of the key to be derived in bits (*LogicalKeySizeInBits*)

AccessCred (input/optional)
A pointer to the set of one or more credentials required to access the base key. The credentials structure can contain an immediate value for the credential, such as a passphrase, or the caller can specify a callback function the CSP can use to obtain one or more credentials. If the *BaseKey* is NULL then this parameter is optional.

BaseKey (input/optional)
The base key used to derive the new key. The base key may be a public key, a private key, or a symmetric key

IterationCount (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 *Seed* is NULL, the cryptographic service provider

will use its default seed handling mechanism.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_DeriveKey( )

NAME

CSSM_CSP_CreateKeyGenContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_CreateKeyGenContext
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS AlgorithmID,
    uint32 KeySizeInBits,
    const CSSM_CRYPTO_DATA *Seed,
    const CSSM_DATA *Salt,
    const CSSM_DATE *StartDate,
    const CSSM_DATE *EndDate,
    const CSSM_DATA *Params,
    CSSM_CC_HANDLE *NewContextHandle)
```

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, and a salt. The cryptographic context handle is returned. The cryptographic context handle can be used to call key/keypair generation functions.

Additional attributes can be added to the newly created context using the function *CSSM_UpdateContextAttributes*(). Incremental attributes of interest for key generation include a handle-pair identifying a Data Storage Library service module and an open data store for CSPs that manage multiple persistent key stores. If a CSP does not support multiple key stores, the CSP ignores the presence or absence of this attribute.

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.

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).

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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_GenerateKey( )
CSSM_GenerateKeyPair( )
CSSM_GetContext( )
CSSM_SetContext( )
CSSM_DeleteContext( )
CSSM_GetContextAttribute( )
CSSM_UpdateContextAttributes( )

NAME

CSSM_CSP_CreatePassThroughContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_CreatePassThroughContext
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY *Key,
    CSSM_CC_HANDLE *NewContextHandle)
```

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.

NewContextHandle (output)
Cryptographic context handle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

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_RETURN CSSMAPI CSSM_GetContext
    (CSSM_CC_HANDLE CCHandle,
    CSSM_CONTEXT_PTR *Context)
```

DESCRIPTION

This function retrieves the context information when provided with a context handle.

PARAMETERS

CCHandle (input)
: The handle to the context information.

Context (output)
: The pointer to the CSSM_CONTEXT_PTR structure that describes the context associated with the handle *CCHandle*. The pointer will be set to NULL if the function fails. Use *CSSM_FreeContext*( ) to free the memory allocated by the CSSM.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_CONTEXT_HANDLE

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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_GetContext( )

NAME

CSSM_SetContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetContext
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function replaces all of the context information associated with an existing context specified by *CCHandle*. The contents of the basic context structure and all of the attributes included in that structure are replaced by the context structure and attribute values contained in the input parameter *Context*.

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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_CONTEXT_HANDLE
CSSMERR_CSSM_INVALID_ATTRIBUTE

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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_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_RETURN CSSMAPI CSSM_GetContextAttribute
    (const CSSM_CONTEXT *Context,
    uint32 AttributeType,
    CSSM_CONTEXT_ATTRIBUTE_PTR *ContextAttribute)
```

DESCRIPTION

This function returns the value of a context attribute. Context references the cryptographic context to be searched for the attribute specified by *AttributeType*. If the specified attribute is not present then a NULL pointer is returned.

PARAMETERS

Context (input)
A pointer to the context.

AttributeType (input)
The attribute type of the desired attribute value.

ContextAttribute (output)
The pointer to the CSSM_CONTEXT_ATTRIBUTE that describes the context attributes associated with the handle *CCHandle* and the attribute type. The pointer will be set to NULL if the function fails. Call *CSSM_DeleteContextAttributes*( ) to free memory allocated by the CSSM.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_ATTRIBUTE_NOT_IN_CONTEXT

SEE ALSO

CSSM_DeleteContextAttributes( )
CSSM_GetContext( )

NAME

CSSM_UpdateContextAttributes

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_UpdateContextAttributes
    (CSSM_CC_HANDLE CCHandle,
    uint32 NumberOfAttributes,
    const CSSM_CONTEXT_ATTRIBUTE *ContextAttributes)
```

DESCRIPTION

This function updates one or more context attribute values stored as part of an existing context specified by *CCHandle*. The basic context structure is not modified by this function. Only the context attributes are updated.

The parameter *NumberOfAttributes* specifies the number of attributes to update. The new attribute values are specified in *ContextAttributes*. If an attribute provided in *ContextAttributes* is already present in the existing context, the existing value is replaced by the new value. If an attribute provided in *ContextAttributes* is not present in the existing context, then the new attribute is added. Attribute values are never deleted from the existing context.

PARAMETERS

CCHandle (input)
The handle to the context.

NumberOfAttributes (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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_CONTEXT_HANDLE
CSSMERR_CSSM_INVALID_ATTRIBUTE

SEE ALSO

CSSM_GetContextAttribute( )
CSSM_DeleteContextAttributes( )

NAME

CSSM_DeleteContextAttributes

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeleteContextAttributes
    (CSSM_CC_HANDLE CCHandle,
    uint32 NumberOfAttributes,
    const CSSM_CONTEXT_ATTRIBUTE *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 attributes to be deleted from the context. Only the attribute type is required. Any attribute values in the CSSM_CONTEXT_ATTRIBUTE structures are ignored.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSSM_INVALID_CONTEXT_HANDLE

SEE ALSO

CSSM_GetContextAttributes()
CSSM_UpdateContextAttributes()

7.7 Cryptographic Sessions and Controlled Access to Keys

The man-page definitions for Cryptographic Sessions and Controlled Access to Keys are presented in this section.

NAME

CSSM_CSP_Login

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_Login
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_DATA *LoginName,
    const void *Reserved)
```

DESCRIPTION

Logs the user into the CSP, allowing for multiple login types.

PARAMETERS

CSPHandle (input)
: Handle of the CSP to log into.

AccessCred (input)
: A pointer to the set of one or more credentials required to log into the token or cryptographic service provider. The credentials structure can contain an immediate value for the credential, such as a passphrase or PIN, or the caller can specify a callback function the CSP can use to obtain one or more credentials.

LoginName (input/optional)
: A name or ID of the caller. The value is used with the provided *AccessCred* to authenticate and authorize the caller for login with the CSP. The CSP can require that a name value be provided. If a name value is not provided, the CSP can assume a default name under which to perform the authentication and authorization check, or the login request can fail.

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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INVALID_LOGIN_NAME
CSSMERR_CSP_ALREADY_LOGGED_IN

SEE ALSO

CSSM_CSP_Logout()
CSSM_CSP_GetLoginAcl()
CSSM_CSP_ChangeLoginAcl()

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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_CSP_Login( )
CSSM_CSP_GetLoginAcl( )
CSSM_CSP_ChangeLoginAcl( )

NAME

CSSM_CSP_GetLoginAcl

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_GetLoginAcl
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_STRING *SelectionTag,
    uint32 *NumberOfAclInfos,
    CSSM_ACL_ENTRY_INFO_PTR *AclInfos)
```

DESCRIPTION

This function returns a description of zero or more ACL entries managed by the CSP and used to control login sessions with the CSP. The optional input *SelectionTag* restricts the returned descriptions to those ACL entries with a matching *EntryTag* value. If a *SelectionTag* value is specified and no matches are found, zero descriptions are returned. If no *SelectionTag* is specified, a description of all ACL entries used to control login sessions are returned by this function.

Each *AclInfo* structure contains:

- Public contents of an ACL entry
- ACL *EntryHandle*, which is a unique value defined and managed by the service provider

The public ACL entry information returned by this function includes:

- The subject type - A CSSM_LIST structure containing one element identifying the type of subject stored in the ACL entry.
- Delegation flag - A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in Authorization.
- Authorization array - A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period - A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A CSSM_STRING containing a user-defined value associated with the ACL entry.

PARAMETERS

CSPHandle (input)
: The module handle that identifies the Cryptographic service provider to perform this operation.

SelectionTag (input/optional)
: A CSSM_STRING value matching the user-defined tag value associated with one or more ACL entries controlling login sessions. To retrieve a description of all ACL entries controlling login sessions, this parameter must be NULL.

NumberOfAclInfos (output)
: The number of entries in the *AclInfos* array. If no ACL entry descriptions are returned, this value is zero.

AclInfos (output)
: An array of CSSM_ACL_ENTRY_INFO structures. The unique handle contained in this structure can be used during the current attach session and the current login session to reference specific ACL entries for editing. The structure is allocated by the service provider and must be released by the caller when the structure is no longer needed. If no ACL entry

descriptions are returned, this value is NULL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_CSP_ChangeLoginAcl( )
CSSM_CSP_Login( )
CSSM_CSP_Logout( )

NAME

CSSM_CSP_ChangeLoginAcl

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_ChangeLoginAcl
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_EDIT *AclEdit)
```

DESCRIPTION

This function edits the stored ACL controlling login sessions for a cryptographic service provider (CSP). The ACL is modified according to the edit mode and information provided in *AclEdit*.

The caller must have a login session in process and must be authorized to modify the target ACL. Caller authentication and authorization to edit the ACL is determined based on the caller-provided *AccessCred*.

The caller must be authorized to add, delete or replace the ACL entries controlling login to the CSP. When adding or replacing an ACL entry, the service provider must reject the creation of duplicate ACL entries.

When adding a new ACL entry to an ACL, the caller must provide a complete ACL entry prototype. All ACL entry items, except the ACL entry Subject must be provided as an immediate value in *AclEdit.NewEntry*. The ACL entry Subject can be provided as an immediate value, from a verifier with a protected data path, from an external authentication or authorization service, or through a callback function specified in *AclEdit.NewEntry.Callback*.

PARAMETERS

CSPHandle (input)
The module handle that identifies the Cryptographic service provider to perform this operation

AccessCred (input)
A pointer to the set of one or more credentials used to authenticate and validate the caller's authorization to modify the ACL controlling login sessions with the CSP. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. Traditionally a caller name has been used to establish the context of a login session. Certificates can be used for the same purpose. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

AclEdit (input)
A structure containing information that defines the edit operation. Valid operations include adding, replacing and deleting entries in an ACL managed by the service provider. The *AclEdit* can contain information for a new ACL entry and a handle uniquely identifying an existing ACL entry. The information controls the edit operation as follows:

Value of AclEdit.EditMode	Use of AclEdit.NewEntry and AclEdit.OldEntryHandle
CSSM_ACL_EDIT_MODE_ADD	Adds a new ACL entry to the set of ACL entries controlling login sessions with the CSP. The new ACL entry is created from the ACL entry prototype contained in *NewEntry*. *OldEntryHandle* is ignored for this *EditMode*.
CSSM_ACL_EDIT_MODE_DELETE	Deletes the ACL entry identified by *OldEntryHandle* and associated with login sessions with the CSP. *NewEntry* is ignored for this *EditMode*.
CSSM_ACL_EDIT_MODE_REPLACE	Replaces the ACL entry identified by *OldEntryHandle* and controlling login sessions with the CSP. The existing ACL is replaced based on the ACL entry prototype contained in the *NewEntry*.

When replacing an existing ACL entry, the caller must replace all of the items in an ACL entry. The replacement prototype includes:

- Subject type and value - A CSSM_LIST structure containing a typed Subject. The Subject identifies the entity authorized by this ACL entry.
- Delegation flag - A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in the authorization array.
- Authorization array - A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period - A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A CSSM_STRING containing a user-defined value associated with the ACL entry.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_CSP_GetLoginAcl( )
CSSM_CSP_Login( )
CSSM_CSP_Logout( )

NAME

CSSM_GetKeyAcl

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetKeyAcl
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY *Key,
    const CSSM_STRING *SelectionTag,
    uint32 *NumberOfAclInfos,
    CSSM_ACL_ENTRY_INFO_PTR *AclInfos)
```

DESCRIPTION

This function returns a description of zero or more ACL entries managed by the CSP and associated with the target Key. The optional input *SelectionTag* restricts the returned descriptions to those ACL entries with a matching *EntryTag* value. If a *SelectionTag* value is specified and no matches are found, zero descriptions are returned. If no *SelectionTag* is specified, a description of all ACL entries associated with the Key are returned by this function.

Each *AclInfo* structure contains:

- Public contents of an ACL entry
- ACL *EntryHandle*, which is a unique value defined and managed by the service provider

The public ACL entry information returned by this function includes:

- The subject type - A CSSM_LIST structure containing one element identifying the type of subject stored in the ACL entry.
- Delegation flag - A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in Authorization.
- Authorization array - A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period - A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A CSSM_STRING containing a user-defined value associated with the ACL entry.

PARAMETERS

CSPHandle (input)
: The module handle that identifies the Cryptographic service provider to perform this operation.

Key (input)
: A pointer to the target key whose associated ACL entries are scanned and returned.

SelectionTag (input/optional)
: A CSSM_STRING value matching the user-defined tag value associated with one or more ACL entries for the target Key. To retrieve a description of all ACL entries for the target Key, this parameter must be NULL.

NumberOfAclInfos (output)
: The number of entries in the *AclInfos* array. If no ACL entry descriptions are returned, this value is zero.

AclInfos (output)
: An array of CSSM_ACL_ENTRY_INFO structures. The unique handle contained in this

structure can be used during the current attach session to reference specific ACL entries for editing. The structure is allocated by the service provider and must be released by the caller when the structure is no longer needed. If no ACL entry descriptions are returned, this value is NULL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_ChangeKeyAcl()

NAME

CSSM_ChangeKeyAcl

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ChangeKeyAcl
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_EDIT *AclEdit,
    const CSSM_KEY *Key)
```

DESCRIPTION

This function edits the stored ACL associated with the target Key. The ACL is modified according to the edit mode and information provided in *AclEdit*.

The caller must be authorized to modify the target ACL. Caller authentication and authorization to edit the ACL is determined based on the caller-provided *AccessCred*.

The caller must be authorized to add, delete or replace the ACL entries associated with the target Key. When adding or replacing an ACL entry, the service provider must reject the creation of duplicate ACL entries.

When adding a new ACL entry to an ACL, the caller must provide a complete ACL entry prototype. All ACL entry items, except the ACL entry *Subject* must be provided as an immediate value in *AclEdit→NewEntry*. The ACL entry *Subject* can be provided as an immediate value, from a verifier with a protected data path, from an external authentication or authorization service, or through a callback function specified in *AclEdit→NewEntry→Callback*.

PARAMETERS

CSPHandle (input)
: The module handle that identifies the Cryptographic service provider to perform this operation

AccessCred (input)
: A pointer to the set of one or more credentials used to authenticate and validate the caller's authorization to modify the ACL associated with the Key. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

AclEdit (input)
: A structure containing information that defines the edit operation. Valid operations include: adding, replacing and deleting entries in an ACL managed by the service provider. The *AclEdit* can contain information for a new ACL entry and a handle uniquely identifying an existing ACL entry. The information controls the edit operation as follows:

Value of AclEdit.EditMode	Use of AclEdit.NewEntry and AclEdit.OldEntryHandle
CSSM_ACL_EDIT_MODE_ADD	Adds a new ACL entry to the set of ACL entries associated with the specified *Key*. The new ACL entry is created from the ACL entry prototype contained in *NewEntry*. *OldEntryHandle* is ignored for this edit mode.
CSSM_ACL_EDIT_MODE_DELETE	Deletes the ACL entry identified by *OldEntryHandle* and associated with the specified *Key*. *NewEntry* is ignored for this edit mode.
CSSM_ACL_EDIT_MODE_REPLACE	Replaces the ACL entry identified by *OldEntryHandle* and associated with the specified *Key*. The existing ACL is replaced based on the ACL entry prototype contained in the *NewEntry*.

When replacing an existing ACL entry, the caller must replace all of the items in an ACL entry. The replacement prototype includes:

- Subject type and value
  A CSSM_LIST structure containing a typed Subject. The Subject identifies the entity authorized by this ACL entry.
- Delegation flag
  A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in the authorization array.
- Authorization array
  A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period
  A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag
  A CSSM_STRING containing a user-defined value associated with the ACL entry.

Key (input)
A pointer to the target key whose associated ACL is being modified.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_GetKeyAcl()

NAME

CSSM_GetKeyOwner

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GetKeyOwner
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY *Key,
    CSSM_ACL_OWNER_PROTOTYPE_PTR Owner)
```

DESCRIPTION

This function returns a CSSM_ACL_OWNER_PROTOTYPE describing the current Owner of the Key.

PARAMETERS

CSPHandle (input)
The module handle that identifies the Cryptographic service provider to perform this operation.

Key (input)
A pointer to the target key whose associated Owner is returned.

Owner (output)
A CSSM_ACL_OWNER_PROTOTYPE describing the current Owner of the Key.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_ChangeKeyOwner( )

NAME

CSSM_ChangeKeyOwner

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ChangeKeyOwner
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    const CSSM_ACL_OWNER_PROTOTYPE *NewOwner)
```

DESCRIPTION

This function takes a CSSM_ACL_OWNER_PROTOTYPE defining the new Owner of the Key.

PARAMETERS

CSPHandle (input)
The module handle that identifies the Cryptographic service provider to perform this operation.

AccessCred (input)
A pointer to the set of one or more credentials used to prove the caller is the current Owner of the Key. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

Key (input)
A pointer to the target key whose associated Owner is changed.

NewOwner (Input)
A CSSM_ACL_OWNER_PROTOTYPE defining the new Owner of the Key.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_GetKeyOwner( )

NAME

CSSM_CSP_GetLoginOwner

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_GetLoginOwner
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ACL_OWNER_PROTOTYPE_PTR Owner)
```

DESCRIPTION

This function returns a CSSM_ACL_OWNER_PROTOTYPE describing the current Login Owner of the CSP.

PARAMETERS

CSPHandle (input)
The module handle that identifies the Cryptographic service provider to perform this operation.

Owner (output)
A CSSM_ACL_OWNER_PROTOTYPE describing the Login Owner.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_CSP_ChangeLoginOwner()

NAME

CSSM_CSP_ChangeLoginOwner

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_ChangeLoginOwner
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_OWNER_PROTOTYPE *NewOwner)
```

DESCRIPTION

This function takes a CSSM_ACL_OWNER_PROTOTYPE describing the new Login Owner.

PARAMETERS

CSPHandle (input)
The module handle that identifies the Cryptographic service provider to perform this operation.

AccessCred (input)
A pointer to the set of one or more credentials used to prove the caller is the current Login Owner. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

NewOwner (Input)
A CSSM_ACL_OWNER_PROTOTYPE defining the new Login Owner.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_CSP_GetLoginOwner( )

7.8 Cryptographic Operations

The man-page definitions for Cryptographic operations are presented in this section.

NAME

CSSM_SignData for the CSSM API
CSP_SignData for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_SignData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_ALGORITHMS DigestAlgorithm,
    CSSM_DATA_PTR Signature)

SPI:
CSSM_RETURN CSSMCSPI CSP_SignData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_ALGORITHMS DigestAlgorithm,
    CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function signs all data contained in the set of input buffers using the private key specified in the context. The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

Signing can include digesting the data and encrypting the digest or signing just the digest (already calculated by the application). If digesting the data and encrypting the digest, then the context should specify the combination digest/encryption algorithm (for example, CSSM_ALGID_MD5WithRSA). In this case, the *DigestAlgorithm* parameter must be set to CSSM_ALGID_NONE. If signing just the digest, then the context should specify just the encryption algorithm and the *DigestAlgorithm* parameter should specify the type of digest (for example, CSSM_ALGID_MD5). Also, *DataBufCount* must be 1.

If the signing algorithm is not reversible or strictly limits the size of the signed data, then the algorithm can specify signing without digesting. In this case, the sign operation is performed on the input data and the size of the input data is restricted by the service provider.

API 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 signed.

DataBufCount (input)
The number of DataBufs to be signed.

DigestAlgorithm (input)
If signing just a digest, specifies the type of digest. In this case, the context should only specify the encryption algorithm. If not signing just a digest, it must be CSSM_ALGID_NONE. In this case, the context should specify the combination digest/encryption algorithm.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR
CSSMERR_CSP_INVALID_DIGEST_ALGORITHM

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specifed in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_VerifyData()
CSSM_SignDataInit()
CSSM_SignDataUpdate()
CSSM_SignDataFinal()

For the CSP SPI:
CSP_VerifyData()
CSP_SignDataInit()
CSP_SignDataUpdate()
CSP_SignDataFinal()

NAME

CSSM_SignDataInit for the CSSM API
CSP_SignDataInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_SignDataInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_SignDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function initializes the staged sign data function.

For staged operations, a combination operation selecting both a digesting algorithm and a signing algorithm must be specified.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_SignData()
CSSM_SignDataUpdate()
CSSM_SignDataFinal()

For the CSP SPI:
CSP_SignData()
CSP_SignDataUpdate()
CSP_SignDataFinal()

NAME

CSSM_SignDataUpdate for the CSSM API
CSP_SignDataUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_SignDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)

SPI:
CSSM_RETURN CSSMCSPI CSP_SignDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function continues the staged signing process over all data contained in the set of input buffers. Signing is performed using the private key specified in the context.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_SignData()
CSSM_SignDataInit()
CSSM_SignDataFinal()

For the CSP SPI:
CSP_SignData()
CSP_SignDataInit()
CSP_SignDataFinal()

NAME

CSSM_SignDataFinal for the CSSM API
CSP_SignDataFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_SignDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Signature)

SPI:
CSSM_RETURN CSSMCSPI 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.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specifed in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_SignData( )
CSSM_SignDataInit( )
CSSM_SignDataUpdate( )

For the CSP SPI:
CSP_SignData( )
CSP_SignDataInit( )
CSP_SignDataUpdate( )

NAME

CSSM_VerifyData for the CSSM API
CSP_VerifyData for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_ALGORITHMS DigestAlgorithm,
    const CSSM_DATA *Signature)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_ALGORITHMS DigestAlgorithm,
    const CSSM_DATA *Signature)
```

DESCRIPTION

This function verifies all data contained in the set of input buffers based on the input signature.

Verifying can include digesting the data and decrypting the digest (from the signature) or verifying just the digest (already calculated by the application). If digesting the data and decrypting the digest, then the context should specify both digest and decryption algorithms (for example, CSSM_ALGID_MD5WithRSA). In this case, the *DigestAlgorithm* parameter must be set to CSSM_ALGID_NONE. If signing just the digest, then the context should specify just the decryption algorithm and the *DigestAlgorithm* parameter should specify the type of digest (for example, CSSM_ALGID_MD5). Also, *DataBufCount* must be 1.

If the signing algorithm is not reversible or strictly limits the size of the signed data, then the algorithm can specify verification without digesting. In this case, the verify operation is performed on the input data and the size of the input data is restricted by the service provider.

API 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.

DigestAlgorithm (input)
If verifying just a digest, specifies the type of digest. In this case, the context should only specify the encryption algorithm. If not verifying just a digest, it must be CSSM_ALGID_NONE. In this case, the context should specify the combination digest/encryption algorithm.

Signature (input)
A pointer to a CSSM_DATA structure which contains the signature and the size of the signature.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INPUT_LENGTH_ERROR
CSSMERR_CSP_VERIFY_FAILED
CSSMERR_CSP_INVALID_SIGNATURE
CSSMERR_CSP_INVALID_DIGEST_ALGORITHM

SEE ALSO

For the CSSM API:
CSSM_SignData()
CSSM_VerifyDataInit()
CSSM_VerifyDataUpdate()
CSSM_VerifyDataFinal()

For the CSP SPI:
CSP_SignData()
CSP_VerifyDataInit()
CSP_VerifyDataUpdate()
CSP_VerifyDataFinal()

NAME

CSSM_VerifyDataInit for the CSSM API
CSP_VerifyDataInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyDataInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function initializes the staged verify data function.

For staged operations, a combination operation selecting both a digesting algorithm and a verification algorithm must be specified.

API PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_VerifyDataUpdate()
CSSM_VerifyDataFinal()
CSSM_VerifyData()

For the CSP SPI:
CSP_VerifyDataUpdate()
CSP_VerifyDataFinal()
CSP_VerifyData()

NAME

CSSM_VerifyDataUpdate for the CSSM API
CSP_VerifyDataUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function continues the staged verification process over all data contained in the set of input. Verification will be based on the signature presented as input when finalizing the staged verification process.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_VerifyData( )
CSSM_VerifyDataInit( )
CSSM_VerifyDataFinal( )

For the CSP SPI:
CSP_VerifyData( )
CSP_VerifyDataInit( )
CSP_VerifyDataFinal( )

NAME

CSSM_VerifyDataFinal for the CSSM API
CSP_VerifyDataFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyDataFinal
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Signature)

SPI:
CSSM_BOOL CSSMCSPI CSP_VerifyDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Signature)
```

DESCRIPTION

This function finalizes the staged verify data function.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INPUT_LENGTH_ERROR
CSSMERR_CSP_VERIFY_FAILED
CSSMERR_CSP_INVALID_SIGNATURE

SEE ALSO

For the CSSM API:
CSSM_VerifyData()
CSSM_VerifyDataInit()
CSSM_VerifyDataUpdate()

For the CSP SPI:
CSP_VerifyData()
CSP_VerifyDataInit()
CSP_VerifyDataUpdate()

NAME

CSSM_DigestData for the CSSM API
CSP_DigestData for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DigestData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Digest)

SPI:
CSSM_RETURN CSSMCSPI CSP_DigestData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function computes a message digest for all data contained in the set of input buffers.

API 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.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specifed in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_DigestDataInit( )
CSSM_DigestDataUpdate( )
CSSM_DigestDataFinal( )
CSSM_DigestDataClone( )

For the CSP SPI:
CSP_DigestDataInit( )
CSP_DigestDataUpdate( )
CSP_DigestDataFinal( )
CSP_DigestDataClone( )

NAME

CSSM_DigestDataInit for the CSSM API
CSP_DigestDataInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DigestDataInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_DigestDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function initializes the staged message digest function.

API PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_DigestData()
CSSM_DigestDataUpdate()
CSSM_DigestDataClone()
CSSM_DigestDataFinal()

For the CSP SPI:
CSP_DigestData()
CSP_DigestDataUpdate()
CSP_DigestDataClone()
CSP_DigestDataFinal()

NAME

CSSM_DigestDataUpdate for the CSSM API
CSP_DigestDataUpdate

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DigestDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)

SPI:
CSSM_RETURN CSSMCSPI CSP_DigestDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function continues the staged process of digesting all data contained in the set of input buffers. The resulting digest value will be returned as part of the staged digesting process.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_DigestData( )
CSSM_DigestDataInit( )
CSSM_DigestDataClone( )
CSSM_DigestDataFinal( )

For the CSP SPI:
CSP_DigestData( )
CSP_DigestDataInit( )
CSP_DigestDataClone( )
CSP_DigestDataFinal( )

NAME

CSSM_DigestDataClone for the CSSM API
CSP_DigestDataClone for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DigestDataClone
    (CSSM_CC_HANDLE CCHandle,
    CSSM_CC_HANDLE *ClonednewCCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_DigestDataClone
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_CC_HANDLE ClonednewCCHandle)
```

DESCRIPTION

This function clones a given staged message digest context with its cryptographic attributes and intermediate result.

API PARAMETERS

CCHandle (input)
The handle that describes the context of a staged message digest operation.

ClonednewCCHandle (output)
The cloned digest context handle. The handle will be set to CSSM_INVALID_HANDLE if the function fails.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

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

For the CSSM API:
CSSM_DigestData()
CSSM_DigestDataInit()
CSSM_DigestDataUpdate()
CSSM_DigestDataFinal()

For the CSP SPI:
CSP_DigestData( )
CSP_DigestDataInit( )
CSP_DigestDataUpdate( )
CSP_DigestDataFinal( )

NAME

CSSM_DigestDataFinal for the CSSM API
CSP_DigestDataFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DigestDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Digest)

SPI:
CSSM_RETURN CSSMCSPI CSP_DigestDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function finalizes the staged message digest function.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_DigestData( )
CSSM_DigestDataInit( )
CSSM_DigestDataUpdate( )
CSSM_DigestDataClone( )

For the CSP SPI:
CSP_DigestData( )
CSP_DigestDataInit( )
CSP_DigestDataUpdate( )
CSP_DigestDataClone( )

NAME

CSSM_GenerateMac for the CSSM API
CSP_GenerateMac for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateMac
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Mac)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateMac
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function computes a message authentication code for all data contained in the set of input buffers.

API 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_DATA structure for the Message Authentication Code.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR

API 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.

SPI COMMENTS

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_GenerateMacInit( )
CSSM_GenerateMacUpdate( )
CSSM_GenerateMacFinal( )

For the CSP SPI:
CSP_GenerateMacInit( )
CSP_GenerateMacUpdate( )
CSP_GenerateMacFinal( )

NAME

CSSM_GenerateMacInit for the CSSM API
CSP_GenerateMacInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateMacInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateMacInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function initializes the staged message authentication code function.

API PARAMETER

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_GenerateMac( )
CSSM_GenerateMacUpdate( )
CSSM_GenerateMacFinal( )

For the CSP SPI:
CSP_GenerateMac( )
CSP_GenerateMacUpdate( )
CSP_GenerateMacFinal( )

NAME

CSSM_GenerateMacUpdate for the CSSM API
CSP_GenerateMacUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateMacUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateMacUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function continues the staged process of computing a message authentication code over all data contained in the set of input buffers. The authentication code will be returned as a result of the final code generation step.

API 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.

ADDITIONAL SPI PARAMETER

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.

Context (input)

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_GenerateMac()
CSSM_GenerateMacInit()
CSSM_GenerateMacFinal()

For the CSP SPI:
CSP_GenerateMac( )
CSP_GenerateMacInit( )
CSP_GenerateMacFinal( )

NAME

CSSM_GenerateMacFinal for the CSSM API
CSP_GenerateMacFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateMacFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Mac)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateMacFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code function.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS ON API

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.

COMMENTS ON SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_GenerateMac( )
CSSM_GenerateMacInit( )
CSSM_GenerateMacUpdate( )

For the CSP SPI:
CSP_GenerateMac( )
CSP_GenerateMacInit( )
CSP_GenerateMacUpdate( )

NAME

CSSM_VerifyMac for the CSSM API
CSP_VerifyMac for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyMac
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    const CSSM_DATA *Mac)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyMac
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount,
    const CSSM_DATA *Mac)
```

DESCRIPTION

This function verifies the message authentication code over all data contained in the set of input buffers based on the input signature.

API 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.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INPUT_LENGTH_ERROR
CSSMERR_CSP_VERIFY_FAILED
CSSMERR_CSP_INVALID_SIGNATURE

SEE ALSO

For the CSSM API:
CSSM_VerifyMacInit( )
CSSM_VerifyMacUpdate( )
CSSM_VerifyMacFinal( )

For the CSP SPI:
CSP_VerifyMacInit( )
CSP_VerifyMacUpdate( )
CSP_VerifyMacFinal( )

NAME

CSSM_VerifyMacInit for the CSSM API
CSP_VerifyMacInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyMacInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyMacInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function initializes the staged message authentication code verification function.

API PARAMETER

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_VerifyMac()
CSSM_VerifyMacUpdate()
CSSM_VerifyMacFinal()

For the CSP SPI:
CSP_VerifyMac()
CSP_VerifyMacUpdate()
CSP_VerifyMacFinal()

NAME

CSSM_VerifyMacUpdate for the CSSM API
CSP_VerifyMacUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyMacUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyMacUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function continues the staged process of verifying the message authentication code over all data in the set of input buffers. Verification will be based on the authentication code presented as input when finalizing the staged verification process.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_VerifyMac()
CSSM_VerifyMacInit()
CSSM_VerifyMacFinal()

For the CSP SPI:
CSP_VerifyMac( )
CSP_VerifyMacInit( )
CSP_VerifyMacFinal( )

NAME

CSSM_VerifyMacFinal for the CSSM API
CSP_VerifyMacFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyMacFinal
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Mac)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyMacFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code verification function.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INPUT_LENGTH_ERROR
CSSMERR_CSP_VERIFY_FAILED
CSSMERR_CSP_INVALID_SIGNATURE

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_VerifyMac()
CSSM_VerifyMacInit()
CSSM_VerifyMacUpdate()

For the CSP SPI:
CSP_VerifyMac()
CSP_VerifyMacInit()
CSP_VerifyMacUpdate()

NAME

CSSM_QuerySize for the CSSM API
CSP_QuerySize for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_QuerySize
    (CSSM_CC_HANDLE CCHandle,
    CSSM_BOOL Encrypt,
    uint32 QuerySizeCount,
    CSSM_QUERY_SIZE_DATA_PTR DataBlockSizes)

SPI:
CSSM_RETURN CSSMCSPI CSP_QuerySize
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    CSSM_BOOL Encrypt,
    uint32 QuerySizeCount,
    CSSM_QUERY_SIZE_DATA_PTR DataBlockSizes)
```

DESCRIPTION

This function queries for the size of the output data for a cryptographic operation. If the context is an encryption or decryption context type then the *Encrypt* parameter will determine which operation is being performed. If *Encrypt* is set to CSSM_TRUE then it is an encrypt operation, otherwise it is a decrypt operation. For all other context types the *Encrypt* parameter is ignored. 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.

API 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 CSSM_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.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_QUERY_SIZE_UNKNOWN

SEE ALSO

For the CSSM API:
CSSM_EncryptData( )
CSSM_EncryptDataUpdate( )
CSSM_DecryptData( )
CSSM_DecryptDataUpdate( )
CSSM_SignData( )
CSSM_VerifyData( )
CSSM_DigestData( )
CSSM_GenerateMac( )

For the CSP SPI:
CSP_EncryptData( )
CSP_EncryptDataUpdate( )
CSP_DecryptData( )
CSP_DecryptDataUpdate( )
CSP_SignData( )
CSP_VerifyData( )
CSP_DigestData( )
CSP_GenerateMac( )

NAME

CSSM_EncryptData for the CSSM API
CSP_EncryptData for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_EncryptData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted,
    CSSM_DATA_PTR RemData)

SPI:
CSSM_RETURN CSSMCSPI CSP_EncryptData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted,
    CSSM_DATA_PTR RemData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function encrypts all data contained in the set of input buffers using information in the context. The *CSSM_QuerySize*() function can be used to estimate the output buffer size required. The minimum number of buffers required to contain the resulting cipher text is produced as output. If the cipher text result does not fit within the set of output buffers, the remaining cipher text is returned in the single output buffer *RemData*.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API 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 remaining cypher text if there is not enough buffer space available in the output data structures.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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. In-place encryption can be done by supplying the same input and output buffers.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

For the CSP SPI:
CSP_QuerySize( )
CSP_DecryptData( )
CSP_EncryptDataInit( )
CSP_EncryptDataUpdate( )
CSP_EncryptDataFinal( )

NAME

CSSM_EncryptDataP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptDataP
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted,
    CSSM_DATA_PTR RemData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_EncryptData*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_EncryptData*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

SEE ALSO

CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

NAME

CSSM_EncryptDataInit for the CSSM API
CSP_EncryptDataInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_EncryptDataInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSP_EncryptDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    CSSM_PRIVILEGE Privilege)
```

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.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API PARAMETER

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )

CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

For the CSP SPI:
CSP_QuerySize( )
CSP_DecryptData( )
CSP_EncryptDataInit( )
CSP_EncryptDataUpdate( )
CSP_EncryptDataFinal( )

NAME

CSSM_EncryptDataInitP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptDataInitP
    (CSSM_CC_HANDLE CCHandle,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function similar to *CSSM_EncryptDataInit*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

For staged operations using privilege initialization functions *CSSM_EncryptDataInitP*( ), the completion functions *CSSM_EncryptDataUpdate*( ) and *CSSM_EncryptDataFinalize*( ) are used.

PARAMETERS

See *CSSM_EncryptDataInit*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See *CSSM_EncryptDataInit*( ).

SEE ALSO

CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

NAME

CSSM_EncryptDataUpdate for the CSSM API
CSP_EncryptDataUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_EncryptDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted)

SPI:
CSSM_RETURN CSSMCSPI CSP_EncryptDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted)
```

DESCRIPTION

This function continues the staged encryption process over all data in the set of input buffers. There can be algorithm-specific and token-specific rules restricting the lengths of data in *CSSM_EncryptUpdate*( ) calls, but multiple input buffers are supported. The minimum number of buffers required to contain the resulting cipher text is produced as output. Excess output buffer space is not remembered across staged encryption calls. Each staged call begins filling one or more new output buffers. The *CSSM_QuerySize*( ) function can be used to estimate the output buffer size required for each update call.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR API

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. In-place encryption can be done by supplying the same input and output buffers.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

For the CSP SPI:
CSP_QuerySize( )
CSP_DecryptData( )
CSP_EncryptDataInit( )
CSP_EncryptDataFinal( )

NAME

CSSM_EncryptDataFinal for the CSSM API
CSP_EncryptDataFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_EncryptDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)

SPI:
CSSM_RETURN CSSMCSPI CSP_EncryptDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged encryption process by returning any remaining cipher text not returned in the previous staged encryption call. The cipher text is returned in a single buffer.

API 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 if necessary.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

For the CSP SPI:
CSP_EncryptData( )
CSP_EncryptDataInit( )
CSP_EncryptDataUpdate( )

NAME

CSSM_DecryptData for the CSSM API
CSP_DecryptData for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DecryptData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted,
    CSSM_DATA_PTR RemData)

SPI:
CSSM_RETURN CSSMCSPI CSP_DecryptData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_DATA *CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted,
    CSSM_DATA_PTR RemData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function decrypts all data contained in the set of input buffers using information in the context. The *CSSM_QuerySize*( ) (CSSM API) or *CSP_QuerySize*( ) (CSP SPI) function can be used to estimate the output buffer size required. The minimum number of buffers required to contain the resulting plain text is produced as output. If the plain text result does not fit within the set of output buffers, the remaining plain text is returned in the single output buffer *RemData*.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API 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 remaining plain text if there is not enough buffer space available in the output data structures.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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. In-place decryption can be done by supplying the same input and output buffers.

COMMENTS FOR SPI

The output is returned to the caller a specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_EncryptData( )
CSSM_DecryptDataInit( )
CSSM_DecryptDataUpdate( )
CSSM_DecryptDataFinal( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

For the CSP SPI:
CSP_QuerySize( )
CSP_EncryptData( )

CSP_DecryptDataInit( )
CSP_DecryptDataUpdate( )
CSP_DecryptDataFinal( )

NAME

CSSM_DecryptDataP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DecryptDataP
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted,
    CSSM_DATA_PTR RemData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function similar to *CSSM_DecryptData*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_DecryptData*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

SEE ALSO

CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

NAME

CSSM_DecryptDataInit for the CSSM API
CSP_DecryptDataInit for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DecryptDataInit
    (CSSM_CC_HANDLE CCHandle)

SPI:
CSSM_RETURN CSSMCSPI CSSM_CSP_DecryptDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function initializes the staged decrypt function.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API PARAMETER

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_DecryptData()
CSSM_DecryptDataUpdate()
CSSM_DecryptDataFinal()
CSSM_DecryptDataP()
CSSM_DecryptDataInitP()

For the CSP SPI:
CSP_DecryptData()

CSP_DecryptDataUpdate()
CSP_DecryptDataFinal()

NAME

CSSM_DecryptDataInitP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DecryptDataInitP
    (CSSM_CC_HANDLE CCHandle,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_DecryptDataInit*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

For staged operations using privilege initialization functions *CSSM_DecryptDataInitP*( ), the completion functions *CSSM_DecryptDataUpdate*( ) and *CSSM_DecryptDataFinalize*( ) are used.

PARAMETERS

See *CSSM_DecryptDataInit*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_EncryptDataInit( )
CSSM_EncryptDataUpdate( )
CSSM_EncryptDataFinal( )
CSSM_EncryptDataP( )
CSSM_EncryptDataInitP( )
CSSM_DecryptP( )
CSSM_DecryptDataInitP( )

NAME

CSSM_DecryptDataUpdate for the CSSM API
CSP_DecryptDataUpdate for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DecryptDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted)

SPI:
CSSM_RETURN CSSMCSPI CSP_DecryptDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted)
```

DESCRIPTION

This function continues the staged decryption process over all data in the set of input buffers. There can be algorithm-specific and token-specific rules restricting the lengths of data in *CSSM_DecryptUpdate*() calls, but multiple input buffers are supported. The minimum number of buffers required to contain the resulting plain text is produced as output. Excess output buffer space is not remembered across staged decryption calls. Each staged call begins filling one or more new output buffers. The *CSSM_QuerySize*() (CSSM API) or *CSP_QuerySize*() (CSP SPI) function can be used to estimate the output buffer size required for each update call.

API 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.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR API

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. In-place decryption can be done by supplying the same input and output buffers.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_QuerySize( )
CSSM_DecryptData( )
CSSM_DecryptDataInit( )
CSSM_DecryptDataFinal( )

For the CSP SPI:
CSP_QuerySize( )
CSP_DecryptData( )
CSP_DecryptDataInit( )
CSP_DecryptDataFinal( )

NAME

CSSM_DecryptDataFinal for the CSSM API
CSP_DecryptDataFinal for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DecryptDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)

SPI:
CSSM_RETURN CSSMCSPI CSP_DecryptDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged decryption process by returning any remaining plain text not returned in the previous staged decryption call. The plain text is returned in a single buffer.

API 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, if necessary.

ADDITIONAL SPI PARAMETER

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.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_BLOCK_SIZE_MISMATCH
CSSMERR_CSP_OUTPUT_LENGTH_ERROR

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

SEE ALSO

For the CSSM API:
CSSM_DecryptData( )
CSSM_DecryptDataInit( )
CSSM_DecryptDataUpdate( )

For the CSP SPI:
CSP_DecryptData( )
CSP_DecryptDataInit( )
CSP_DecryptDataUpdate( )

NAME

CSSM_QueryKeySizeInBits for the CSSM API
CSP_QueryKeySizeInBits for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_QueryKeySizeInBits
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY *Key,
    CSSM_KEY_SIZE_PTR KeySize)

SPI:
CSSM_RETURN CSSMCSPI CSP_QueryKeySizeInBits
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_KEY *Key,
    CSSM_KEY_SIZE_PTR KeySize)
```

DESCRIPTION

This function queries a Cryptographic Service Provider (CSP) for the logical and effective sizes of a specified key.

The cryptographic service provider (handle) and the key can be specified either in the cryptographic context or as parameters to the function call. If a valid cryptographic context handle parameter is specified, the CSP handle and key parameters are ignored.

API PARAMETERS

CSPHandle (input/optional)

The handle that describes the cryptographic service provider module used to perform this function.

For the API, this parameter is ignored if a valid cryptographic context handle is specified.

CCHandle (input/optional)

A handle to a context that describes a cryptographic operation. The cryptographic context should contain a handle to the CSP that is being queried and the key about which key-size information is being requested.

Key (input/optional)

A pointer to a CSSM_KEY structure containing the key about which key-size information is being requested. This parameter is ignored if a valid cryptographic context handle is specified.

KeySize (output)

Pointer to a CSSM_KEY_SIZE data structure. The logical and effective sizes (in bits) for the key are returned in this structure.

For the API, if no context handle is provided, only the CSSM_KEY_SIZE *LogicalKeySizeInBits* field is set.

ADDITIONAL SPI PARAMETER

Context (input)

Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_QUERY_SIZE_UNKNOWN

SEE ALSO

For the CSSM API:
CSSM_GenerateRandom()
CSSM_GenerateKeyPair()
CSSM_GenerateKey()

For the CSP SPI:
CSP_GenerateRandom()
CSP_GenerateKeyPair()
CSP_GenerateKey()

NAME

CSSM_GenerateKey for the CSSM API
CSP_GenerateKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateKey
    (CSSM_CC_HANDLE CCHandle,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR Key)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR Key)
```

DESCRIPTION

This function generates a symmetric key. The *KeyUsage*, and *KeyAttr* are used to initialize the keyheader for the newly created key. These values are not retained in the cryptographic Context, which contains additional parameters for this operation. 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.

Authorization policy can restrict the set of callers who can create a new resource. In this case, the caller must present a set of access credentials for authorization. Upon successfully authenticating the credentials, the template that verified the presented samples identifies the ACL entry that will be used in the authorization computation. If the caller is authorized, the new resource is created.

The caller must provide an initial ACL entry to be associated with the newly created resource. This entry is used to control future access to the new resource and (since the subject is deemed to be the "Owner") exercise control over its associated ACL. The caller can specify the following items for initializing an ACL entry:

- Subject - A CSSM_LIST structure, containing the type of the subject and a template value that can be used to verify samples that are presented in credentials when resource access is requested.
- Delegation flag - A value indicating whether the Subject can delegate the permissions recorded in the *AuthorizationTag*. (This item only applies to public key subjects).
- Authorization tag - The set of permissions that are granted to the Subject.
- Validity period - The start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A user-defined string value associated with the ACL entry.

The service provider can modify the caller-provided initial ACL entry to conform to any innate resource-access policy that the service provider may be required to enforce. If the initial ACL entry provided by the caller contains values or permissions that are not supported by the service provider, then the service provider can modify the initial ACL appropriately or can fail the request to create the new resource. Service providers list their supported *AuthorizationTag* values in their Module Directory Services primary record.

API 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/optional)
Pointer to a byte string that will be used as the label for the key.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the CSP to acquire the credentials and/or the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be an empty list.

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.

ADDITIONAL/CHANGED SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

SEE ALSO

For the CSSM API:
CSSM_GenerateRandom( )
CSSM_GenerateKeyPair( )

For the CSP SPI:
CSP_GenerateRandom( )
CSP_GenerateKeyPair( )

NAME

CSSM_GenerateKeyP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateKeyP
    (CSSM_CC_HANDLE CCHandle,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR Key,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_GenerateKey*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_GenerateKey*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See *CSSM_GenerateKey*( ).

SEE ALSO

CSSM_GenerateRandom( )
CSSM_GenerateKeyPairP( )

NAME

CSSM_GenerateKeyPair for the CSSM API
CSP_GenerateKeyPair for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateKeyPair
    (CSSM_CC_HANDLE CCHandle,
    uint32 PublicKeyUsage,
    uint32 PublicKeyAttr,
    const CSSM_DATA *PublicKeyLabel,
    CSSM_KEY_PTR PublicKey,
    uint32 PrivateKeyUsage,
    uint32 PrivateKeyAttr,
    const CSSM_DATA *PrivateKeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR PrivateKey)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateKeyPair
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    uint32 PublicKeyUsage,
    uint32 PublicKeyAttr,
    const CSSM_DATA *PublicKeyLabel,
    CSSM_KEY_PTR PublicKey,
    uint32 PrivateKeyUsage,
    uint32 PrivateKeyAttr,
    const CSSM_DATA *PrivateKeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR PrivateKey,
    CSSM_PRIVILEGE Privilege)
```

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.

Authorization policy can restrict the set of callers who can create a new resource. In this case, the caller must present a set of access credentials for authorization. Upon successfully authenticating the credentials, the template that verified the presented samples identifies the ACL entry that will be used in the authorization computation. If the caller is authorized, the new resource is created.

The caller must provide an initial ACL entry to be associated with the newly created resource. This entry is used to control future access to the new resource and (since the subject is deemed to be the "Owner") exercise control over its associated ACL. The caller can specify the following items for initializing an ACL entry:

- Subject - A CSSM_LIST structure, containing the type of the subject and a template value that can be used to verify samples that are presented in credentials when resource access is requested.

- Delegation flag - A value indicating whether the Subject can delegate the permissions recorded in the *AuthorizationTag*. (This item only applies to public key subjects).
- Authorization tag - The set of permissions that are granted to the Subject.
- Validity period - The start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A user-defined string value associated with the ACL entry.

The service provider can modify the caller-provided initial ACL entry to conform to any innate resource-access policy that the service provider may be required to enforce. If the initial ACL entry provided by the caller contains values or permissions that are not supported by the service provider, then the service provider can modify the initial ACL appropriately or can fail the request to create the new resource. Service providers list their supported *AuthorizationTag* values in their Module Directory Services primary record.

API 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/optional)
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 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.

PrivateKeyAttr (input)
A bit mask defining attribute values for the new private key.

PrivateKeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the private key.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the CSP to acquire the credentials and/or the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be an empty list.

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*, *PrivateKeyUsage*, *PrivateKeyAttr*, and *PrivateKeyLabel* input parameters.

ADDITIONAL SPI 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.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified privilege.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS

COMMENTS

The *KeyData* fields of the CSSM_KEY structures are allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

SEE ALSO

For the CSSM API:
CSSM_GenerateKey( )
CSSM_GenerateRandom( )

For the CSP SPI:
CSP_GenerateKey( )
CSP_GenerateRandom( )

NAME

CSSM_GenerateKeyPairP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateKeyPairP
    (CSSM_CC_HANDLE CCHandle,
    uint32 PublicKeyUsage,
    uint32 PublicKeyAttr,
    const CSSM_DATA *PublicKeyLabel,
    CSSM_KEY_PTR PublicKey,
    uint32 PrivateKeyUsage,
    uint32 PrivateKeyAttr,
    const CSSM_DATA *PrivateKeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR PrivateKey,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_GenerateKeyPair*(). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_GenerateKeyPair*().

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS

SEE ALSO

CSSM_GenerateKeyPair().

NAME

CSSM_GenerateRandom for the CSSM API
CSP_GenerateRandom for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateRandom
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RandomNumber)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateRandom
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    CSSM_DATA_PTR RandomNumber)
```

DESCRIPTION

This function generates random data.

API 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.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

NAME

CSSM_CSP_ObtainPrivateKeyFromPublicKey for the CSSM API
CSP_ObtainPrivateKeyFromPublicKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CSP_ObtainPrivateKeyFromPublicKey
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY *PublicKey,
    CSSM_KEY_PTR PrivateKey)

SPI:
CSSM_RETURN CSSMCSPI CSP_ObtainPrivateKeyFromPublicKey
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY *PublicKey,
    CSSM_KEY_PTR PrivateKey)
```

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.

API AND SPI 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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_PRIVATE_KEY_NOT_FOUND

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

NAME

CSSM_WrapKey for the CSSM API
CSP_WrapKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_WrapKey
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    const CSSM_DATA *DescriptiveData,
    CSSM_WRAP_KEY_PTR WrappedKey)

SPI:
CSSM_RETURN CSSMCSPI CSP_WrapKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    const CSSM_DATA *DescriptiveData,
    CSSM_WRAP_KEY_PTR WrappedKey,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function wraps the supplied key using the context. It allows a key to be exported from a CSP. Four types of wrapping exist:

1. Wrap a symmetric key with a symmetric key.
2. Wrap a symmetric key with an asymmetric public key.
3. Wrap an asymmetric private key with a symmetric key.
4. Wrap an asymmetric private key with an asymmetric public key.

For types 1 and 3, a symmetric context should be provided. For types 2 and 4, an asymmetric context is provided. If there is a CSSM_ATTRIBUTE_WRAPPED_KEY_FORMAT argument in the context represented by the *CCHandle*, the value of the attribute specifies the format of the wrapped key. If this argument is not present, the symmetric key is wrapped according to CMS for types 1 and 3, and according to PKCS8 for types 2 and 4. If the wrapping algorithm in the context is CSSM_ALGID_NONE, then the key is returned in raw format, if permitted and supported by the CSP (in this case the CSSM_ATTRIBUTE_WRAPPED_KEY_FORMAT attribute is ignored). All significant key attributes are incorporated into the *KeyHeader* of the returned *WrappedKey*, such that the state of the key can be fully restored by the *unwrap* process.

The CSP can require that the cryptographic context includes access credentials for authentication and authorization checks when using the secret or private key.

API PARAMETERS

CCHandle (input)
The handle to the context that describes this cryptographic operation.

AccessCred (input)
A pointer to the set of one or more credentials required to access the private or secret key to be exported from the CSP. The credentials structure can contain an immediate value for the credential, such as a passphrase, or the caller can specify a callback function the CSP can use

to obtain one or more credentials.

Key (input)
A pointer to the key to be wrapped.

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.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

SEE ALSO

For the CSSM API:
CSSM_UnwrapKey( )

For the CSP SPI:
CSP_UnwrapKey( )

NAME

CSSM_WrapKeyP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_WrapKeyP
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_KEY *Key,
    const CSSM_DATA *DescriptiveData,
    CSSM_WRAP_KEY_PTR WrappedKey,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_WrapKey*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_WrapKey*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See *CSSM_WrapKey*( ).

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) call, or with the memory functions registered for the *CSPHandle*.

NAME

CSSM_UnwrapKey for the CSSM API
CSP_UnwrapKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_UnwrapKey
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY *PublicKey,
    const CSSM_WRAP_KEY *WrappedKey,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR UnwrappedKey,
    CSSM_DATA_PTR DescriptiveData)

SPI:
CSSM_RETURN CSSMCSPI CSP_UnwrapKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    const CSSM_KEY *PublicKey,
    const CSSM_WRAP_KEY *WrappedKey,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR UnwrappedKey,
    CSSM_DATA_PTR DescriptiveData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function unwraps the wrapped key using the context. The wrapped key can be a symmetric key or a private key. If the unwrapping algorithm is a symmetric algorithm, then a symmetric context must be provided. If the unwrapping algorithm is an asymmetric algorithm, then an asymmetric context must be provided. If the key is a private key, then an asymmetric context must be provide describing the unwrapping algorithm. The CSP can require the caller to provide credentials authorizing the caller to store the unwrapped key within the CSP. The CSP can also require that the caller provide an initial ACL entry to control future access to the newly stored key. These credentials and the initial ACL entry value are provided in *CredAndAclEntry* parameter. If the unwrapping algorithm is CSSM_ALGID_NONE 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 CSSM_ALGID_NONE 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.

Authorization policy can restrict the set of callers who can create a new resource. In this case, the caller must present a set of access credentials for authorization. Upon successfully authenticating the credentials, the template that verified the presented samples identifies the ACL entry that will be used in the authorization computation. If the caller is authorized, the new resource is created.

The caller must provide an initial ACL entry to be associated with the newly created resource. This entry is used to control future access to the new resource and (since the subject is deemed to be the "Owner") exercise control over its associated ACL. The caller can specify the following items for initializing an ACL entry:

- Subject - A CSSM_LIST structure, containing the type of the subject and a template value that can be used to verify samples that are presented in credentials when resource access is requested.
- Delegation flag - A value indicating whether the Subject can delegate the permissions recorded in the *AuthorizationTag*. (This item only applies to public key subjects).
- Authorization tag - The set of permissions that are granted to the Subject.
- Validity period - The start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A user-defined string value associated with the ACL entry.

The service provider can modify the caller-provided initial ACL entry to conform to any innate resource-access policy that the service provider may be required to enforce. If the initial ACL entry provided by the caller contains values or permissions that are not supported by the service provider, then the service provider can modify the initial ACL appropriately or can fail the request to create the new resource. Service providers list their supported *AuthorizationTag* values in their Module Directory Services primary record.

API PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation.

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)
A bit mask indicating all permitted uses for the unwrapped key. If no value is specified, the CSP defines the usage mask for the unwrapped key.

KeyAttr (input)
A bit mask defining other 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.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the CSP to acquire the credentials and/or the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be an empty list.

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.

ADDITIONAL SPI 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.

Privilege (input)
The export privilege to be applied during the cryptographic operation. This parameter is forwarded to the CSP after CSSM verifies the caller and service provider privilege set includes the specified PRIVILEGE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS
CSSMERR_CSP_PUBLIC_KEY_INCONSISTENT
CSSMERR_CSP_PRIVATE_KEY_ALREADY_EXIST

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

SEE ALSO

For the CSSM API:
CSSM_WrapKey( )

For the CSP SPI:
CSP_WrapKey( )

NAME

CSSM_UnwrapKeyP

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_UnwrapKeyP
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY *PublicKey,
    const CSSM_WRAP_KEY *WrappedKey,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR UnwrappedKey,
    CSSM_DATA_PTR DescriptiveData,
    CSSM_PRIVILEGE Privilege)
```

DESCRIPTION

This function is similar to *CSSM_UnwrapKey*( ). It also accepts a USEE tag as a privilege request parameter. CSSM checks that either its own privilege set or the Application's privilege set (if the Application is signed) includes the tag. If the tag is found, and the service provider privilege set indicates that it is supported, the tag is forwarded to the service provider.

PARAMETERS

See *CSSM_UnwrapKey*( ).

Privilege (input)
The privilege to be applied during the cryptographic operation.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS
CSSMERR_CSP_PUBLIC_KEY_INCONSISTENT
CSSMERR_CSP_PRIVATE_KEY_ALREADY_EXIST

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) call, or with the memory functions registered for the CSPHandle.

NAME

CSSM_DeriveKey for the CSSM API
CSP_DeriveKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DeriveKey
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Param,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR DerivedKey)

SPI:
CSSM_RETURN CSSMCSPI CSP_DeriveKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    CSSM_DATA_PTR Param,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA *KeyLabel,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    CSSM_KEY_PTR DerivedKey)
```

DESCRIPTION

This function derives a new symmetric key using the context and/or information from the base key in the context. The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

Authorization policy can restrict the set of callers who can create a new resource. In this case, the caller must present a set of access credentials for authorization. Upon successfully authenticating the credentials, the template that verified the presented samples identifies the ACL entry that will be used in the authorization computation. If the caller is authorized, the new resource is created.

The caller must provide an initial ACL entry to be associated with the newly created resource. This entry is used to control future access to the new resource and (since the subject is deemed to be the "Owner") exercise control over its associated ACL. The caller can specify the following items for initializing an ACL entry:

- Subject - A CSSM_LIST structure, containing the type of the subject and a template value that can be used to verify samples that are presented in credentials when resource access is requested.
- Delegation flag - A value indicating whether the Subject can delegate the permissions recorded in the *AuthorizationTag*. (This item only applies to public key subjects).
- Authorization tag - The set of permissions that are granted to the Subject.
- Validity period - The start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A user-defined string value associated with the ACL entry.

The service provider can modify the caller-provided initial ACL entry to conform to any innate resource-access policy that the service provider may be required to enforce. If the initial ACL entry provided by the caller contains values or permissions that are not supported by the service provider, then the service provider can modify the initial ACL appropriately or can fail the request to create the new resource. Service providers list their supported *AuthorizationTag* values in their Module Directory Services primary record.

The CSP can require that the cryptographic context include access credentials for authentication and authorization checks when using a private key or a secret key.

API PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Param (input/output)
This parameter varies depending on the derivation algorithm. Password based derivation algorithms use this parameter to return a cipher block chaining initialization vector. Concatenation algorithms use this parameter to get the second item to concatenate.

KeyUsage (input)
A bit mask indicating all permitted uses for the new derived key.

KeyAttr (input)
A bit mask defining other 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.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the CSP to acquire the credentials and/or the subject of the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be empty.

DerivedKey (output)
A pointer to a CSSM_KEY structure that returns the derived key.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_KEY_LABEL_ALREADY_EXISTS

COMMENTS

The *KeyData* field of the CSSM_KEY structure is allocated by the CSP. The application is required to free this memory using the *CSSM_FreeKey*( ) (CSSM API) or *CSP_FreeKey*( ) (CSP SPI) call, or with the memory functions registered for the *CSPHandle*.

SEE ALSO

CSSM_CSP_CreateDeriveKeyContext( )

NAME

CSSM_FreeKey for the CSSM API
CSP_FreeKey for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_FreeKey
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    CSSM_KEY_PTR KeyPtr,
    CSSM_BOOL Delete)

SPI:
CSSM_RETURN CSSMCSPI CSP_FreeKey
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    CSSM_KEY_PTR KeyPtr,
    CSSM_BOOL Delete)
```

DESCRIPTION

This function requests the cryptographic service provider to clean up any key material associated with the key, and to possibly delete the key from the CSP completely. 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 can have an effect on the long term persistence or transience of the key.

API AND SPI PARAMETERS

CSPHandle (input)
The handle that describes the module to perform this operation.

AccessCred (input/optional)
If the target key referenced by *KeyPtr* is protected and *Delete* has the value CSSM_TRUE, this parameter must contain the certificates and samples required to access the target key. The certificates must be presented as immediate values in the input structure. The samples can be immediate values, be obtained through a protected mechanism, or be obtained through a *callback* function.

KeyPtr (input)
The key whose associated keying material can be discarded at this time.

Delete (input)
If this value is CSSM_TRUE, the key data in the key structure will be removed and any internal storage related to that key will also be removed. In this case the key no longer exists in any form, unless previously wrapped out of the CSP by the application. If this value is CSSM_FALSE, then only the resources related to the key structure are released. The key may still be accessible by other means internally to the CSP.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

NAME

CSSM_GenerateAlgorithmParams for the CSSM API
CSP_GenerateAlgorithmParams for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GenerateAlgorithmParams
    (CSSM_CC_HANDLE CCHandle,
    uint32 ParamBits,
    CSSM_DATA_PTR Param)

SPI:
CSSM_RETURN CSSMCSPI CSP_GenerateAlgorithmParams
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    uint32 ParamBits,
    CSSM_DATA_PTR Param,
    uint32 *NumberOfUpdatedAttributes,
    CSSM_CONTEXT_ATTRIBUTE_PTR *UpdatedAttributes)
```

DESCRIPTION

This function generates algorithm parameters for the specified context. These parameters include *Diffie-Hellman* key agreement parameters and DSA key generation parameters. In most cases the algorithm parameters will be added directly to the cryptographic context (by returning an array of CSSM_CONTEXT_ATTRIBUTE structures), but an algorithm may return some data to the caller via the *Param* parameter. The generated parameters are added to the context as an attribute of type CSSM_ATTRIBUTE_ALG_PARAMS. Other attributes returned are added to the context, or replace existing values in the context.

API 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 a CSSM_DATA structure used to provide information to the parameter generation process, or to receive information resulting from the generation process that is not required as a parameter to the algorithm. For instance, phase 2 of the KEA algorithm requires a private random value, rA, and a public version, Ra, to be generated. The private value, rA, is added to the context and the public value, Ra, is returned to the caller. In some cases, when both input and output is required, a data structure is passed to the algorithm. In this situation, *Param→Data* references the structure and *Param→Length* is set to the length of the structure.

ADDITIONAL SPI 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.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Modifying this structure has no effect on the internal structure maintained by the CSSM. It is only a copy of the actual data. Changes to the context attributes must be returned using the *UpdatedAttributes* return parameter.

NumberOfUpdatedAttributes (output)
The number of CSSM_CONTEXT_ATTRIBUTE structures contained in the *UpdatedAttributes* array. If this value is zero, *UpdatedAttributes* should be set to NULL.

UpdatedAttributes (output)
An array of attributes that will be added to the context should be returned using this parameter. Memory for the attribute structures should be allocated using the CSSM_UPCALLS callbacks provided to the service provider module when *CSSM_SPI_ModuleAttach*( ) is called.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR API

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.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

7.9 Miscellaneous Functions

The man-page definitions for Miscellaneous CSP functions are presented in this section.

NAME

CSSM_CSP_GetOperationalStatistics for the CSSM API
CSP_GetOperationalStatistics for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CSP_GetOperationalStatistics
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CSP_OPERATIONAL_STATISTICS *Statistics)

SPI:
CSSM_RETURN CSSMCSPI CSSM_CSP_GetOperationalStatistics
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CSP_OPERATIONAL_STATISTICS *Statistics)
```

DESCRIPTION

Obtain the current operational values of a subservice. The information is returned in a structure of type CSSM_CSP_OPERATIONAL_STATISTICS. This information includes login status and available storage space. The data structure to hold the returned results must be provided by the caller. The CSP does not allocate memory on behalf of the caller.

API AND SPI PARAMETERS

CSPHandle (input)
Handle of the cryptographic service provider that will perform the operation.

Statistics (output)
Structure containing the subservice's current statistics.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

NAME

CSSM_GetTimeValue for the CSSM API
CSP_GetTimeValue for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_GetTimeValue
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS TimeAlgorithm,
    CSSM_DATA *TimeData)

SPI:
CSSM_RETURN CSSMCSPI CSP_GetTimeValue
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_ALGORITHMS TimeAlgorithm,
    CSSM_DATA *TimeData)
```

DESCRIPTION

This function returns a time value maintained by a CSP. This feature will be supported primarily by hardware tokens with an onboard real time clock.

API AND SPI PARAMETERS

CSPHandle (input)
Handle of the cryptographic service provider that will perform the operation.

TimeAlgorithm (input)
A CSSM algorithm type that indicates the method for fetching the time. The following algorithm types are currently supported:

CSSM_ALGID_UTC
Returns a time value in the form YYYYMMDDhhmmss (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second). The time returned is GMT.

CSSM_ALGID_RUNNING_COUNTER
The current value of a running hardware counter that operates while the device is in operation. This value can be read from a processor counter provided by some platform architectures.

TimeData (output)
The time value of counter value returned in response to the request.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS

1. 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.

2. Some tokens require authentication before returning a time value.

NAME

CSSM_RetrieveUniqueId for the CSSM API
CSP_RetrieveUniqueId for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_RetrieveUniqueId
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_DATA_PTR UniqueID)

SPI:
CSSM_RETURN CSSMCSPI CSP_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.

API AND SPI 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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

NAME

CSSM_RetrieveCounter for the CSSM API
CSP_RetrieveCounter for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_RetrieveCounter
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_DATA_PTR Counter)

SPI:
CSSM_RETURN CSSMCSPI CSP_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.

API AND SPI 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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

COMMENTS FOR SPI

The output is returned to the caller as specified in Section 7.3.8 on page 130.

NAME

CSSM_VerifyDevice for the CSSM API
CSP_VerifyDevice for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_VerifyDevice
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *DeviceCert)

SPI:
CSSM_RETURN CSSMCSPI CSP_VerifyDevice
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *DeviceCert)
```

DESCRIPTION

This function triggers the cryptographic module to perform self verification and integrity checking.

API AND SPI 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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_DEVICE_VERIFY_FAILED

7.10 Extensibility Function

The *PassThrough* function is provided to allow CSP developers to extend the crypto functionality of CDSA. 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 for the CSSM API
CSP_PassThrough for the CSP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CSP_PassThrough
    (CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void *InData,
    void **OutData)

SPI:
CSSM_RETURN CSSMCSPI CSP_PassThrough
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context,
    uint32 PassThroughId,
    const void *InData,
    void **OutData)
```

DESCRIPTION

The *CSSM_CSP_PassThrough*() (CSSM API) or *CSP_PassThrough*() (CSP SPI) function is provided to allow CSP developers to extend the crypto functionality of the CSSM API.

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.

OutData (output)
A pointer to a module, implementation-specific structure containing the output data. The service provider will allocate the memory for this structure. The application should free the memory for the structure.

ADDITIONAL SPI PARAMETERS

CSPHandle (input)
Handle of the CSP supporting the *PassThrough* function.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this custom context structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CSP_INVALID_PASSTHROUGH_ID

7.11 Module Management Function

The *CSP_EventNotify*( ) function is used by the CSSM Core to interact with the CSP module.

Because this function is only exposed to CSSM as a function pointer, the function name internal to the CSP can be assigned at the discretion of the CSP module developer. However, the parameter list and return value types must match those defined for this function.

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.

NAME

CSP_EventNotify

SYNOPSIS

```
CSSM_RETURN CSSMCSPI CSP_EventNotify
    (CSSM_MODULE_HANDLE CSPHandle,
    CSSM_CONTEXT_EVENT Event,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT *Context)
```

DESCRIPTION

This function is used to notify the service module of a context event related to a particular attach handle. Valid events include creation, deletion, or modification of a cryptographic context. The service module can examine the new or modified context referenced by *pContext* to determine whether the context is acceptable to the service module.

If the cryptographic context is acceptable (of the service module examines the contents of the context only upon use of the context), then the service module should return CSSM_OK. If the cryptographic context is not acceptable, then the service module should return CSSM_FAIL.

Upon receiving a return value of CSSM_OK, CSSM completes the operation signaled by this event and successfully returns to the calling application. If the return value is CSSM_FAIL, CSSM discards a newly created context or modifications to an existing context, and returns the failed result to the calling application. When deleting a cryptographic context, CSSM always returns success to the calling application.

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:

Event	**Description**
CSSM_CONTEXT_EVENT_CREATE	A caller using this module attach handle has created a new cryptographic context via *CSSM_Create<*>Context* .
CSSM_CONTEXT_EVENT_DELETE	A caller using this module attach handle has deleted a cryptographic context via *CSSM_DeleteContext*( ).
CSSM_CONTEXT_EVENT_UPDATE	A caller using this module attach handle has updated an existing cryptographic context.

CCHandle (input)

The cryptographic context handle for the context affected by the *Event*.

Context

A pointer to the cryptographic context affected by the *Event*. The results of the *Event* are visible in the context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

SEE ALSO

CSSM_CSP_CreateSignatureContext( )
CSSM_CSP_CreateSymmetricContext( )
CSSM_CSP_CreateDigestContext( )
CSSM_CSP_CreateMacContext( )
CSSM_CSP_CreateRandomContext( )
CSSM_CSP_CreateAsymmetricContext( )
CSSM_CSP_CreateDeriveKeyContext( )
CSSM_CSP_CreateKeyGenContext( )
CSSM_CSP_CreatePassThroughContext( )
CSSM_DeleteContext( )
CSSM_UpdateContextAttributes( )

Technical Standard

Part 4:

Trust Policy (TP) Services

The Open Group

Chapter 8

Trust Policy Services API

8.1 Overview

8.1.1 Digital Certificate

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

8.1.2 Trust Model

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.

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
- A set of data objects targeted by those actions
- Evidence of the results of the trust evaluation

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. 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 final analysis, the trust and authorizations that are asserted are based on the authority implied by a set of assumed or otherwise-specified public keys.

8.1.3 Trust Services

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.

The CDSA Trust Policy modules include interfaces for two categories of trust services:

- Evaluating trust based on the policies of a specific application domain

  In this case, applications are viewed as executing solely 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.

- Providing access to the services of other authoritative agents that support the life cycle of certificates and certificate revocation lists, upon which trust is based.

  In this case, the Trust Policy interfaces provide access to authorities providing services that support the complete life cycles of certificates and certificate life cycles.

  The certificate life cycle is presented in Figure 8-1.

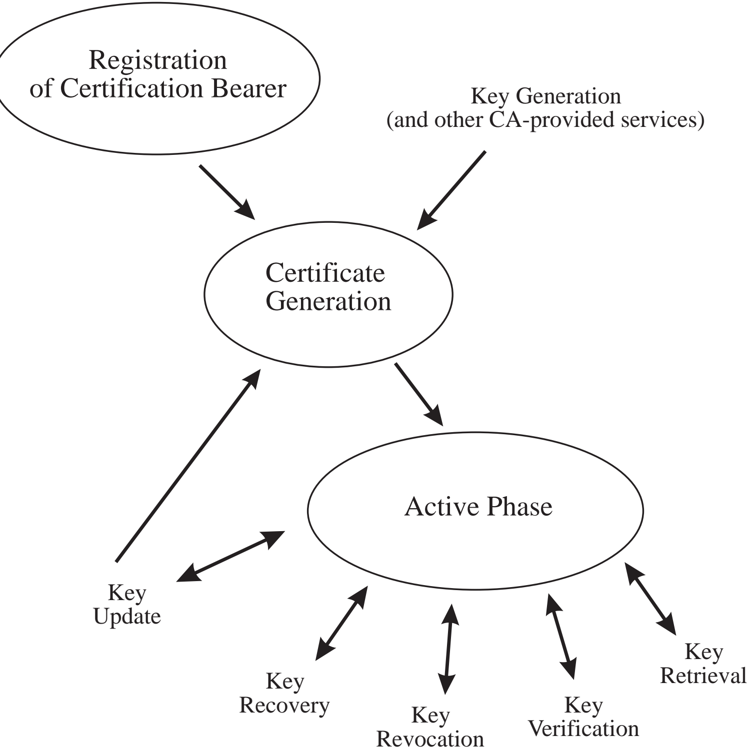


Figure 8-1 Certificate Life Cycle States and Actions

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 back-office 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 private key associated with some certificate for signing or for decryption

— Update—issuing a new public/private key pair when a legitimate pair has or will expire soon.

These services can be provided by a local authority or a remote authority. Requesters must be authorized to request a service and to receive the results of that service. Ultimately access is controlled by the authority's policy. Policy enforcement can be distributed. By distributed enforcement we mean a local client system may enforce portions of the CA's policy prior to submitting request to the remote authority. The local policy statement can be the empty set (if appropriate). The remote authority enforces policy and performs the service. If the requester is authorized and the service succeeds, then results are returned to the requester. To support asynchronous completion of services, submitting a request and retrieving a result are performed using separate interfaces. The function pairs are:

- Authenticated Submit and Authenticated Retrieve
  One function to request a controlled service and one to retrieve the result of that controlled request. The caller is authenticated by the local service provider before any service is performed.

- Authenticated Confirm and Receive
  One function to acknowledge successful retrieval of results and one to receive the confirmation.

- Non-Authenticated Submit and Non-Authenticated Retrieve
  One function to request a public service and one to retrieve the result of that public service request.

The request service may not complete for several days. Applications can submit requests and retrieve results in separate executions of the application.

8.2 CDSA TP Features

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 supports all of the currently provided operations supporting the life cycle of 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 MDS records that describe the TP service provider should include a list of the authorities and life cycle services the TP module can access on behalf of a caller. If the authority is remote, the protocol used to communication with the remote authority is encapsulated in the local Trust Policy Service Provider. The TP service provider can record the implemented protocols in it's associated MDS records. Applications can query MDS records to determine what services are provided by a TP module.

8.3 SPI TP

8.3.1 Add-In Module

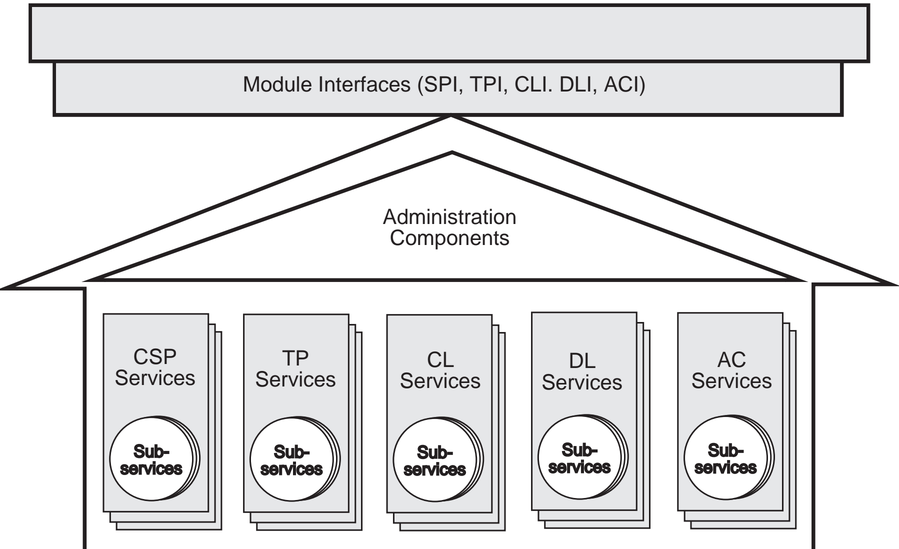


Figure 8-2 CDSA Add-In Module Structure

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. In this case it provides Trust Policy (TP) services.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions that 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 **Part 14** of this Technical Standard.

8.3.2 Operations

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.

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 the function *CSSM_RegisterServices*( )). See **Part 14** (*CSSM Add-in Module Structure and Administration Specification*) for details of module installation and registration.

Many applications are hard-coded to select a specific Trust Policy. The Module Directory Services (MDS) system provides query mechanisms so applications can access 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.

8.4 Data Structures

8.4.1 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 CSSM_MODULE_HANDLE CSSM_TP_HANDLE; /* Trust Policy Handle */
```

8.4.2 CSSM_TP_AUTHORITY_ID

This data structure identifies an Authority who provides security-related services. It is used as input to functions requesting authority services.

```
typedef struct cssm_tp_authority_id {
    CSSM_DATA *AuthorityCert;
    CSSM_NET_ADDRESS_PTR AuthorityLocation;
} CSSM_TP_AUTHORITY_ID, *CSSM_TP_AUTHORITY_ID_PTR;
```

Definitions

AuthorityCert
A pointer to the CSSM_DATA structure containing the desired Authority's certificate. If the *AuthorityCert* is NULL, a service provider module can provide a certificate identifying a default authority.

AuthorityLocation
A pointer to a network address directly or indirectly identifying the location of the authority process. If the input is NULL, a service provider module can determine an authority process and its location based on the *AuthorityCert* input parameter or can assume a default authority location.

8.4.3 CSSM_TP_AUTHORITY_REQUEST_TYPE

This extensible list defines the type of a request to an Authority providing certificate-related services.

```
typedef uint32 CSSM_TP_AUTHORITY_REQUEST_TYPE,
*CSSM_TP_AUTHORITY_REQUEST_TYPE_PTR;

#define CSSM_TP_AUTHORITY_REQUEST_CERTISSUE       (0x01)
#define CSSM_TP_AUTHORITY_REQUEST_CERTREVOKE      (0x02)
#define CSSM_TP_AUTHORITY_REQUEST_CERTSUSPEND     (0x03)
#define CSSM_TP_AUTHORITY_REQUEST_CERTRESUME      (0x04)
#define CSSM_TP_AUTHORITY_REQUEST_CERTVERIFY      (0x05)
#define CSSM_TP_AUTHORITY_REQUEST_CERTNOTARIZE    (0x06)
#define CSSM_TP_AUTHORITY_REQUEST_CERTUSERECOVER  (0x07)

#define  CSSM_TP_AUTHORITY_REQUEST_CRLISSUE       (0x100)
```

8.4.4 CSSM_TP_VERIFICATION_RESULTS_CALLBACK

This type defines the form of the callback function a service provider module must use to incrementally return verified certificates to a caller whose credentials are being verified.

```
typedef CSSM_RETURN (CSSMAPI * CSSM_TP_VERIFICATION_RESULTS_CALLBACK)
    (CSSM_MODULE_HANDLE ModuleHandle,
    void *CallerCtx,
    CSSM_DATA_PTR VerifiedCert);
```

Definitions

ModuleHandle
The CSSM_MODULE_HANDLE for the attach session under which the verification process is being performed and incrementally reported to the caller.

CallerCtx
A generic pointer to context information that was provided by the original requester and is being returned to its originator.

VerifiedCert
A pointer to a CSSM_DATA structure containing the most recently verified certificate in an incremental certificate verification process.

8.4.5 CSSM_TP_POLICYINFO

This data structure contains a set of one or more policy identifiers and application-domain-specific information used to control evaluation of the named policies.

```
typedef struct cssm_tp_policyinfo {
    uint32 NumberOfPolicyIds;
    CSSM_FIELD_PTR PolicyIds;
    void *PolicyControl;
} CSSM_TP_POLICYINFO, *CSSM_TP_POLICYINFO_PTR;
```

Definitions

NumberOfPolicyIds
The number of policy identifiers provided in the *PolicyIds* parameter.

PolicyIds
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 use when applying the policy. The name space for policy identifiers is defined externally by the application domains served by the certification authority module.

PolicyControl
A pointer to provider-specific data to be used when evaluating the policies specified by *PolicyIds*.

8.4.6 CSSM_TP_SERVICES

This bit mask defines the additional back-office services that a Certification Authority (CA) can offer. Such services include (but are not limited to) archiving a certificate and keypair, publishing a 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 may be defined over time.

```
typedef uint32 CSSM_TP_SERVICES;

/* bit masks for additional Authority services available through TP */
#define CSSM_TP_KEY_ARCHIVE (0x0001)       /* archive cert & keys */
#define CSSM_TP_CERT_PUBLISH (0x0002)      /* register cert in
                                                      directory */
#define CSSM_TP_CERT_NOTIFY_RENEW (0x0004) /* notify at renewal time */
#define CSSM_TP_CERT_DIR_UPDATE (0x0008)   /* update cert registry
                                                          entry */
#define CSSM_TP_CRL_DISTRIBUTE (0x0010)    /* push CRL to everyone */
```

8.4.7 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;

#define    CSSM_TP_ACTION_DEFAULT     (0)
```

8.4.8 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
                                                  TRUE or FALSE */
    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;
```

8.4.9 CSSM_TIMESTRING

Values must be expressed in Greenwich Mean Time (Zulu) and must include seconds, even when the value of seconds is "00". Values must not include fractional seconds.

This string contains a date and time in the format "YYYYMMDDhhmmss", defined by:

YYYY - four characters representing the year

MM - two characters representing the month within a year

DD - two characters representing the day within a month

hh - two characters representing the hours within a day ["00" through "23"]

mm - two characters representing the minutes within a hour ["00" through "59"]

ss - two characters representing the seconds within a minute ["00" through "59"]

```
typedef char *CSSM_TIMESTRING;
```

8.4.10 CSSM_TP_CALLERAUTH_CONTEXT

This data structure contains the basic parameters required to authenticate the caller who is requesting a certificate-related service from a certificate authority.

```
typedef struct cssm_tp_callerauth_context {
    CSSM_TP_POLICYINFO Policy;
    CSSM_TIMESTRING VerifyTime;
    CSSM_TP_STOP_ON VerificationAbortOn;
    CSSM_TP_VERIFICATION_RESULTS_CALLBACK CallbackWithVerifiedCert;
    uint32 NumberOfAnchorCerts;
    CSSM_DATA_PTR AnchorCerts;
    CSSM_DL_DB_LIST_PTR DBList;
    CSSM_ACCESS_CREDENTIALS_PTR CallerCredentials;
} CSSM_TP_CALLERAUTH_CONTEXT, *CSSM_TP_CALLERAUTH_CONTEXT_PTR;
```

Definitions

Policy
A structure identifying one or more policies and associated control information. The authentication should be performed under one or more of the specified policies.

VerifyTime
A CSSM_TIMESTRING specifying the point-in-time for which the caller's credentials should be authenticated for validity.

VerificationAbortOn
When multiple conditions or multiple policies are supported, the service provider module can allow the caller to specify when to abort the verification process. If supported, the selected option can affect the verification evidence returned to the caller. The default stopping condition is to stop evaluation according to the implicit policy defined by the service provider. 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 service provider module can ignore the caller's specified stopping condition and revert to the implicit, default stopping policy.

CallbackWithVerifiedCert
A caller defined function to be invoked by the service provider module once for each

certificate examined in the verification process. The verified certificate is passed back to the caller via this function. The reported certificate is represented as an encoded certificate in a CSSM_DATA structure allocated by the service provider. The caller implementing the callback function must free the CSSM_DATA structure and its contents. If the verification process completes in a single verification step, then no callbacks are made from the service provider module to the caller. If the callback function pointer is NULL, no callbacks are performed.

NumberOfAnchorCerts
The number of anchor certificates provided in the *AnchorCerts* parameter.

AnchorCerts
A pointer to an array of the CSSM_DATA structures containing one or more certificates to be used in the process of authenticating a caller of this function. These certificates are in addition to the caller-owned credentials.

DBList
A list of databases containing credentials that can be used in the process of authenticating a caller of this function. The list contains a Data Library Module handle and an open data base handle. The Data Library interfaces must be used to access these data bases. Each data base can contains multiple credential types appropriate to supporting the authentication process, such as certificates and certificate revocation lists.

CallerCredentials
A pointer to the CSSM_ACCESS_CREDENTIALS structure containing one or more credentials the caller requires to authenticate to the service provider. The information is input to certification authority functions that require caller authentication before being serviced. The credentials structure can contain an immediate value for the credential, such as a passphrase or the caller can specify a callback function the CSP can use to carry out a credential acquisition protocol with the caller to obtain one or more credentials. The supported protocols are recorded in the MDS entry for the service provider module.

8.4.11 CSSM_CRL_PARSE_FORMAT

This extensible list defines the parse formats for a CRL.

```
typedef uint32 CSSM_CRL_PARSE_FORMAT, * CSSM_CRL_PARSE_FORMAT_PTR;

#define CSSM_CRL_PARSE_FORMAT_NONE (0x00)
#define CSSM_CRL_PARSE_FORMAT_CUSTOM (0x01)
#define CSSM_CRL_PARSE_FORMAT_SEXPR (0x02)
#define CSSM_CRL_PARSE_FORMAT_COMPLEX (0x03)
#define CSSM_CRL_PARSE_FORMAT_OID_NAMED (0x04)
#define CSSM_CRL_PARSE_FORMAT_TUPLE (0x05)
#define CSSM_CRL_PARSE_FORMAT_MULTIPLE (0x7FFE)
#define CSSM_CRL_PARSE_FORMAT_LAST (0x7FFF)

/* Applications wishing to define their own custom parse
 * format should create a uint32 value greater than the
 * CSSM_CL_CUSTOM_CRL_PARSE_FORMAT */
#define CSSM_CL_CUSTOM_CRL_PARSE_FORMAT (0x8000)
```

8.4.12 CSSM_PARSED_CRL

This structure holds a parsed representation of a CRL. The CRL type and representation format are included in the structure.

```
typedef struct cssm_parsed_crl {
    CSSM_CRL_TYPE CrlType; /* CRL type */
    CSSM_CRL_PARSE_FORMAT ParsedCrlFormat; /* struct of ParsedCrl */
    void *ParsedCrl; /* parsed CRL (to be typecast) */
} CSSM_PARSED_CRL, *CSSM_PARSED_CRL_PTR ;
```

Definitions

CrlType
Indicates the type of CRL that had been parsed to yield ParsedCrl.

ParsedCrlFormat
Indicates the structure and format representation of the parsed CRL. If the parsed representation is not available, then this value is CSSM_CRL_PARSE_FORMAT_NONE.

ParsedCrl
A pointer to a parsed CRL represented in the structure and format indicated by *ParsedCrlFormat*.

8.4.13 CSSM_CRL_PAIR

This structure holds a parsed representation and an encoded representation of a CRL. The two elements should be different representations of a single CRL.

```
typedef struct cssm_crl_pair {
    CSSM_ENCODED_CRL EncodedCrl; /* an encoded CRL blob */
    CSSM_PARSED_CRL ParsedCrl; /* equivalent parsed CRL */
} CSSM_CRL_PAIR, *CSSM_CRL_PAIR_PTR;
```

Definitions

EncodedCrl
A CSSM_ENCODED_CRL structure containing:

- A reference to an opaque, single byte-array representation of the CRL
- A CRL type descriptor
- A CRL encoding descriptor.

The CRL can have an equivalent parsed representation. If the parsed representation is provided it is contained in *ParsedCrl*.

ParsedCrl
A CSSM_PARSED_CRL structure containing:

- A CRL type
- A reference to a parsed representation of the CRL
- A parse format descriptor

The CRL can have an equivalent encoded representation. If the encoded representation is provided it is contained in *EncodedCrl*.

8.4.14 CSSM_CRLGROUP_TYPE

This extensible list defines the type of a CRL group. A group can contain a single type of CRL or multiple types of CRLs. Each CRL in the group can be represented in an encoded representation or a parsed representation.

```
typedef uint32 CSSM_CRLGROUP_TYPE,  * CSSM_CRLGROUP_TYPE_PTR;

#define CSSM_CRLGROUP_DATA         (0x00)
#define CSSM_CRLGROUP_ENCODED_CRL  (0x01)
#define CSSM_CRLGROUP_PARSED_CRL   (0x02)
#define CSSM_CRLGROUP_CRL_PAIR     (0x03)
```

8.4.15 CSSM_CRLGROUP

This data structure aggregates a group of one or more memory-resident CRLs. The CRLs can be of one type or of mixed types and encodings.

```
typedef struct cssm_crlgroup {
    CSSM_CRL_TYPE CrlType;
    CSSM_CRL_ENCODING CrlEncoding;
    uint32 NumberOfCrls;
    union {
        CSSM_DATA_PTR  CrlList ;                /* CRL blob */
        CSSM_ENCODED_CRL_PTR  EncodedCrlList ;  /* CRL blob
                                                   w/ separate type */
        CSSM_PARSED_CRL_PTR ParsedCrlList;      /* bushy, parsed CRL */
        CSSM_CRL_PAIR_PTR PairCrlList;
    } GroupCrlList;
    CSSM_CRLGROUP_TYPE CrlGroupType;
} CSSM_CRLGROUP, *CSSM_CRLGROUP_PTR;
```

Definitions

CrlType
If all CRLs in the CrlList are of the same type, this variable lists that type. Otherwise, the type should be CSSM_CRL_TYPE_MULTIPLE.

CrlEncoding
If all CRLs in the *CrlList* are of the same encoding, this variable gives that encoding. Otherwise, the type should be CSSM_CRL_ENCODING_MULTIPLE.

NumberOfCrls
The number of entries in the *CrlList* array.

GroupList

An array of CRLs. The array contains exactly *NumberOfCrls* entries. *CrlGroupType* defines the type of structure contained in the array. The group types are described as follows:

CrlGroupType Value	Field Name	Description
CSSM_CRLGROUP_DATA	CrlList	(Legacy) A pointer to an array of CSSM_DATA structures. Each CrlList array entry references a single CRL structure and indicates the length of the structure. A single type and encoding apply to all CRLs in this group. The type and encoding are indicated in CrlType and CrlEncoding respectively.
CSSM_CRLGROUP_ENCODED_CRL	EncodedCrlList	A pointer to an array of CSSM_ENCODED_CRL structures. Each EncodedCrlList array entry references a CRL in an opaque, single byte-array representation, and describes the format of the CRL data contained in the byte-array. Each CRL encoding and type can be distinct, as indicated in each array element.
CSSM_CRLGROUP_PARSED_CRL	ParsedCrlList	A pointer to an array of CSSM_PARSED_CRL structures. Each ParsedCrlList array entry references a CRL in a parsed representation, and indicates the CRL type and parse format of that CRL.
CSSM_CRLGROUP_CRL_PAIR	PairCrlList	A pointer to an array of CSSM_CRL_PAIR structures. Each PairCrlList array entry aggregates two CRL representations: an opaque encoded CRL blob, and a parsed CRL representation. At least one of the two representations must be present in each array entry. If both are present, they are assumed but not guaranteed to correspond to one another. If the parsed form is being used in a security sensitive operation, then it must have been verified against the packed, encoded form, whose signature must have been verified.

CrlGroupType - The type of the group.

8.4.16 CSSM_FIELDGROUP

This data structure groups a set of OID/value pairs into a single structure with a count of the number of fields in the array.

```
typedef struct cssm_fieldgroup {
    int NumberOfFields ;      /* number of fields in the array */
    CSSM_FIELD_PTR Fields ;   /* array of fields */
} CSSM_FIELDGROUP, *CSSM_FIELDGROUP_PTR ;
```

Definitions

NumberOfFields
The number of entries in the array Fields.

Fields
A pointer to an ordered array of OID/value pairs. Each structure contains a single pair.

8.4.17 CSSM_EVIDENCE_FORM

This type defines constants corresponding to known representations of verification evidence returned by a verification function or service. The constant values are used as type indicators in the structure CSSM_EVIDENCE.

```
typedef uint32 CSSM_EVIDENCE_FORM;

#define CSSM_EVIDENCE_FORM_UNSPECIFIC    0x0
#define CSSM_EVIDENCE_FORM_CERT          0x1
#define CSSM_EVIDENCE_FORM_CRL           0x2
#define CSSM_EVIDENCE_FORM_CERT_ID       0x3
#define CSSM_EVIDENCE_FORM_CRL_ID        0x4
#define CSSM_EVIDENCE_FORM_VERIFIER_TIME 0x5
#define CSSM_EVIDENCE_FORM_CRL_THISTIME  0x6
#define CSSM_EVIDENCE_FORM_CRL_NEXTTIME  0x7
#define CSSM_EVIDENCE_FORM_POLICYINFO    0x8
#define CSSM_EVIDENCE_FORM_TUPLEGROUP    0x9
```

8.4.18 CSSM_EVIDENCE

This structure contains certificates, CRLs and other information used as audit trail evidence.

```
typedef struct cssm_evidence {
    CSSM_EVIDENCE_FORM EvidenceForm;
    void *Evidence;            /* Evidence content */
} CSSM_EVIDENCE, *CSSM_EVIDENCE_PTR;
```

Definitions

EvidenceForm
An identifier indicating the type and the data structure of the evidence.

Evidence
Buffer containing audit trail information.

The correspondence between the value of *EvidenceForm* and the structure type contained in the *Evidence* field is defined as follows:

Value of *EvidenceForm*	Data Type of *Evidence*
CSSM_EVIDENCE_FORM_UNSPECIFIC	void *
CSSM_EVIDENCE_FORM_CERT	CSSM_ENCODED_CERT
CSSM_EVIDENCE_FORM_CRL	CSSM_ENCODED_CRL
CSSM_EVIDENCE_FORM_CERT_ID	Big-endian integer
CSSM_EVIDENCE_FORM_CRL_ID	Big-endian integer
CSSM_EVIDENCE_FORM_VERIFIER_TIME	CSSM_TIMESTRING
CSSM_EVIDENCE_FORM_CRL_THISTIME	CSSM_TIMESTRING
CSSM_EVIDENCE_FORM_CRL_NEXTTIME	CSSM_TIMESTRING
CSSM_EVIDENCE_FORM_TUPLEGROUP	CSSM_TUPLEGROUP
CSSM_EVIDENCE_FORM_POLICYINFO	CSSM_TP_POLICYINFO

8.4.19 CSSM_TP_VERIFY_CONTEXT

This data structure contains all of the credentials and information required as input to a verification service. The verification service verifies a set of credentials for fitness to perform an application-defined action on a specified data object.

```
typedef struct cssm_tp_verify_context {
CSSM_TP_ACTION Action;
    CSSM_DATA ActionData;
    CSSM_CRLGROUP Crls;
CSSM_TP_CALLERAUTH_CONTEXT_PTR Cred;
} CSSM_TP_VERIFY_CONTEXT, *CSSM_TP_VERIFY_CONTEXT_PTR;
```

Definitions

Action
An application-specific and application-defined action to be performed under the authority of the input credentials. If no action is specified, the service provider module can define a default action and performs verification assuming that action is being requested.

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 a target data object, then the service provider module defines one or more default data objects upon which the action or default action would be performed.

Crls
A group of one or more memory-resident Certificate Revocation Lists (CRLs) to be used in support of the verification process. Persistent CRLs managed by a Data Storage Library module are referenced in *Cred.DbList*.

Cred
A pointer to a CSSM_TP_CALLERAUTH_CONTEXT structure containing one or more credentials to be verified for fitness to perform the requested action on the target data object. The structure also contains control information to guide the credential verification process.

8.4.20 CSSM_TP_VERIFY_CONTEXT_RESULT

This data structure contains the results of an evaluation by a local trust policy. The trust policy evaluation process can return to the requester the following:

- description of the how the trust policy was applied during the evaluation process
- A set of evidence compiled during the evaluation process.

```
typedef struct cssm_tp_verify_context_result {
    uint32 NumberOfEvidences;
    CSSM_EVIDENCE_PTR Evidence;
} CSSM_TP_VERIFY_CONTEXT_RESULT, *CSSM_TP_VERIFY_CONTEXT_RESULT_PTR;
```

Definitions

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.

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.

8.4.21 CSSM_TP_REQUEST_SET

This data structure specifies the input data required when requesting one or more requests from a Certification Authority. The initialized structure is input to *CSSM_TP_SubmitCredRequest*().

```
typedef struct cssm_tp_request_set {
    uint32 NumberOfRequests;
    void *Requests;
} CSSM_TP_REQUEST_SET, *CSSM_TP_REQUEST_SET_PTR;
```

Definitions

NumberOfRequests
The number of entries in the array *Requests*.

Requests
A pointer to an ordered array of service-specific certificate request structures. Each structure contains a single request.

8.4.22 CSSM_TP_RESULT_SET

This data structure specifies the final output data returned by a call to *CSSM_TP_RetrieveCredAuthResult*(). The structure contains an ordered array. Each array entry corresponds to one request.

A call to *CSSM_TP_RetrieveCredAuthResult*() must have a corresponding call to the request function *CSSM_TP_SubmitCredRequest*().

```
typedef struct cssm_tp_result_set {
    uint32 NumberOfResults;
    void *Results;
```

```
} CSSM_TP_RESULT_SET, *CSSM_TP_RESULT_SET_PTR;
```

Definitions

NumberOfResults
The number of certificate service results contained in *Results*.

Results
A pointer to an ordered array of certificate service output structures. The array is ordered corresponding to the array of input requests.

8.4.23 CSSM_TP_CONFIRM_STATUS

This type defines constants used to indicate acceptance or rejection of the results of a service authority.

```
typedef uint32 CSSM_TP_CONFIRM_STATUS, *CSSM_TP_CONFIRM_STATUS_PTR;

#define CSSM_TP_CONFIRM_STATUS_UNKNOWN 0x0
    /* indeterminate */
#define CSSM_TP_CONFIRM_ACCEPT 0x1
    /* accept results of executing a submit-retrieve function pair */
#define CSSM_TP_CONFIRM_REJECT 0x2
    /* reject results of executing a submit-retrieve function pair */
```

8.4.24 CSSM_TP_CONFIRM_RESPONSE

This data structure specifies the input response vector required when confirming the results returned by a Certification Authority through the *CSSM_TP_RetrieveCredResult*() function. The response vector is ordered corresponding to the set of returned results, one response per result. The valid response values include:

- CSSM_TP_CONFIRM_ACCEPT - indicates the corresponding result was accepted by the requester
- CSSM_TP_CONFIRM_REJECT - indicates the corresponding result was rejected by the requester
- CSSM_TP_CONFIRM_STATUS_UNKNOWN - indicates that the requester can not accept or reject the returned result. The status of the returned result in indeterminate

```
typedef struct cssm_tp_confirm_response {
    uint32 NumberOfResponses;
    CSSM_TP_CONFIRM_STATUS_PTR Responses;
} CSSM_TP_CONFIRM_RESPONSE, *CSSM_TP_CONFIRM_RESPONSE_PTR;
```

Definitions

NumberOfResponses
The number of entries in the vector Responses.

Responses
A pointer to an ordered vector of confirmation responses. Each response corresponds to a result returned by a certificate-service authority.

8.4.25 CSSM_ESTIMATED_TIME_UNKNOWN

The value used by an authority or process to indicate that an estimated completion time can not be determined.

```
#define CSSM_ESTIMATED_TIME_UNKNOWN   -1
```

8.4.26 CSSM_ELAPSED_TIME_UNKNOWN

A constant value indicating that the service provider module does not know the time elapsed since a specific prior event.

```
#define CSSM_ELAPSED_TIME_UNKNOWN    (-1)
```

8.4.27 CSSM_ELAPSED_TIME_COMPLETE

A constant value indicating that request has completed and the elapsed time since a specific prior event is no longer of significance.

```
#define CSSM_ELAPSED_TIME_COMPLETE   (-2)
```

8.4.28 CSSM_TP_CERTISSUE_INPUT

This data structure aggregates the input information for a request to issue a single new certificate.

```
typedef struct cssm_tp_certissue_input {
    CSSM_SUBSERVICE_UID CSPSubserviceUid;
    CSSM_CL_HANDLE CLHandle;
    uint32 NumberOfTemplateFields;
    CSSM_FIELD_PTR SubjectCertFields;
    CSSM_TP_SERVICES MoreServiceRequests;
    uint32 NumberOfServiceControls;
    CSSM_FIELD_PTR ServiceControls;
    CSSM_ACCESS_CREDENTIALS_PTR UserCredentials;
} CSSM_TP_CERTISSUE_INPUT, *CSSM_TP_CERTISSUE_INPUT_PTR;
```

Definitions

CSPSubserviceUid
The identifier that uniquely describes the CSP module subservice where the private key associated with the newly issued certificate is to be stored. Optionally, an immediate service provider module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

CLHandle
The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate a certificate of the requested type on the local system. This is an optional input parameter for this request type.

NumberOfTemplateFields
The number of entries in the *SubjectCertFields* array.

SubjectCertFields
An array of OID/value pairs providing input values to the TP. The TP can include these values in the new certificate or can use the values to support the creation process without explicitly including them in the new certificate. A count of the number of OID/value pairs is

included in the structure.

MoreServiceRequests
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, etc.

NumberOfServiceControls
The number of entries in the *ServiceControls* array.

ServiceControls
A pointer to an array of CSSM_FIELD structures. Each array element contains:

- A CSSM_OID structure - the *Oid* value identifies one additional TP service associated with a primary TP service request. The name space for OID values is defined by the TP service domain.
- A CSSM_DATA structure - the *Data* value is input information to the additional TP service identified by the *Oid* value.

UserCredentials
A pointer to the set of one or more credentials being presented for authentication by the caller making the certificate request. The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function a service provider module can use to obtain one or more credentials. The *callback* function can be used to obtain a passphrase, a biometric input, or to perform a challenge and response protocol. The type of credentials accepted by a service provider module is defined and recorded in a record in the Module Directory Service records describing that provider. If the service provider module does not require credentials from a caller, then this field can be NULL.

8.4.29 CSSM_TP_CERTISSUE_STATUS

This set of defined constants indicates the status of a service request to issue a certificate.

```
typedef uint32 CSSM_TP_CERTISSUE_STATUS

#define CSSM_TP_CERTISSUE_STATUS_UNKNOWN 0x0
    /* indeterminate */

#define CSSM_TP_CERTISSUE_OK 0x1
    /* cert issued as requested */

#define CSSM_TP_CERTISSUE_OKWITHCERTMODS 0x2
    /* cert issued but cert contents were updated by the
       issuing authority */

#define CSSM_TP_CERTISSUE_OKWITHSERVICEMODS 0x3
    /* cert issued but some requested backend services were
       not performed by the issuing authority */

#define CSSM_TP_CERTISSUE_REJECTED 0x4
    /* cert was not issued due to some error condition */
```

```
#define CSSM_TP_CERTISSUE_NOT_AUTHORIZED 0x5
    /* cert was not issued, the request was not authorized */

#define CSSM_TP_CERTISSUE_WILL_BE_REVOKED 0x6
    /* cert was issued, but TP has initiated a revocation
       of the certificate */
```

8.4.30 CSSM_TP_CERTISSUE_OUTPUT

This data structure aggregates the output information generated in response to a single request to issue a new certificate. Each result includes a certificate group composed of one or more certificates. The first certificate in the group is the newly issued certificate.

```
typedef struct cssm_tp_certissue_output {
    CSSM_TP_CERTISSUE_STATUS IssueStatus;
    CSSM_CERTGROUP_PTR CertGroup;
    CSSM_TP_SERVICES PerformedServiceRequests;
} CSSM_TP_CERTISSUE_OUTPUT, *CSSM_TP_CERTISSUE_OUTPUT_PTR;
```

Definitions

IssueStatus
A value indicating the status of the newly issued certificate.

CertGroup
A pointer to a structure containing a reference to a certificate group and the number of certificates contained in that group. The certificate group contains the newly issued certificate created in response to a request. The certificate group can also contain supporting certificates related to the newly issued certificate. By convention the new certificate is the first member of the certificate group.

PerformedServiceRequests
A bit mask indicating the additional certificate-creation-related services that were performed by the Certificate Authority while issuing the certificate. Possible services include:

- Backup or archive of the private key associated with the newly issued certificate
- Publication of the new certificate in a certificate directory service
- Registration for out-of-band certificate renewal notification
- Other authority-provided services

8.4.31 CSSM_TP_CERTCHANGE_ACTION

This type defines the constants identifying the certificates state changes that are posted to a certificate revocation list for distribution to all concerned systems.

```
typedef uint32 CSSM_TP_CERTCHANGE_ACTION;

#define CSSM_TP_CERTCHANGE_NONE    (0x0) /* no change */

#define CSSM_TP_CERTCHANGE_REVOKE  (0x1) /* Revoke the certificate */
    /* This action type indicates a request to revoke a single
    certificate. Notice of the revocation operation remains
    in affect until the certificate itself expires. Revocation
    should be used to permanently remove a certificate from use. */
```

```
#define CSSM_TP_CERTCHANGE_HOLD    (0x2) /* Hold/suspend the
                                            certificate */
    /* This action type indicates a request to suspend a
    single certificate. A suspension operation implies
    that the requester intends, at some time in the future,
    to request that the certificate be released from hold,
    making it available for use again. Placing a hold on
    a certificate does not obligate the requester to
    request a release. In practice, a certificate may
    remain on hold until the certificate itself expires.
    Revocation should be used to permanently remove a
    certificate from use. */

#define CSSM_TP_CERTCHANGE_RELEASE (0x3) /* Release the held
                                            certificate */
    /* This action type indicates a request to release a
    single certificate currently on hold. A release
    operation makes a certificate available for use again.
    Revocation should be used to permanently remove a
    certificate from use. */
```

8.4.32 CSSM_TP_CERTCHANGE_REASON

This type defines the constants reasons for a change in certificate state.

```
typedef uint32 CSSM_TP_CERTCHANGE_REASON;

#define CSSM_TP_CERTCHANGE_REASON_UNKNOWN            (0x0)
    /* unspecified */

#define CSSM_TP_CERTCHANGE_REASON_KEYCOMPROMISE      (0x1)
    /* Subject key believed to be compromised */

#define CSSM_TP_CERTCHANGE_REASON_CACOMPROMISE       (0x2)
    /* CA's key believed to be compromised */

#define CSSM_TP_CERTCHANGE_REASON_CEASEOPERATION     (0x3)
    /*certificate holder ceases operation under the
    jurisdiction of this certificate */

#define CSSM_TP_CERTCHANGE_REASON_AFFILIATIONCHANGE  (0x4)
    /*certificate holder has moved from this jurisdiction */

#define CSSM_TP_CERTCHANGE_REASON_SUPERCEDED         (0x5)
    /*certificate holder as issued a new, superceding
    certificate */

#define CSSM_TP_CERTCHANGE_REASON_SUSPECTEDCOMPROMISE (0x6)
    /*certificate could be compromised */

#define CSSM_TP_CERTCHANGE_REASON_HOLDRELEASE        (0x7)
    /*certificate holder resumes operation under
    the jurisdiction of this certificate */
```

8.4.33 CSSM_TP_CERTCHANGE_INPUT

This data structure aggregates the input information for a request to change the state of a single certificate.

```
typedef struct cssm_tp_certchange_input{
    CSSM_TP_CERTCHANGE_ACTION Action;
    CSSM_TP_CERTCHANGE_REASON Reason;
    CSSM_CL_HANDLE CLHandle;
    CSSM_DATA_PTR Cert;
    CSSM_FIELD_PTR ChangeInfo;
    CSSM_TIMESTRING StartTime;
    CSSM_ACCESS_CREDENTIALS_PTR CallerCredentials;
} CSSM_TP_CERTCHANGE_INPUT, *CSSM_TP_CERTCHANGE_INPUT_PTR;
```

Definitions

Action
: An identifier indicating a state change that is posted to a certificate revocation list for distribution to all concerned systems.

Reason
: An identifier indicating why the certificate state should be changed.

CLHandle
: The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate a certificate of the requested type on the local system. This is an optional input parameter for this request type.

Cert
: The certificate that is the target of the status change.

ChangeInfo
: A pointer to a CSSM_FIELD structure containing optional information for inclusion in an authority's record regarding the status of *Cert*. The structure contains an OID and a value. The OID identifies the contents of the value field.

StartTime
: The time at which the external event that created the need for a certificate state change took place.

CallerCredentials
: A set of credentials that can be submitted to the certificate authority process for verification when processing the status change request. The supplied credentials must be transportable, because the certificate authority process can be remote. Verification determines whether the requester is authorized to submit the revocation request. If no credentials are required by the certificate authority, then this value can be NULL.

8.4.34 CSSM_TP_CERTCHANGE_STATUS

These constants define the change values that can be returned by a certificate authority that provides state change services for certificates.

```
typedef uint32 CSSM_TP_CERTCHANGE_STATUS;

#define CSSM_TP_CERTCHANGE_STATUS_UNKNOWN (0x0)
    /* indeterminate */

#define CSSM_TP_CERTCHANGE_OK             (0x1)
    /* cert state was successfully changed
    beginning at the specified time */

#define CSSM_TP_CERTCHANGE_OKWITHNEWTIME  (0x2)
    /* cert state was successfully changed, at
    a modified effective time */

#define CSSM_TP_CERTCHANGE_WRONGCA        (0x3)
    /* cert state was not changed, the selected
    CA is not authorized to change the cert state */

#define CSSM_TP_CERTCHANGE_REJECTED       (0x4)
    /* cert state was not changed due to some
    error condition */

#define CSSM_TP_CERTCHANGE_NOT_AUTHORIZED (0x5)
    /* cert state was not changed, the requester is
    not authorized to change the cert state */
```

8.4.35 CSSM_TP_CERTCHANGE_OUTPUT

This type defines the output results returned by a certificate authority in response to an action request to revoke, hold, or release a hold on a certificate. An array of output values is aggregated in a CSSM_TP_RESULT_SET structure and is an output parameter of the function *CSSM_TP_RetrieveCredResult*().

```
typedef struct cssm_tp_certchange_output {
    CSSM_TP_CERTCHANGE_STATUS ActionStatus;
    CSSM_FIELD RevokeInfo;
} CSSM_TP_CERTCHANGE_OUTPUT, *CSSM_TP_CERTCHANGE_OUTPUT_PTR;
```

Definitions

ActionStatus
A CSSM_TP_CERTCHANGE_STATUS value indicating success or specifying a particular error condition.

RevokeInfo
A pointer to a CSSM_FIELD structure containing optional suspension information provided by the Certification Authority to the requester.

8.4.36 CSSM_TP_CERTVERIFY_INPUT

This data structure aggregates the input information for a request to verify a single target certificate.

```
typedef struct cssm_tp_certverify_input{
    CSSM_CL_HANDLE CLHandle;
    CSSM_DATA_PTR Cert;
    CSSM_TP_VERIFY_CONTEXT_PTR VerifyContext;
} CSSM_TP_CERTVERIFY_INPUT, *CSSM_TP_CERTVERIFY_INPUT_PTR;
```

Definitions

CLHandle
The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate a certificate of the requested type on the local system. This is an optional input parameter for this request type.

Cert
The certificate that is the target of a verification operation.

VerifyContext
A pointer to a CSSM_TP_VERIFY_CONTEXT structure containing additional credentials and information to be used in the verification process carried out by the verification authority. The verification request can include:

- An Action - an indicator of the action for which verification is being performed
- An ActionData Object- the data object that is the target of the action

8.4.37 CSSM_TP_CERTVERIFY_STATUS

These constants define the status values that can be returned by a certificate authority that provides a credential verification service. The status CSSM_TP_CERTVERIFY_REVOKED indicates that the certificate in question is explicitly revoked by a revocation record. A certificate can be invalid for one of several reasons, including expiration, unrecognized certificate issuer, invalid signature (implying tampering), unrecognized certificate contents, etc.

```
typedef uint32 CSSM_TP_CERTVERIFY_STATUS

#define CSSM_TP_CERTVERIFY_UNKNOWN                    (0x0)
#define CSSM_TP_CERTVERIFY_VALID                      (0x1)
#define CSSM_TP_CERTVERIFY_INVALID                    (0x2)
#define CSSM_TP_CERTVERIFY_REVOKED                    (0x3)
#define CSSM_TP_CERTVERIFY_SUSPENDED                  (0x4)
#define CSSM_TP_CERTVERIFY_EXPIRED                    (0x5)
#define CSSM_TP_CERTVERIFY_NOT_VALID_YET              (0x6)
#define CSSM_TP_CERTVERIFY_INVALID_AUTHORITY          (0x7)
#define CSSM_TP_CERTVERIFY_INVALID_SIGNATURE          (0x8)
#define CSSM_TP_CERTVERIFY_INVALID_CERT_VALUE         (0x9)
#define CSSM_TP_CERTVERIFY_INVALID_CERTGROUP          (0xA)
#define CSSM_TP_CERTVERIFY_INVALID_POLICY             (0xB)
#define CSSM_TP_CERTVERIFY_INVALID_POLICY_IDS         (0xC)
#define CSSM_TP_CERTVERIFY_INVALID_BASIC_CONSTRAINTS (0xD)
#define CSSM_TP_CERTVERIFY_INVALID_CRL_DIST_PT        (0xE)
#define CSSM_TP_CERTVERIFY_INVALID_NAME_TREE          (0xF)
```

```
#define CSSM_TP_CERTVERIFY_UNKNOWN_CRITICAL_EXT      (0x10)
```

8.4.38 CSSM_TP_CERTVERIFY_OUTPUT

This data structure aggregates the verification status of a single target certificate.

```
typedef struct cssm_tp_certverify_output {
    CSSM_TP_CERTVERIFY_STATUS VerifyStatus;
    uint32 NumberOfEvidence;
    CSSM_EVIDENCE_PTR Evidence;
} CSSM_TP_CERTVERIFY_OUTPUT, *CSSM_TP_CERTVERIFY_OUTPUT_PTR;
```

Definitions

VerifyStatus
The status result of a certificate verification process.

NumberOfEvidence
The number of entries in *Evidence*

Evidence
A pointer to a CSSM_EVIDENCE structure containing an audit trail and conditional information supporting the *VerifyStatus* of a certificate. This can include:

- Certificates (or the unique identifiers for certificates) used as the basis of verification
- CRLs (or the unique names for a CRL) searched during the verification process
- Conditional information such as the timeframe for which the *VerifyStatus* applies Other domain-specific conditions

8.4.39 CSSM_TP_CERTNOTARIZE_INPUT

This data structure aggregates the input information for a request to notarize a certificate and optionally add more field values to the certificate prior to over-signing by a CA process.

```
typedef struct cssm_tp_certnotarize_input {
    CSSM_CL_HANDLE CLHandle;
    uint32 NumberOfFields;
    CSSM_FIELD_PTR MoreFields;
    CSSM_FIELD_PTR SignScope;
    uint32 ScopeSize;
    CSSM_TP_SERVICES MoreServiceRequests;
    uint32 NumberOfServiceControls;
    CSSM_FIELD_PTR ServiceControls;
    CSSM_ACCESS_CREDENTIALS_PTR UserCredentials;
} CSSM_TP_CERTNOTARIZE_INPUT, *CSSM_TP_CERTNOTARIZE_INPUT_PTR;
```

Definitions

CLHandle
The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate a certificate of the requested type on the local system. This is an optional input parameter for this request type.

NumberOfFields
The number of certificate field values specified in *MoreFields*.

MoreFields
A pointer to an array of OID/value pairs providing new, additional certificate field values that can be added to an existing certificate prior to over-signing the existing certificate and the newly added fields. The specified fields can not replace any existing certificate fields. Only appended field values are permitted. These fields are optional.

SignScope
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 CA assumes and includes a default set of certificate fields in the signing process.

ScopeSize
The number of entries in the sign scope list. If no signing scope is specified, then *ScopeSize* must be zero.

MoreServiceRequests
A bit mask requesting additional certificate-notary-related services from the Certificate Authority performing the operation.

NumberOfServiceControls
The number of entries in the *ServiceControls* array.

ServiceControls
A pointer to an array of CSSM_FIELD structures. Each array element contains:

- A CSSM_OID structure - the *Oid* value identifies one additional CA service associated with the notary request. The name space for OID values is defined by the CA service domain.
- A CSSM_DATA structure - the *Data* value is input information to the additional CA service identified by the *Oid* value.

UserCredentials
A pointer to the set of one or more credentials being presented for authentication by the caller making the certificate request. The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function a service provider module can use to obtain one or more credentials. The callback function can be used to obtain a passphrase, a biometric input, or to perform a challenge and response protocol. The type of credentials accepted by a service provider module is defined and recorded in a record in the Module Directory Service records describing that provider. If the service provider module does not require credentials from a caller, then this field can be NULL.

8.4.40 CSSM_TP_CERTNOTARIZE_STATUS

This set of defined constants indicates the status of a service request to notarize an existing certificate by over-signing the certificate and one or more of its current signatures.

typedef uint32 CSSM_TP_CERTNOTARIZE_STATUS;

```
#define CSSM_TP_CERTNOTARIZE_STATUS_UNKNOWN    (0x0)
   /* indeterminate */

#define CSSM_TP_CERTNOTARIZE_OK             (0x1)
   /* cert fields were added and the result was notarized
   as requested */

#define CSSM_TP_CERTNOTARIZE_OKWITHOUTFIELDS   (0x2)
   /* non-conflicting cert fields were added,
    conflicting cert fields were ignored,
    and the result was notarized as requested */

#define CSSM_TP_CERTNOTARIZE_OKWITHSERVICEMODS (0x3)
   /* cert fields were added and the result was notarized
   as requested, but some requested backend services
   were not performed by the notary */

#define CSSM_TP_CERTNOTARIZE_REJECTED        (0x4)
   /* cert was not notarized due to some error condition */

#define CSSM_TP_CERTNOTARIZE_NOT_AUTHORIZED    (0x5)
   /* cert was not notarized, the request was not authorized */
```

8.4.41 CSSM_TP_CERTNOTARIZE_OUTPUT

This data structure aggregates the output information generated in response to a single request to notarize an existing certificate. Each result includes a certificate group composed of one or more certificates. The first certificate in the group is the notarized certificate.

```
typedef struct cssm_tp_certnotarize_output {
    CSSM_TP_CERTNOTARIZE_STATUS NotarizeStatus;
    CSSM_CERTGROUP_PTR NotarizedCertGroup;
    CSSM_TP_SERVICES PerformedServiceRequests;
} CSSM_TP_CERTNOTARIZE_OUTPUT, *CSSM_TP_CERTNOTARIZE_OUTPUT_PTR;
```

Definitions

NotarizeStatus
A CSSM_TP_CERTNOTARIZE_STATUS value indicating the status of the newly notarized certificate.

NotarizedCertGroup
A pointer to a structure containing a reference to a certificate group and the number of certificates contained in that group. The certificate group contains the notarized certificate created in response to a notary request. The certificate group can also contain supporting certificates related to the newly notarized certificate. By convention the new notarized certificate is the first member of the certificate group.

PerformedServiceRequests
A bit mask indicating the additional certificate-notary-related services that were performed by the Certificate Authority while notarizing the certificate. Possible services include:

- Publication of the notarized certificate in a certificate directory service
- Auditing of the notary operation
- Other authority-provided services

8.4.42 CSSM_TP_CERTRECLAIM_INPUT

This data structure aggregates the input information for a request to reclaim an existing certificate and recover its associated private key.

```
typedef struct cssm_tp_certreclaim_input {
    CSSM_CL_HANDLE CLHandle;
    uint32 NumberOfSelectionFields;
    CSSM_FIELD_PTR SelectionFields;
    CSSM_ACCESS_CREDENTIALS_PTR UserCredentials;
} CSSM_TP_CERTRECLAIM_INPUT, *CSSM_TP_CERTRECLAIM_INPUT_PTR;
```

Definitions

CLHandle
The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate a certificate of the requested type on the local system. This is an optional input parameter for this request type.

NumberOfSelectionFields
The number of certificate field values in the *SelectionFields* array.

SelectionFields
A pointer to an array of OID/value pairs providing search values for selecting one or more caller-owner certificates issued by a certificate authority. These fields are optional. If no selection fields are provided, then all caller-owner certificates from the CA will be returned for possible reclamation.

UserCredentials
A pointer to the set of one or more credentials being presented for authentication by the caller making the certificate request. The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function a service provider module can use to obtain one or more credentials. The callback function can be used to obtain a passphrase, a biometric input, or to perform a challenge and response protocol. The type of credentials accepted by a service provider module is defined and recorded in a record in the Module Directory Service records describing that provider. If the service provider module does not require credentials from a caller, then this field can be NULL.

8.4.43 CSSM_TP_CERTRECLAIM_STATUS

This set of defined constants indicates the status of a service request to reclaim a certificate.

```
typedef uint32 CSSM_TP_CERTRECLAIM_STATUS;

#define CSSM_TP_CERTRECLAIM_STATUS_UNKNOWN  (0x0)
    /* indeterminate */

#define CSSM_TP_CERTRECLAIM_OK              (0x1)
    /* a set of one or more certificates were returned by the CA
    for local recovery of the associated private key */

#define CSSM_TP_CERTRECLAIM_NOMATCH         (0x2)
    /* no certificates owned by the requester were found matching
     the specified selection fields */

#define CSSM_TP_CERTRECLAIM_REJECTED        (0x3)
    /* certificate reclamation failed due to some error condition */

#define CSSM_TP_CERTRECLAIM_NOT_AUTHORIZED  (0x4)
    /* certificate reclamation was not performed, the request
    was not authorized */
```

8.4.44 CSSM_TP_CERTRECLAIM_OUTPUT

This data structure aggregates the output information generated in response to a single request to reclaim a set of certificates and recovery the use of each associated private key. The returned certificates must be scanned and reclaimed individually using the functions *CSSM_TP_CertReclaimKey*() and *CSSM_TP_CertReclaimAbort*() to recover the private key to the local system.

```
typedef struct cssm_tp_certreclaim_output {
    CSSM_TP_CERTRECLAIM_STATUS ReclaimStatus;
    CSSM_CERTGROUP_PTR ReclaimedCertGroup;
    CSSM_LONG_HANDLE KeyCacheHandle;
} CSSM_TP_CERTRECLAIM_OUTPUT, *CSSM_TP_CERTRECLAIM_OUTPUT_PTR;
```

Definitions

ReclaimStatus
A value indicating the status of the certificate reclamation request.

ReclaimedCertGroup
A pointer to a structure containing a reference to a group of certificates and the number of certificates contained in that group. The certificate group contains all certificates that are candidates for reclamation. A certificate is a candidate if it satisfies at least the following criteria:

- The certificate is owned by the requester
- Recovery of the certificate is within the jurisdiction of the responding Certification Authority
- The certificate matched the input selection criteria

KeyCacheHandle
A handle to a cache of protected private keys. The cache is managed by the service provider. Each key is associated with one of the certificates in *ReclaimedCertGroup*.

8.4.45 CSSM_TP_CRLISSUE_INPUT

This data structure aggregates the input information for a request to issue the most current certificate revocation list (CRL). The desired CRL is identified by at least one of the following input values:

- *CrlIdentifier* - requesting the most current CRL issued after the CRL named by this identifier
- *CrlThisTtime* - requesting the most current CRL issued after this time
- *PolicyIdentifier* - requesting the most current CRL governing the policy domain indicated by this Policy identifier.

If CRL retrieval is a controlled operation, the caller can present the necessary credentials in *CallerCredentials*.

```
typedef struct cssm_tp_crlissue_input {
    CSSM_CL_HANDLE CLHandle;
    uint32 CrlIdentifier;
    CSSM_TIMESTRING CrlThisTime;
    CSSM_FIELD_PTR PolicyIdentifier;
    CSSM_ACCESS_CREDENTIALS_PTR CallerCredentials;
} CSSM_TP_CRLISSUE_INPUT, *CSSM_TP_CRLISSUE_INPUT_PTR;
```

Definitions

CLHandle
The identifier that uniquely describes a Certificate Library service provider that can be used to manipulate certificates and CRLs of the requested type on the local system. This is an optional input parameter for this request type.

CrlIdentifier
An integer that uniquely identifies the most current certificate revocation list known or held by the requester. If the CRL identifier is not known, or the requester has no previous local copy of a CRL, then this value can be zero.

CrlThisTime
A CSSM_TIMESTRING_PTR referencing a string containing the date and time of the most current certificate revocation list known or held by the requester. If the time is not known, or the requester has no previous local copy of a CRL from which to extract a time, then this pointer value can be NULL.

PolicyIdentifier
The policy identifier is an OID-value pair. The CSSM_OID structure contains the name of the policy and the value (contained in a CSSM_DATA structure) is an optional, caller-specified input value governing the interpretation and application of the policy. The name space for policy identifiers is defined externally by the application domains served by the Certification Authority. If the policy domain is implied by *CrlIdentifier* or can be determine by the jurisdiction of the requester's *CallerCredentialsi*, then this pointer value can be NULL.

CallerCredentials
A pointer to the set of one or more credentials being presented for authentication by the caller requesting a CRL from a Certification Authority. The credentials structure can

contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function a service provider module can use to obtain one or more credentials. The callback function can be used to obtain a passphrase, a biometric input, or to perform a challenge and response protocol. The type of credentials accepted by a service provider module is defined and recorded in a record in the Module Directory Service records describing that provider. If the service provider module does not require authenticated callers, then this field can be NULL.

8.4.46 CSSM_TP_CRLISSUE_STATUS

This set of defined constants indicates the status of a service request to issue the most recent certificate revocation list.

```
typedef uint32 CSSM_TP_CRLISSUE_STATUS;

#define CSSM_TP_CRLISSUE_STATUS_UNKNOWN      (0x0)
    /* indeterminate */
#define CSSM_TP_CRLISSUE_OK                  (0x1)
    /* a copy of the most current CRL was issued as requested
    and the time for issuing the next CRL is also returned */
#define CSSM_TP_CRLISSUE_NOT_CURRENT         (0x2)
    /* either no CRL has been issued since the CRL identified
    in the request, or it is not time to issue an updated CRL.
    no CRL has been returned, but the time for issuing the
    next CRL is included in the results */
#define CSSM_TP_CRLISSUE_INVALID_DOMAIN      (0x3)
    /* CRL domain was not recognized or was outside the
    CA jurisdiction, no CRL or time for the next CRL
    has been returned */
#define CSSM_TP_CRLISSUE_UNKNOWN_IDENTIFIER  (0x4)
    /* unrecognized CRL identifier,
    no CRL or time for the next CRL has been returned */
#define CSSM_TP_CRLISSUE_REJECTED            (0x5)
    /* CRL was not issued due to some error condition,
     no CRL or time for the next CRL has been returned */
#define CSSM_TP_CRLISSUE_NOT_AUTHORIZED      (0x6)
    /* CRL was not issued, the request was not authorized,
    no CRL or time for the next CRL has been returned */
```

8.4.47 CSSM_TP_CRLISSUE_OUTPUT

This data structure aggregates the output information generated in response to a single request to issue a copy of the most recent certificate revocation list (CRL).

```
typedef struct cssm_tp_crlissue_output {
    CSSM_TP_CRLISSUE_STATUS IssueStatus;
    CSSM_ENCODED_CRL_PTR Crl;
    CSSM_TIMESTRING CrlNextTime;
} CSSM_TP_CRLISSUE_OUTPUT, *CSSM_TP_CRLISSUE_OUTPUT_PTR;
```

Definitions

IssueStatus
A CSSM_TP_CRLISSUE_STATUS value indicating the status of a copy of the newly issued CRL and the time specified for issuing the next CRL update.

Crl
A pointer to a structure containing the encoded certificate revocation list and indicators specifying the type and encoding of the CRL representation. If the request failed and no CRL is returned, then this pointer value is NULL.

CrlNextTime
A CSSM_TIMESTRING referencing a string containing the anticipated date and time for issuing the next CRL update in the domain of the returned CRL. If the time is not known, then this pointer value can be NULL.

8.4.48 CSSM_TP_FORM_TYPE

```
typedef uint32 CSSM_TP_FORM_TYPE;

#define CSSM_TP_FORM_TYPE_GENERIC (0x0)
#define CSSM_TP_FORM_TYPE_REGISTRATION (0x1)
```

8.5 Error Codes and Error Values

The Error Values that can be returned by TP functions can be either derived from the Common Error Codes defined in Appendix A on page 925, or from a Common set that more than one TP function can return, or they are specific to a TP function.

The TP Error Values defined in this section list the TP Error Values in the Common set, plus any Error Values that are specific to a function.

8.5.1 TP Error Values Derived from Common Error Codes

```
#define CSSMERR_TP_INTERNAL_ERROR \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)
#define CSSMERR_TP_MEMORY_ERROR  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)
#define CSSMERR_TP_MDS_ERROR  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)
#define CSSMERR_TP_INVALID_POINTER  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)
#define CSSMERR_TP_INVALID_INPUT_POINTER  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)
#define CSSMERR_TP_INVALID_OUTPUT_POINTER  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)
#define CSSMERR_TP_FUNCTION_NOT_IMPLEMENTED  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)
#define CSSMERR_TP_SELF_CHECK_FAILED  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)
#define CSSMERR_TP_OS_ACCESS_DENIED  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)
#define CSSMERR_TP_FUNCTION_FAILED  \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)
#define CSSMERR_TP_INVALID_CONTEXT_HANDLE \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_CONTEXT_HANDLE)
#define CSSMERR_TP_INVALID_DATA \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DATA)
#define CSSMERR_TP_INVALID_DB_LIST \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_LIST)
#define CSSMERR_TP_INVALID_CERTGROUP_POINTER \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_CERTGROUP_POINTER)
#define CSSMERR_TP_INVALID_CERT_POINTER \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_CERT_POINTER)
#define CSSMERR_TP_INVALID_CRL_POINTER \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_CRL_POINTER)
#define CSSMERR_TP_INVALID_FIELD_POINTER \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_FIELD_POINTER)
#define CSSMERR_TP_INVALID_DATA \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DATA)
#define CSSMERR_TP_INVALID_NETWORK_ADDR \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_NETWORK_ADDR)
#define CSSMERR_TP_CRL_ALREADY_SIGNED \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_CRL_ALREADY_SIGNED)
#define CSSMERR_TP_INVALID_NUMBER_OF_FIELDS \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_NUMBER_OF_FIELDS)
#define CSSMERR_TP_VERIFICATION_FAILURE \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_VERIFICATION_FAILURE)
#define CSSMERR_TP_INVALID_DB_HANDLE     \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_HANDLE)
#define CSSMERR_TP_UNKNOWN_FORMAT \
```

```
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_UNKNOWN_FORMAT)
#define CSSMERR_TP_UNKNOWN_TAG \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_UNKNOWN_TAG)
#define CSSMERR_TP_INVALID_PASSTHROUGH_ID \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_PASSTHROUGH_ID)
#define CSSMERR_TP_INVALID_CSP_HANDLE \
    (CSSM_TP_BASE_ERROR + CSSM_ERRCODE_INVALID_CSP_HANDLE)
#define CSSMERR_TP_INVALID_DL_HANDLE \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DL_HANDLE)
#define CSSMERR_TP_INVALID_CL_HANDLE \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_CL_HANDLE)
#define CSSMERR_TP_INVALID_DB_LIST_POINTER \
    (CSSM_TP_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_LIST_POINTER)
```

8.5.2 Common TP Error Values

These values can be returned by one or more TP APIs.

```
#define CSSM_TP_BASE_TP_ERROR \
    (CSSM_TP_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT)
#define CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER \
    (CSSM_TP_BASE_TP_ERROR+1)
```

Invalid context pointer

```
#define CSSMERR_TP_INVALID_IDENTIFIER_POINTER (CSSM_TP_BASE_TP_ERROR+2)
```

Invalid identifier pointer

```
#define CSSMERR_TP_INVALID_KEYCACHE_HANDLE (CSSM_TP_BASE_TP_ERROR+3)
```

Invalid key cache handle

```
#define CSSMERR_TP_INVALID_CERTGROUP (CSSM_TP_BASE_TP_ERROR+4)
```

Invalid structure or unknown format for certificate group or certificates in the group

```
#define CSSMERR_TP_INVALID_CRLGROUP (CSSM_TP_BASE_TP_ERROR+5)
```

Invalid structure or unknown format for CRL group or CRLs in the group

```
#define CSSMERR_TP_INVALID_CRLGROUP_POINTER (CSSM_TP_BASE_TP_ERROR+6)
```

Invalid CRL group pointer

```
#define CSSMERR_TP_AUTHENTICATION_FAILED (CSSM_TP_BASE_TP_ERROR+7)
```

Invalid/unauthorized credentials

```
#define CSSMERR_TP_CERTGROUP_INCOMPLETE (CSSM_TP_BASE_TP_ERROR+8)
```

Incomplete certificate group

```
#define CSSMERR_TP_CERTIFICATE_CANT_OPERATE (CSSM_TP_BASE_TP_ERROR+9)
```

Cannot perform the requested operation (sign/verify/database-apply) with input signer or revoker certificate

```
#define CSSMERR_TP_CERT_EXPIRED (CSSM_TP_BASE_TP_ERROR+10)
```

Certificate has expired

```
#define CSSMERR_TP_CERT_NOT_VALID_YET (CSSM_TP_BASE_TP_ERROR+11)
```

Certificate not valid until a future date

```
#define CSSMERR_TP_CERT_REVOKED (CSSM_TP_BASE_TP_ERROR+12)
```

Certificate has been revoked

```
#define CSSMERR_TP_CERT_SUSPENDED (CSSM_TP_BASE_TP_ERROR+13)
```

Certificate is currently suspended from use

```
#define CSSMERR_TP_INSUFFICIENT_CREDENTIALS (CSSM_TP_BASE_TP_ERROR+14)
```

Insufficient caller credentials for this operation

```
#define CSSMERR_TP_INVALID_ACTION (CSSM_TP_BASE_TP_ERROR+15)
```

Invalid action

```
#define CSSMERR_TP_INVALID_ACTION_DATA (CSSM_TP_BASE_TP_ERROR+16)
```

Invalid data specified for this action

```
#define CSSMERR_TP_INVALID_ANCHOR_CERT (CSSM_TP_BASE_TP_ERROR+17)
```

Invalid anchor certificate

```
#define CSSMERR_TP_INVALID_AUTHORITY (CSSM_TP_BASE_TP_ERROR+18)
```

Invalid or unreachable authority

```
#define CSSMERR_TP_VERIFY_ACTION_FAILED (CSSM_TP_BASE_TP_ERROR+19)
```

Unable to determine trust for action

```
#define CSSMERR_TP_INVALID_CERTIFICATE (CSSM_TP_BASE_TP_ERROR+20)
```

Invalid certificate

```
#define CSSMERR_TP_INVALID_CERT_AUTHORITY (CSSM_TP_BASE_TP_ERROR+21)
```

Certificate group is not signed by a recognized issuing authority

```
#define CSSMERR_TP_INVALID_CRL_AUTHORITY (CSSM_TP_BASE_TP_ERROR+22)
```

Certificate Revocation List is from an unrecognized issuing authority.

```
#define CSSMERR_TP_INVALID_CRL_ENCODING (CSSM_TP_BASE_TP_ERROR+23)
```

Invalid encoding for CRL

```
#define CSSMERR_TP_INVALID_CRL_TYPE (CSSM_TP_BASE_TP_ERROR+24)
```

Invalid type for CRL

```
#define CSSMERR_TP_INVALID_CRL (CSSM_TP_BASE_TP_ERROR+25)
```

Invalid CRL

```
#define CSSMERR_TP_INVALID_FORM_TYPE (CSSM_TP_BASE_TP_ERROR+26)
```

Invalid argument for form type

```
#define CSSMERR_TP_INVALID_ID (CSSM_TP_BASE_TP_ERROR+27)
```

Invalid pass through ID

```
#define CSSMERR_TP_INVALID_IDENTIFIER (CSSM_TP_BASE_TP_ERROR+28)
```

Unknown reference identifier

```
#define CSSMERR_TP_INVALID_INDEX (CSSM_TP_BASE_TP_ERROR+29)
```

Certficate index is invalid

```
#define CSSMERR_TP_INVALID_NAME (CSSM_TP_BASE_TP_ERROR+30)
```

Certificate contains unrecognized names

```
#define CSSMERR_TP_INVALID_POLICY_IDENTIFIERS (CSSM_TP_BASE_TP_ERROR+31)
```

Invalid policy identifier

```
#define CSSMERR_TP_INVALID_TIMESTRING (CSSM_TP_BASE_TP_ERROR+32)
```

Invalid CSSM_TIMESTRING

```
#define CSSMERR_TP_INVALID_REASON (CSSM_TP_BASE_TP_ERROR+33)
```

Invalid argument for reason

```
#define CSSMERR_TP_INVALID_REQUEST_INPUTS (CSSM_TP_BASE_TP_ERROR+34)
```

Invalid request input parameters

```
#define CSSMERR_TP_INVALID_RESPONSE_VECTOR (CSSM_TP_BASE_TP_ERROR+35)
```

Invalid vector of responses

```
#define CSSMERR_TP_INVALID_SIGNATURE (CSSM_TP_BASE_TP_ERROR+36)
```

Certificate signature is invalid

```
#define CSSMERR_TP_INVALID_STOP_ON_POLICY (CSSM_TP_BASE_TP_ERROR+37)
```

Invalid stop on policy

```
#define CSSMERR_TP_INVALID_CALLBACK (CSSM_TP_BASE_TP_ERROR+38)
```
Invalid callback

```
#define CSSMERR_TP_INVALID_TUPLE (CSSM_TP_BASE_TP_ERROR+39)
```
Invalid tuple

```
#define CSSMERR_TP_NOT_SIGNER (CSSM_TP_BASE_TP_ERROR+40)
```
Signer certificate is not signer of subject

```
#define CSSMERR_TP_NOT_TRUSTED (CSSM_TP_BASE_TP_ERROR+41)
```
Signature can not be trusted

```
#define CSSMERR_TP_NO_DEFAULT_AUTHORITY (CSSM_TP_BASE_TP_ERROR+42)
```
Unspecified authority with no default authority

```
#define CSSMERR_TP_REJECTED_FORM (CSSM_TP_BASE_TP_ERROR+43)
```
Form was rejected

```
#define CSSMERR_TP_REQUEST_LOST (CSSM_TP_BASE_TP_ERROR+44)
```
Authority lost the request. Must resubmit

```
#define CSSMERR_TP_REQUEST_REJECTED (CSSM_TP_BASE_TP_ERROR+45)
```
Authority rejected the request with no specific results returned

```
#define CSSMERR_TP_UNSUPPORTED_ADDR_TYPE (CSSM_TP_BASE_TP_ERROR+46)
```
Unsupported type of network address

```
#define CSSMERR_TP_UNSUPPORTED_SERVICE (CSSM_TP_BASE_TP_ERROR+47)
```
Unsupported TP service requested

```
#define CSSMERR_TP_INVALID_TUPLEGROUP_POINTER (CSSM_TP_BASE_TP_ERROR+48)
```
Invalid tuple group pointer

```
#define CSSMERR_TP_INVALID_TUPLEGROUP (CSSM_TP_BASE_TP_ERROR+49)
```
Invalid structure or unknown format for tuple group or tuples in the group

8.6 Trust Policy Operations

The man-page definitions for Trust Policy operations are presented in this section.

NAME

CSSM_TP_SubmitCredRequest for the CSSM API
TP_SubmitCredRequest for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_SubmitCredRequest
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority,
    CSSM_TP_AUTHORITY_REQUEST_TYPE RequestType,
    const CSSM_TP_REQUEST_SET *RequestInput,
    const CSSM_TP_CALLERAUTH_CONTEXT *CallerAuthContext,
    sint32 *EstimatedTime,
    CSSM_DATA_PTR ReferenceIdentifier)

SPI:
CSSM_RETURN CSSMTPI TP_SubmitCredRequest
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority,
    CSSM_TP_AUTHORITY_REQUEST_TYPE RequestType,
    const CSSM_TP_REQUEST_SET *RequestInput,
    const CSSM_TP_CALLERAUTH_CONTEXT *CallerAuthContext,
    sint32 *EstimatedTime,
    CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

If the caller is successfully authenticated, then this function submits a request to the Authority identified by *PreferredAuthority*. The authority service can be local or remote. If the Authority is not specified, then the TP module can assume a default authority based on the *RequestType* and the *CallerAuthContext*. *RequestType* indicates the type of Authority request and *RequestInput* specifies the input parameters needed by the authority to perform the request.

The request is submitted to the authority only if the TP module can successfully authenticate the caller. The *CallerAuthContext* presents the caller's credentials and a list of one or more policies under which the caller should be authenticated. Caller credentials can be presented in several forms:

- Memory-resident credential values, directly referenced by the structure
- Data bases containing credentials
- Callback functions that can be invoked to obtain credentials from an active entity

The local service provider must select and forward the credentials required by the Authority. The caller must provide all necessary credentials through the *CallerAuthContext* parameter.

If the caller can not be authenticated by the local service provider, the function fails and the request is not submitted to the selected authority.

This function returns a *ReferenceIdentifier* and an *EstimatedTime* (specified in seconds). *ReferenceIdentifier* is an ID for the submitted request. *EstimatedTime* defines the expected time to process the request. This time may be substantial when the request requires offline authentication procedures by the Authority process. In contrast, the estimated time can be zero, meaning the result can be obtained immediately using *CSSM_TP_RetrieveCredResult*() (CSSM API) or *TP_RetrieveCredResult*() (TP SPI). After the specified time has elapsed, the caller must use the function *CSSM_TP_RetrieveCredResult*() (CSSMAPI) or *TP_RetrieveCredResult*() (TP SPI) with the reference identifier, to obtain the result of the request.

PARAMETERS

TPHandle (input)
The handle that describes the certification authority module used to perform this function.

PreferredAuthority (input/optional)
The identifier which uniquely describes the Certificate Service Authority to submit the request to.

RequestType (input)
The identifier of the type of request to submit.

RequestInput (input)
A pointer to the input parameters to be submitted to the authority who will perform the requested service.

CallerAuthContext (input/optional)
This structure contains a set of caller authentication credentials. The authentication information can be a passphrase, a PIN, a completed registration form, a certificate, or a template of user-specific data. The required set of credentials is defined by the service provider module and recorded in the MDS Primary relation. Multiple credentials can be required. If the local service provider module does not require credentials from a caller, then the *CallerCredentials* field of this verification context structure can be NULL. The structure optionally contains additional credentials that can be used to support the authentication process. Authentication credentials required by the authority should be included in the *RequestInput*. The local service provider module can forward this credential information to the authority, as appropriate, but is not required to do so.

EstimatedTime (output)
The number of estimated seconds before the service results are ready to be retrieved. A (default) value of zero indicates that the results can be retrieved immediately via the corresponding *CSSM_TP_RetrieveCredResult*( ) (CSSM API) or *TP_RetrieveCredResult*( ) (TP SPI) function call. When the local service provider module or the authority cannot estimate the time required to perform the requested service, 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 and becomes undefined when all local processing of the request has completed. Local processing is completed in one of two ways:

- For certificate services that do not require explicit confirmation by the requester, the reference identifier is invalidated when the corresponding *CSSM_TP_RetrieveCredResult*( ) (CSSM API) or *TP_RetrieveCredResult*( ) (TP SPI) function completes (by returning valid results or by failure, which blocks returned results)
- For certificate services that require explicit confirmation by the requester, the reference identifier is invalidated by successfully invoking the function *CSSM_TP_ConfirmCredResu*( ) (CSSM API) or *CSSM_TP_ConfirmCredResult*( ) (TP SPI).

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_AUTHORITY
CSSMERR_TP_NO_DEFAULT_AUTHORITY
CSSMERR_TP_UNSUPPORTED_ADDR_TYPE
CSSMERR_TP_INVALID_NETWORK_ADDR
CSSMERR_TP_UNSUPPORTED_SERVICE
CSSMERR_TP_INVALID_REQUEST_INPUTS
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME

SEE ALSO

For the CSSM API:
CSSM_TP_RetrieveCredResult( )

For the TP SPI:
TP_RetrieveCredResult( )

NAME

CSSM_TP_RetrieveCredResult

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_TP_RetrieveCredResult
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA *ReferenceIdentifier,
    const CSSM_TP_CALLERAUTH_CONTEXT *CallerAuthCredentials,
    sint32 *EstimatedTime,
    CSSM_BOOL *ConfirmationRequired,
    CSSM_TP_RESULT_SET_PTR *RetrieveOutput)
```

DESCRIPTION

This function returns the results of a *CSSM_TP_SubmitCredRequest*() call.

The single identifier *ReferenceIdentifier* denotes the *CSSM_TP_SubmitCredRequest*() invocation that initiated the request.

It is possible that the results are not ready to be retrieved when this call is made. In that case, an *EstimatedTime* to complete processing is returned. The caller must attempt to retrieve the results again after the estimated time to completion has elapsed.

This function can fail in total for any one of the following reasons:

- The reference identifier is invalid
- The TP process can not be located
- The TP process encountered a fatal error when attempting to process the request.

When this function completes, the set of return results is ordered corresponding to the order of the originating request.

Some certificate services require the requester to confirm retrieval of the results. *ConfirmationRequired* indicates whether the caller must confirm completion of *CSSM_TP_RetrieveCredResult*() by calling *CSSM_TP_ConfirmCredResult*().

PARAMETERS

TPHandle (input)
The handle that describes the certification authority module used to perform this function.

ReferenceIdentifier (input)
A reference identifier that uniquely identifies the *CSSM_TP_SubmitCredRequest*() call that initiated the certificate service request whose results are returned by this function. The identifier persists across application executions and becomes undefined when all local processing of the request has completed. Local processing is completed in one of two ways:

- For certificate services that do not require explicit confirmation by the requester, the reference identifier is invalidated when the corresponding *CSSM_TP_RetrieveCredResult*() function completes (by returning valid results or by failure, which blocks returned results)
- For certificate services that require explicit confirmation by the requester, the reference identifier is invalidated by successfully invoking the function *CSSM_TP_ConfirmCredResult*().

CallerAuthCredentials (input/optional)
This structure contains a set of caller authentication credentials. The authentication information can be a passphrase, a PIN, a completed registration form, a certificate, or a

template of user-specific data. The required set of credentials is defined by the service provider module and recorded in a record in the MDS Primary relation. Multiple credentials can be required. If the local service provider module does not require credentials from a caller, then the *Credentials* field of this verification context structure can be NULL. The structure optionally contains additional credentials that can be used to support the authentication process. Authentication credentials required by the authority should be included in the *RequestInput*. The local TP module can forward information from the *CallerAuthCredentials* to the authority, as appropriate, but is not required to do so.

EstimatedTime (output)
The number of seconds estimated before the results of a requested service will be returned to the requester. When the local TP module or the authority process cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ConfirmationRequired (output)
A boolean value indicating whether the caller must invoke *CSSM_TP_ConfirmCredResult*() to acknowledge retrieving the results of the service request. CSSM_TRUE indicates the caller must call *CSSM_TP_ConfirmCredResult*(). CSSM_FALSE indicates that the caller must not call *CSSM_TP_ConfirmCredResult*(). The value of this output parameter is not applicable until the *CSSM_TP_RetrieveCredResult*() completes by returning results of the request or terminates in unrecoverable failure.

RetrieveOutput (output)
A pointer to the results returned by the authority in response to the service requests submitted by *CSSM_TP_SubmitCredRequest*(). The output results are ordered corresponding to the requests. The structure of the response set is determined by the type of request. The caller and the service provider must retain knowledge of the request type associated with the *ReferenceIdentifier*.

RETURN VALUE

A CSSM_RETURN value combined with estimated time to indicate one of three results:

Complete Function Result	Function Return Value	RetrieveOutput	EstimatedTime
Request results returned to caller	CSSM_OK	Non-NULL pointer	NA
Request results not ready, but expected in the future	CSSM_OK	NULL pointer	CSSM_ESTIMATED_TIME_UNKNOWN or <estimated seconds>
Fatal Error, results will never be returned	(!CSSM_OK)	NA	NA

For a return value of (!CSSM_OK) the return value represents a specific error code.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_IDENTIFIER_POINTER
CSSMERR_TP_INVALID_IDENTIFIER
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK

CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_REQUEST_LOST
CSSMERR_TP_REQUEST_REJECTED

SEE

For the CSSM API:
CSSM_TP_SubmitCredRequest( )

For the TP SPI:
TP_SubmitCredRequest( )

NAME

CSSM_TP_ConfirmCredResult for the CSSM API
TP_ConfirmCredResult for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_ConfirmCredResult
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA *ReferenceIdentifier,
    const CSSM_TP_CALLERAUTH_CONTEXT *CallerAuthCredentials,
    const CSSM_TP_CONFIRM_RESPONSE *Responses,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority)

SPI:
CSSM_RETURN CSSMTPI TP_ConfirmCredResult
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA *ReferenceIdentifier,
    const CSSM_TP_CALLERAUTH_CONTEXT *CallerAuthCredentials,
    const CSSM_TP_CONFIRM_RESPONSE *Responses,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority)
```

DESCRIPTION

This function submits a vector of acknowledgements to a Certificate Authority for a set of requests and corresponding results identified by *ReferenceIdentifier*. The caller must execute the call sequence *CSSM_TP_SubmitCredRequest*( ) and *CSSM_TP_RetrieveCredResult*( ) (or the equivalent TP SPI calls) to submit a set of requests and to retrieve the results of those requests. Some Certificate Authority services accessed through the request and retrieve functions require confirmation. The function *CSSM_TP_RetrieveCredResult*( ) (CSSM API) or *TP_RetrieveCredResult*( ) (TP SPI) returns a value indicating whether the caller must invoke *CSSM_TP_ConfirmCredResult*( ) (CSSM API) or *TP_ConfirmCredResult*( ) (TP SPI) to successfully complete the service.

The *Responses* vector accepts or rejects each result independently. If the caller rejects a returned result, the action taken by the authority depends on the requested type of service.

The *ReferenceIdentifier* also identifies the authority process state associated with the function pair *CSSM_TP_SubmitCredRequest*( ) and *CSSM_TP_RetrieveCredResult*( ) (or the equivalent TP SPI calls). The *PreferredAuthority* information can be used to further identify the authority to receive the acknowledgement. After successful execution of this function, the value of the *ReferenceIdentifier* is undefined and should not be used in subsequent operations in the current module attach session.

This function fails if *ReferenceIdentifier* is invalid or the Authority process can not be located.

PARAMETERS

TPHandle (input)
The handle that describes the certification authority module used to perform this function.

ReferenceIdentifier (input)
A reference identifier that uniquely identifies execution of the call sequence *CSSM_TP_SubmitCredRequest*( ) and *CSSM_TP_RetrieveCredResult*( ) (or the equivalent TP SPI call pair) to submit a set of requests and to retrieve the results of those requests.

CallerAuthCredentials (input/optional)
This structure contains a set of caller authentication credentials. The authentication information can be a passphrase, a PIN, a completed registration form, a certificate, or a

template of user-specific data. The required set of credentials is defined by the service provider module and recorded in a record in the MDS Primary relation. Multiple credentials can be required. If the local service provider module does not require credentials from a caller, then the *Credentials* field of this verification context structure can be NULL. The structure optionally contains additional credentials that can be used to support the authentication process. Authentication credentials required by the authority should be included in the *RequestInput*. The local TP module can forward information from the *CallerAuthCredentials* to the authority, as appropriate, but is not required to do so.

Responses (input)
An ordered vector of acknowledges indicating the caller's acceptance or rejection of results. The vector contains one acknowledgement per result returned by *CSSM_TP_RetrieveCredResult*( ) (CSSM API) or *TP_RetrieveCredResult*( ) (TP SPI).

PreferredAuthority (input/optional)
The identifier which uniquely describes the Authority to receive the acknowledgements. The structure can include:

- An identity certificate for the authority
- The location of the authority

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_IDENTIFIER_POINTER
CSSMERR_TP_INVALID_IDENTIFIER
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_INVALID_RESPONSE_VECTOR
CSSMERR_TP_INVALID_AUTHORITY
CSSMERR_TP_NO_DEFAULT_AUTHORITY
CSSMERR_TP_UNSUPPORTED_ADDR_TYPE

CSSMERR_TP_INVALID_NETWORK_ADDR

SEE ALSO

For the CSSM API:
CSSM_TP_SubmitCredRequest( )
CSSM_TP_RetrieveCredResult( )
CSSM_TP_ReceiveConfirmation( )

For the TP SPI:
TP_SubmitCredRequest( )
TP_RetrieveCredResult( )
TP_ReceiveConfirmation( )

NAME

CSSM_TP_ReceiveConfirmation for the CSSM API
TP_ReceiveConfirmation for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_ReceiveConfirmation
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA *ReferenceIdentifier,
    CSSM_TP_CONFIRM_RESPONSE_PTR *Responses,
    sint32 *ElapsedTime)
SPI:
CSSM_RETURN CSSMTPI TP_ReceiveConfirmation
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA *ReferenceIdentifier,
    CSSM_TP_CONFIRM_RESPONSE_PTR *Responses,
    sint32 *ElapsedTime)
```

DESCRIPTION

A certificate authority uses this function to poll for confirmation from a requester who has been served by the authority. A requester sends a confirmation to the authority by successfully invoking the function *CSSM_TP_ConfirmCredResult*( ) (CSSM API) or *TP_ConfirmCredResult*( ) (TP SPI).

The *ReferenceIdentifier* uniquely identifies the service request and corresponding results for which confirmation is expected. This reference identifier need not be identical to the reference identifier used by the requester, but a one-to-one mapping between the two name spaces must be well-defined.

Responses is an ordered vector of acknowledgements indicating, for each returned result, whether the result was accepted or rejected by the original requester for whom the service was performed. If a result is rejected by the receiver, then the authority process must undo the service if a reverse operation is possible and available.

If a fatal error occurs, this function returns an error code, indicating that the function call can never be completed. If confirmation has not been received, the function return value is CSSM_OK and the *ElapsedTime* is returned to the caller of this function. The time represents elapsed seconds since the service results were produced by the authority process. Note that there can be a difference between the time the authority process produces the results and the time the results are actually received by the requester. Elapsed time is relative and should increase with consecutive calls using the same *ReferenceIdentifier*. If the TP module has no knowledge of the elapsed time, the value CSSM_ELAPSED_TIME_UNKNOWN must be returned. If the service requester has confirmed receipt of the service results, this function returns CSSM_OK and *ElapsedTime* is CSSM_ELAPSED_TIME_COMPLETE.

This function can be invoked repeatedly until the confirmation is received or until the caller decides the acknowledgement may be lost and chooses to undo the results of the original service request.

This function fails if the *ReferenceIdentifier* is invalid or does not match any requested service for which confirmation is expected.

PARAMETERS

TPHandle (input)
The handle that describes the certification authority module used to perform this function.

ReferenceIdentifier (input)
A reference identifier that uniquely identifies a set of service requests and the results created in response to those requests.

Responses (output)
An ordered vector of acknowledges indicating the caller's acceptance or rejection of results. The vector contains one acknowledgement per result created by the certificate authority.

ElapsedTime (output) If the confirmation has not been received, this output value is the number of seconds elapsed since the certificate authority created the results or CSSM_ELAPSED_TIME_UNKNOWN. If the confirmation has been received, this output value is CSSM_ELAPSED_TIME_COMPLETE.

RETURN VALUE

A CSSM return value combined with elapsed time to indicate one of three results:

Complete Function Result	Function Return Value	RetrieveOutput	EstimatedTime
Confirmation Received	CSSM_OK	CSSM_ELAPSED_TIME_COMPLETE	
Confirmation not received, but expected in the future	CSSM_OK	CSSM_ELAPSED_TIME_UNKNOWN or <elapsed seconds>	
Fatal Error, Confirmation is not expected	(!CSSM_OK)	NA	

For a return value of (!CSSM_OK) the return value is an error code.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_IDENTIFIER_POINTER
CSSMERR_TP_INVALID_IDENTIFIER

SEE ALSO

For the CSSM API:
CSSM_TP_ConfirmCredResult( )

For the TP SPI:
CSSM_TP_ConfirmCredResult( )

NAME

CSSM_TP_CertReclaimKey for the CSSM API
TP_CertReclaimKey for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertReclaimKey
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_CERTGROUP *CertGroup,
    uint32 CertIndex,
    CSSM_LONG_HANDLE KeyCacheHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry)

SPI:
CSSM_RETURN CSSMTPI TP_CertReclaimKey
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_CERTGROUP *CertGroup,
    uint32 CertIndex,
    CSSM_LONG_HANDLE KeyCacheHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry)
```

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 reclaimed from a certificate authority.

The particular private key to be recovered to the local system is identified by its associated certificate. The certificate is identified by its *CertIndex* position within the *CertGroup*.

The reclamation process associates the private key with the public key contained in the certificate, and securely stores the private key in the specified cryptographic service provider. The CSP can require that the caller provide access credentials authorizing inserting a new key into the CSP through an *UnwrapKey* operation. The caller should also provide an initial Access Control List (ACL) entry for the newly inserted key. The ACL entry is used to control future use of the recovered private key. These inputs are provided in *CredAndAclEntry*.

When all required private keys have been reclaimed, the key cache can be discarded using the function *CSSM_TP_CertReclaimAbort*() (CSSM API) or *TP_CertReclaimAbort*() (TP SPI). The caller must free the *CertGroup* when it is no longer needed.

PARAMETERS

TPHandle (input)
The handle that describes the service provider module used to perform this operation.

CertGroup (input)
A pointer to a structure containing a reference to a group of certificates and the number of certificates contained in that group. The certificate group contains all certificates that are candidates for reclamation.

CertIndex (input)
An index value that identifies the certificate whose associated private key is to be recovered and stored in the local CSP. This index value I references the I-th certificate in *CertGroup*.

KeyCacheHandle (input)
A reference handle that uniquely identifies the cache of protected private keys associated with the reclaimed certificates contained in *CertGroup*. The structure of the cache is opaque to the caller.

CSPHandle (input)
The handle that describes the CSP module where the private key is to be stored. Optionally, the CA service provider can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the CSP to acquire the credentials and/or the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be an empty list.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_INDEX
CSSMERR_TP_INVALID_KEYCACHE_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS

SEE ALSO

For the CSSM API:
CSSM_TP_RetrieveCredResult( )
CSSM_TP_Cert_ReclaimAbort( )

For the TP SPI:
TP_RetrieveCredResult( )
TP_Cert_ReclaimAbort( )

NAME

CSSM_TP_CertReclaimAbort for the CSSM API
TP_CertReclaimAbort for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertReclaimAbort
    (CSSM_TP_HANDLE TPHandle,
    CSSM_LONG_HANDLE KeyCacheHandle)

SPI:
CSSM_RETURN CSSMTPI TP_CertReclaimAbort
    (CSSM_TP_HANDLE TPHandle,
    CSSM_LONG_HANDLE KeyCacheHandle)
```

DESCRIPTION

This function terminates the iterative process of reclaiming certificates and recovering their associated private keys from a protected key cache. This function must be called even if all private keys are recovered from the cache. This function destroys all intermediate state and secret information used during the reclamation process. At completion of this function, the cache handle is invalid.

PARAMETERS

TPHandle (input)
The handle that describes the service provider module used to perform this function.

KeyCacheHandle (input)
An opaque handle that identifies the cache of protected private keys reclaimed from a certificate authority for potentially recovery on the local system.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_KEYCACHE_HANDLE

SEE ALSO

For the CSSM API:
CSSM_TP_CertReclaimKey()

For the TP SPI:
TP_CertReclaimKey()

NAME

CSSM_TP_FormRequest for the CSSM API
TP_FormRequest for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_FormRequest
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority,
    CSSM_TP_FORM_TYPE FormType,
    CSSM_DATA_PTR BlankForm)

SPI:
CSSM_RETURN CSSMTPI TP_FormRequest
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_TP_AUTHORITY_ID *PreferredAuthority,
    CSSM_TP_FORM_TYPE FormType,
    CSSM_DATA_PTR BlankForm)
```

DESCRIPTION

This function returns a blank form of type *FormType* from an Authority. If the *PreferredAuthority* list is NULL, the CA module can use a default authority name and location based on *FormType*. The CA module must incorporate knowledge of the interface protocol for communicating with the authority.

PARAMETERS

TPHandle (input)
The handle that describes the certification authority module used to perform this function.

PreferredAuthority (input/optional)
A CSSM_TP_AUTHORITY_ID structure containing either a certificate that identifies the Authority process, or a network address directly or indirectly identifying the location of the authority. If the input is NULL, the module can assume a default authority location. If a default authority can not be assumed, the request can not be initiated and the operation fails.

FormType (input) ,br Indicates the type of form being requested.

BlankForm (output)
A CSSM_DATA structure containing the requested form. The caller must have knowledge of the structure of the form based on *FormType*.

RETURN VALUE

ERRORS

CSSMERR_TP_INVALID_AUTHORITY
CSSMERR_TP_NO_DEFAULT_AUTHORITY
CSSMERR_TP_UNSUPPORTED_ADDR_TYPE
CSSMERR_TP_INVALID_NETWORK_ADDR
CSSMERR_TP_INVALID_FORM_TYPE

SEE ALSO

For the CSSM API:
CSSM_TP_FormSubmit( )

For the TP SPI:
TP_FormSubmit()

NAME

CSSM_TP_FormSubmit for the CSSM API
TP_FormSubmit for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_FormSubmit
    (CSSM_TP_HANDLE TPHandle,
    CSSM_TP_FORM_TYPE FormType,
    const CSSM_DATA *Form,
    const CSSM_TP_AUTHORITY_ID *ClearanceAuthority,
    const CSSM_TP_AUTHORITY_ID *RepresentedAuthority,
    CSSM_ACCESS_CREDENTIALS_PTR Credentials)

SPI:
CSSM_RETURN CSSMTPI TP_FormSubmit
    (CSSM_TP_HANDLE TPHandle,
    CSSM_TP_FORM_TYPE FormType,
    const CSSM_DATA *Form,
    const CSSM_TP_AUTHORITY_ID *ClearanceAuthority,
    const CSSM_TP_AUTHORITY_ID *RepresentedAuthority,
    CSSM_ACCESS_CREDENTIALS_PTR Credentials)
```

DESCRIPTION

The completed Form is submitted to a *ClearanceAuthority*, who is acting on behalf of a *RepresentedAuthority*. Typically the submitted form is requesting an authorization credential required as input to future service requests to the *RepresentedAuthority*.

If the form is honored by the *ClearanceAuthority*, then a set of one or more *Credentials* is returned to the requester. These credential can be used as the input credential in future service requests submitted to the *RepresentedAuthority*.

PARAMETERS

TPHandle (input)
A handle for the service provider module that will perform the operation.

FormType (input)
Indicates the type of form being submitted.

Form (input)
A pointer to the CSSM_DATA structure containing the completed form to be submitted to the *ClearanceAuthority*.

ClearanceAuthority (input/optional)
A CSSM_TP_AUTHORITY_ID structure containing either a certificate that identifies the clearance authority process, or a network address directly or indirectly identifying the location of the authority. If the input is NULL, the service provider module can assume a default authority based on the *FormType* and contents of *Form*. If a default authority can not be assumed, the request can not be initiated and the operation fails.

RepresentedAuthority (input/optional)
A CSSM_TP_AUTHORITY_ID structure containing either a certificate that identifies the authority represented by the *ClearanceAuthority*, or a network address directly or indirectly identifying the location of the authority. If the input is NULL, the service provider module can assume a default authority based on the *FormType* and contents of *Form*. If a default authority can not be assumed, the request can not be initiated and the operation fails.

Credentials (output/optional)

A pointer to a structure containing one or more credentials issued in response to the contents of the *Form*. If the output is NULL, either no credentials were returned or an error occurred.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_FORM_TYPE
CSSMERR_TP_INVALID_AUTHORITY
CSSMERR_TP_NO_DEFAULT_AUTHORITY
CSSMERR_TP_UNSUPPORTED_ADDR_TYPE
CSSMERR_TP_INVALID_NETWORK_ADDR
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_REJECTED_FORM

SEE ALSO

For the CSSM API:
CSSM_TP_FormRequest()

For the TP SPI:
TP_FormRequest()

8.7 Local Application-Domain-Specific Trust Policy Functions

The man-page definitions for Local Application-Domain-Specific Trust Policy functions are presented in this section.

NAME

CSSM_TP_CertGroupVerify for the CSSM API
TP_CertGroupVerify for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertGroupVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CERTGROUP *CertGroupToBeVerified,
    const CSSM_TP_VERIFY_CONTEXT *VerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR VerifyContextResult)

SPI:
CSSM_RETURN CSSMTPI TP_CertGroupVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CERTGROUP *CertGroupToBeVerified,
    const CSSM_TP_VERIFY_CONTEXT *VerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR VerifyContextResult)
```

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 *VerifyContext* 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 that can be used to perform the cryptographic operations required to carry out the verification. If no CSP handle is specified, the TP module allocates a suitable CSP.

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/optional)
A structure containing credentials, policy information, and contextual information to be used in the verification process. All of the input values in the context are optional except *Action*. The service provider can define default values or can attempt to operate without input for all the other fields of this input structure. The operation can fail if a necessary input value is omitted and the service module can not define an appropriate default value.

VerifyContextResult (output/optional)
A pointer to a structure containing information generated during the verification process. The information can include:

Evidence	(output/optional)
NumberOfEvidences	(output/optional)

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED

CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME

NAME

CSSM_TP_CertCreateTemplate for the CSSM API
TP_CertCreateTemplate for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertCreateTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CertFields,
    CSSM_DATA_PTR CertTemplate)

SPI:
CSSM_RETURN CSSMTPI TP_CertCreateTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CertFields,
    CSSM_DATA_PTR CertTemplate)
```

DESCRIPTION

This function allocates and initializes memory for an encoded certificate template output in *CertTemplate→Data*. The template values are specified by the input OID/value pairs contained in *CertFields*. The initialization process includes encoding all certificate field values according to the certificate type and certificate template encoding supported by the trust policy library module. The *CertTemplate* output is an unsigned certificate template in the format supported by the TP.

The memory for *CertTemplate→Data* is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory.

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 certificate library module used to perform this function.

NumberOfFields (input)
The number of certificate field values specified in the *CertFields*.

CertFields (input)
A pointer to an array of OID/value pairs that identifies the field values to initialize a new certificate.

CertTemplate (output)
A pointer to a CSSM_DATA structure that will contain the unsigned certificate template as a result of this function.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_FIELD_POINTER
CSSMERR_TP_UNKNOWN_TAG
CSSMERR_TP_INVALID_NUMBER_OF_FIELDS

SEE ALSO

For the CSSM API:
CSSM_TP_CertGetAllTemplateFields( )
CSSM_TP_CertSign( )

For the TP SPI:
TP_CertGetAllTemplateFields( )
TP_CertSign( )

NAME

CSSM_TP_CertGetAllTemplateFields for the CSSM API
TP_CertGetAllTemplateFields for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertGetAllTemplateFields
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *CertTemplate,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *CertFields)

SPI:
CSSM_RETURN CSSMTPI TP_CertGetAllTemplateFields
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *CertTemplate,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *CertFields)
```

DESCRIPTION

This function extracts and returns all field values from *CertTemplate*. *CertTemplate* is an unsigned certificate template in the format supported by the TP. Fields are returned as a set of OID-value pairs. The OID identifies the TP certificate template field and the data format of the value extracted from that field. The Trust Policy module defines and uses a preferred data format for returning field values from this function. Memory for the *CertFields* output is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory, by calling *CSSM_CL_FreeFields*() (CSSM API) or *CL_FreeFields*() (TP SPI).

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 certificate library module used to perform this function.

CertTemplate (input)
A pointer to the CSSM_DATA structure containing the packed, encoded certificate template.

NumberOfFields (output)
The length of the output array of fields.

CertFields (output)
A pointer to an array of CSSM_FIELD structures which contains the OIDs and values of the fields of the input certificate template.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_FIELD_POINTER

CSSMERR_TP_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_TP_CertCreateTemplate( )
CSSM_TP_CertSign( )

For the TP SPI:
TP_CertCreateTemplate( )
TP_CertSign( )

NAME

CSSM_TP_CertSign for the CSSM API
TP_CertSign for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertTemplateToBeSigned,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *SignerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR SignerVerifyResult,
    CSSM_DATA_PTR SignedCert)

SPI:
CSSM_RETURN CSSMTPI TP_CertSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertTemplateToBeSigned,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *SignerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR SignerVerifyResult,
    CSSM_DATA_PTR SignedCert)
```

DESCRIPTION

The TP module decides whether the signer certificate is trusted to sign the *CertTemplateToBeSigned*. 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. The caller must provide a credential that permits the caller to use the private key for this signing operation. The credential can be provided in the cryptographic context *CCHandle*. If *CCHandle* is NULL, the credentials in the *SignerVerifyContext* specify the credential value.

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. If the trust policy module does not assume defaults or the default CSP is not available on the local system, an error occurs.

CertTemplateToBeSigned (input)
A pointer to a structure containing a certificte template to be signed. The CRL type and encoded are included in this structure.

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/optional)
A structure containing credentials, policy information, and contextual information to be used in the verification process. All of the input values in the context are optional. The service provider can define default values or can attempt to operate without input for all the other fields of this input structure. The operation can fail if a necessary input value is omitted and the service module can not define an appropriate default value.

SignerVerifyResult (output/optional)
A pointer to a structure containing information generated during the verification process. The information can include:

Evidence	(output/optional)
NumberOfEvidences	(output/optional)

SignedCert (output)
A pointer to the CSSM_DATA structure containing the signed certificate. The *SignedCert→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CONTEXT_HANDLE
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_UNKNOWN_FORMAT
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST

CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE

SEE ALSO

For the CSSM API:
CSSM_TP_CertCreateTemplate( )
CSSM_TP_CrlSign( )

For the TP SPI:
TP_CertCreateTemplate( )
TP_CrlSign( )

NAME

CSSM_TP_CrlVerify for the CSSM API
TP_CrlVerify for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CrlVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ENCODED_CRL *CrlToBeVerified,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *VerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult)

SPI:
CSSM_RETURN CSSMTPI TP_CrlVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ENCODED_CRL *CrlToBeVerified,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *VerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult)
```

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.

The caller can specify additional points of trust represented by anchor certificates in the *VerifyContext*. The trust policy module can use these additional points of trust in the verification 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 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.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing a signed certificate revocation list to be verified. The CRL type and encoding are included in this structure.

SignerCertGroup (input)

A pointer to the CSSM_CERTGROUP structure containing one or more related certificates that paretially 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 — the caller can specify additional points of trust represented by anchor certificates in the *VerifyContext*. The trust policy module can use these additional points of trust in the verification process.

VerifyContext (input/optional)

A structure containing credentials, policy information, and contextual information to be used in the verification process. All of the input values in the context are optional. The service provider can define default values or can attempt to operate without input for all the other fields of this input structure. The operation can fail if a necessary input value is omitted and the service module can not define an appropriate default value

RevokerVerifyResult (output/optional)

A pointer to a structure containing information generation during the verification process. The information can include:

Evidence	(output/optional)
NumberOfEvidences	(output/optional)

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_CRL_TYPE
CSSMERR_TP_INVALID_CRL_ENCODING
CSSMERR_TP_INVALID_CRL_POINTER
CSSMERR_TP_INVALID_CRL
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER

CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlVerify()

For the TP SPI:
CL_CrlVerify()

NAME

CSSM_TP_CrlCreateTemplate for the CSSM API
TP_CrlCreateTemplate for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CrlCreateTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlFields,
    CSSM_DATA_PTR NewCrlTemplate)

SPI:
CSSM_RETURN CSSMTPI TP_CrlCreateTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlFields,
    CSSM_DATA_PTR NewCrlTemplate)
```

DESCRIPTION

This function creates an unsigned, memory-resident CRL template. Fields in the CRL are initialized based on the descriptive data specified by the OID/value input pairs in *CrlFields* and the local domain policy of the TP. 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. The *NewCrlTemplate* output is an unsigned CRL template in the format supported by the TP.

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.

NumberOfFields (input)
The number of OID/value pairs specified in the *CrlFields* input parameter.

CrlFields (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields

NewCrlTemplate (output)
A pointer to the CSSM_DATA structure containing the new CRL. The *NewCrl→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_FIELD_POINTER

CSSMERR_TP_UNKNOWN_TAG
CSSMERR_TP_INVALID_NUMBER_OF_FIELDS

SEE ALSO

For the CSSM API:
CSSM_TP_CrlSignWithKey()
CSSM_TP_CrlSignWithCert()

For the TP SPI:
TP_CrlSignWithKey()
TP_CrlSignWithCert()

NAME

CSSM_TP_CertRevoke for the CSSM API
TP_CertRevoke for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertRevoke
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *OldCrlTemplate,
    const CSSM_CERTGROUP *CertGroupToBeRevoked,
    const CSSM_CERTGROUP *RevokerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *RevokerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult,
    CSSM_TP_CERTCHANGE_REASON Reason,
    CSSM_DATA_PTR NewCrlTemplate)
SPI:
CSSM_RETURN CSSMTPI TP_CertRevoke
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *OldCrlTemplate,
    const CSSM_CERTGROUP *CertGroupToBeRevoked,
    const CSSM_CERTGROUP *RevokerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *RevokerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult,
    CSSM_TP_CERTCHANGE_REASON Reason,
    CSSM_DATA_PTR NewCrlTemplate)
```

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.

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.

OldCrlTemplate (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.

CertGroupToBeRevoked (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.

RevokerVerifyResult (output/optional)
A pointer to a structure containing information generated during the verification process. The information can include:

Evidence	(output/optional)
NumberOfEvidences	(output/optional)

Reason (input/optional)
The reason for revoking the subject certificate.

NewCrlTemplate (outputl)
A pointer to the CSSM_DATA structure containing the updated certificate revocation list. If the pointer is NULL, an error has occurred.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_CRL_POINTER
CSSMERR_TP_INVALID_CRL
CSSMERR_TP_UNKNOWN_FORMAT
CSSMERR_TP_CRL_ALREADY_SIGNED
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT

CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE
CSSMERR_TP_INVALID_REASON

SEE ALSO

For the CSSM API:
CSSM_CL_CrlAddCert()

For the TP SPI:
CL_CrlAddCert()

NAME

CSSM_TP_CertRemoveFromCrlTemplate for the CSSM API
TP_CertRemoveFromCrlTemplate for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertRemoveFromCrlTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *OldCrlTemplate,
    const CSSM_CERTGROUP *CertGroupToBeRemoved,
    const CSSM_CERTGROUP *RevokerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *RevokerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult,
    CSSM_DATA_PTR NewCrlTemplate)

SPI:
CSSM_RETURN CSSMTPI TP_CertRemoveFromCrlTemplate
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA *OldCrlTemplate,
    const CSSM_CERTGROUP *CertGroupToBeRemoved,
    const CSSM_CERTGROUP *RevokerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *RevokerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR RevokerVerifyResult,
    CSSM_DATA_PTR NewCrlTemplate)
```

DESCRIPTION

The TP module determines whether the revoking certificate group can remove the subject certificate group from the CRL template. The revoker certificate group is first authenticated and its applicability to perform this operation is determined. Once the trust is established, the TP removes the certificates from the CRL template.

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.

OldCrlTemplate (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.

CertGroupToBeRemoved (input)
A group of one or more certificates to be removed from the the CRL template.

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.

RevokerVerifyResult (output/optional)
A pointer to a structure containing information generated during the verification process. The information can include:

Evidence (output/optional)

NumberOfEvidences (output/optional)

NewCrlTemplate (output)
A pointer to the CSSM_DATA structure containing the updated certificate revocation list. If the pointer is NULL, an error has occurred.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_CRL_POINTER
CSSMERR_TP_INVALID_CRL
CSSMERR_TP_UNKNOWN_FORMAT
CSSMERR_TP_CRL_ALREADY_SIGNED
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS

CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlAddCert( )

For the TP SPI:
CL_CrlAddCert( )

NAME

CSSM_TP_CrlSign for the CSSM API
TP_CrlSign for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CrlSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_ENCODED_CRL *CrlToBeSigned,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *SignerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR SignerVerifyResult,
    CSSM_DATA_PTR SignedCrl)

SPI:
CSSM_RETURN CSSMTPI TP_CrlSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_ENCODED_CRL *CrlToBeSigned,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *SignerVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR SignerVerifyResult,
    CSSM_DATA_PTR SignedCrl)
```

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. The caller must provide a credential that permits the caller to use the private key for this signing operation. The credential can be provided in the cryptographic context *CCHandle*. If *CCHandle* is NULL, the credentials in the *SignerVerifyContext* specify the credential value.

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.

CrlToBeSigned (input)
A pointer to the CSSM_DATA structure containing a certificate revocation list to be signed.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related 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.

SignerVerifyContext (input/optional)
A structure containing credentials, policy information, and contextual information to be used in the verification process. All of the input values in the context are optional. The service provider can define default values or can attempt to operate without input for all the other fields of this input structure. The operation can fail if a necessary input value is omitted and the service module can not define an appropriate default value.

SignerVerifyResult (output/optional)
A pointer to a structure containing information generation during the verification process. The information can include:

Evidence	(output/optional)
NumberOfEvidences	(output/optional)

SignedCrl (output)
A pointer to the CSSM_DATA structure containing the signed certificate revocation list. The *SignedCrl→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE
A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS
See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CONTEXT_HANDLE
CSSMERR_TP_INVALID_CRL_TYPE
CSSMERR_TP_INVALID_CRL_ENCODING
CSSMERR_TP_INVALID_CRL_POINTER
CSSMERR_TP_INVALID_CRL
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE

CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED
CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlSign( )

For the TP SPI:
CL_CrlSign( )

NAME

CSSM_TP_ApplyCrlToDb for the CSSM API
TP_ApplyCrlToDb for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_ApplyCrlToDb
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ENCODED_CRL *CrlToBeApplied,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *ApplyCrlVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR ApplyCrlVerifyResult)

SPI:
CSSM_RETURN CSSMTPI TP_ApplyCrlToDb
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_ENCODED_CRL *CrlToBeApplied,
    const CSSM_CERTGROUP *SignerCertGroup,
    const CSSM_TP_VERIFY_CONTEXT *ApplyCrlVerifyContext,
    CSSM_TP_VERIFY_CONTEXT_RESULT_PTR ApplyCrlVerifyResult)
```

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. If optional, the caller will set this value to 0.

CrlToBeApplied (input)
A pointer to a structure containing the encoded certificate revocation list to be applied to the data store. The CRL type and encoding are included in this structure.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related 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.

ApplyCrlVerifyContext (input/optional)
A structure containing credentials, policy information, and contextual information to be used in the verification process. All of the input values in the context are optional. The service provider can define default values or can attempt to operate without input for all the other fields of this input structure. The operation can fail if a necessary input value is omitted and the service module can not define an appropriate default value.

ApplyCrlVerifyResult (output/optional)
A pointer to a structure containing information generated during the verification process. The information can include:

Evidence (output/optional)
NumberOfEvidences (output/optional)

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_CRL_TYPE
CSSMERR_TP_INVALID_CRL_ENCODING
CSSMERR_TP_INVALID_CRL_POINTER
CSSMERR_TP_INVALID_CRL
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_INVALID_ACTION
CSSMERR_TP_INVALID_ACTION_DATA
CSSMERR_TP_VERIFY_ACTION_FAILED
CSSMERR_TP_INVALID_CRLGROUP_POINTER
CSSMERR_TP_INVALID_CRLGROUP
CSSMERR_TP_INVALID_CRL_AUTHORITY
CSSMERR_TP_INVALID_CALLERAUTH_CONTEXT_POINTER
CSSMERR_TP_INVALID_POLICY_IDENTIFIERS
CSSMERR_TP_INVALID_TIMESTRING
CSSMERR_TP_INVALID_STOP_ON_POLICY
CSSMERR_TP_INVALID_CALLBACK
CSSMERR_TP_INVALID_ANCHOR_CERT
CSSMERR_TP_CERTGROUP_INCOMPLETE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_AUTHENTICATION_FAILED
CSSMERR_TP_INSUFFICIENT_CREDENTIALS
CSSMERR_TP_NOT_TRUSTED

CSSMERR_TP_CERT_REVOKED
CSSMERR_TP_CERT_SUSPENDED
CSSMERR_TP_CERT_EXPIRED
CSSMERR_TP_CERT_NOT_VALID_YET
CSSMERR_TP_INVALID_CERT_AUTHORITY
CSSMERR_TP_INVALID_SIGNATURE
CSSMERR_TP_INVALID_NAME
CSSMERR_TP_CERTIFICATE_CANT_OPERATE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstItem()
CSSM_CL_CrlGetNextItem()
CSSM_DL_CertRevoke()

For the TP SPI:
CL_CrlGetFirstItem()
CL_CrlGetNextItem()
DL_CertRevoke()

8.8 Group Functions

The man-page definitions for TP Group functions are presented in this section.

NAME

CSSM_TP_CertGroupConstruct for the CSSM API
TP_CertGroupConstruct for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertGroupConstruct
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST *DBList,
    const void *ConstructParams,
    const CSSM_CERTGROUP *CertGroupFrag,
    CSSM_CERTGROUP_PTR *CertGroup)

SPI:
CSSM_RETURN CSSMTPI TP_CertGroupConstruct
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST *DBList,
    const void *ConstructParams,
    const CSSM_CERTGROUP *CertGroupFrag,
    CSSM_CERTGROUP_PTR *CertGroup)
```

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 output set is a sequence of certificates ordered by the relationship among them. The result set can be augmented by adding semantically-related certificates obtained by searching the certificate data stores specified in *DBList*. The data stores are searched in order of appearance in *DBList*. If the TP supports a hierarchical model of certificates, the function output is an uninterrupted, ordered chain of certificates based on the first certificate as the leaf of the certificate chain. If the certificate is multiply-signed, then the ordered chain will follow the first signing certificate. The function should also detect cross-certificate pairs and should include both certificates without duplicating either certificate.

Extraneous certificates in the *CertGroupFrag* fragment or contained in the *DBList* data stores are ignored. The certificate group returned by this function can be used as input to the function *CSSM_TP_CertGroupVerify*( ) (CSSM API) or *TP_CertGroupVerify*( ) (TP SPI).

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 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, identifying certificate databases containing certificates (and possibly other security objects) that are managed by that module. certificates (and possibly other security objects). The data stores should be searched to complete construction of a semantically-related certificate group.

ConstructParams (input/optional)
A pointer to data that can be used by the add-in trust policy module in constructing the *CertGroup.The*semantics*of*this the trust policy and the credential model supported by that policy. The input parameter can consist of a set of values, each guiding some aspect of the construction process. Parameter values can:

- Limit the certificates that are added to the constructed set.
- Identify other sources of certificates for inclusion in the constructed set.

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. Subsequent intermediate certificates can be supplied by the caller. They need not be in any particular order.

CertGroup (output)
A pointer to a complete certificate group based on the original subset of certificates and the certificate data stores. The CSSM_CERTGROUP and its sub-structure is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CSP_HANDLE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP

CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_CERTGROUP_INCOMPLETE

SEE ALSO

For the CSSM API:
CSSM_TP_CertGroupPrune()
CSSM_TP_CertGroupVerify()

For the TP SPI:
TP_CertGroupPrune()
TP_CertGroupVerify()

NAME

CSSM_TP_CertGroupPrune for the CSSM API
TP_CertGroupPrune for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertGroupPrune
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DL_DB_LIST *DBList,
    const CSSM_CERTGROUP *OrderedCertGroup,
    CSSM_CERTGROUP_PTR *PrunedCertGroup)

SPI:
CSSM_RETURN CSSMTPI TP_CertGroupPrune
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DL_DB_LIST *DBList,
    const CSSM_CERTGROUP *OrderedCertGroup,
    CSSM_CERTGROUP_PTR *PrunedCertGroup)
```

DESCRIPTION

This function removes any locally issued anchor certificates from a constructed 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 *CSSM_TP_CertGroupConstruct*( ) (CSSM API) ot *TP_CertGroupConstruct*( ) (TP SPI) operation. The pruned certificate group should be suitable for export to external hosts/entities, 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, identifying certificate databases containing certificates (and possibly other security objects) that are managed by that module. 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 semantically-related certificates — for example, the result of a *CSSM_TP_CertGroupConstruct*( ) (CSSM API) or *TP_CertGroupConstruct*( ) (TP SPI) call — from which certificates will be selectively removed.

PrunedCertGroup (output)
A pointer to a certificate group containing those certificates which are verifiable credentials outside of the local system. The CSSM_CERTGROUP and its sub-structure is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP
CSSMERR_TP_INVALID_CERTIFICATE
CSSMERR_TP_CERTGROUP_INCOMPLETE

SEE ALSO

For the CSSM API:
CSSM_TP_CertGroupConstruct( )
CSSM_TP_CertGroupVerify( )

For the TP SPI:
TP_CertGroupConstruct( )
TP_CertGroupVerify( )

NAME

CSSM_TP_CertGroupToTupleGroup for the CSSM API
TP_CertGroupToTupleGroup for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_CertGroupToTupleGroup
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE  CLHandle,
    const CSSM_CERTGROUP *CertGroup,
    CSSM_TUPLEGROUP_PTR *TupleGroup)

SPI:
CSSM_RETURN CSSMTPI TP_CertGroupToTupleGroup
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE  CLHandle,
    const CSSM_CERTGROUP *CertGroup,
    CSSM_TUPLEGROUP_PTR *TupleGroup)
```

DESCRIPTION

This function creates a set of authorization tuples based on a set of input certificates. The certificates must be of the type managed by the Trust Policy module. The trust policy module may require that the input certificates be successfully verified before being translated to tuples. It is assumed that the certificates carry authorizations. The trust policy service provider interprets the certificate authorization fields and generates one or more tuples corresponding to those authorizations. The certificates of the type managed by the Trust Policy module. The resulting tuples can be input to an authorization evaluation function, such as *CSSM_AC_AuthCompute*() (CSSM API) or *AC_AuthCompute*() (AC SPI), which determines whether a particular action is authorized under a basic set of authorization assumptions.

PARAMETERS

TPHandle (input)
The handle that describes the trust policy service module used to perform this function.

CLHandle (input/optional)
The handle that describes the certificate library module that can be used to scan the certificate fields for values. If no certificate library module is specified, the TP module uses an assumed CL module.

CertGroup (input)
A group of certificates in the native certificate format supported by the Trust Policy module. The certificates carry authorizations for one or more certificate subjects.

TupleGroup (output)
A pointer to a structure containing references to one or more tuples resulting from the translation process. Storage for structure and the tuples is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CERTGROUP_POINTER
CSSMERR_TP_INVALID_CERTGROUP

SEE ALSO

For the CSSM API:
CSSM_TP_TupleGroupToCertGroup( )
CSSM_AC_AuthCompute( )

For the TP SPI:
TP_TupleGroupToCertGroup( )
AC_AuthCompute( )

NAME

CSSM_TP_TupleGroupToCertGroup for the CSSM API
TP_TupleGroupToCertGroup for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_TupleGroupToCertGroup
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE  CLHandle,
    const CSSM_TUPLEGROUP *TupleGroup,
    CSSM_CERTGROUP_PTR *CertTemplates)

SPI:
CSSM_RETURN CSSMTPI TP_TupleGroupToCertGroup
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE  CLHandle,
    const CSSM_TUPLEGROUP *TupleGroup,
    CSSM_CERTGROUP_PTR *CertTemplates)
```

DESCRIPTION

This function creates a set of certificate templates based on a set of input tuples. The tuples describe a set of authorizations for one or more subjects. The trust policy service provider maps these authorizations to appropriate template values for one or more certificates of the type managed by the Trust Policy module. The resulting certificate templates can be input to a certificate creation function, such as *CSSM_CL_CertSign*() (CSSM API) or *CL_CertSign*() (TP SPI). The signed certificates created by these functions should carry the authorizations described in the input tuples.

PARAMETERS

TPHandle (input)
The handle that describes the trust policy service module used to perform this function.

CLHandle (input/optional)
The handle that describes the certificate library module that can be used to assist in the creation of field values. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

TupleGroup (input)
A pointer to a group of CSSM_TUPLE describing authorizations for one or more subjects.

CertTemplates (output)
A pointer to a structure containing references to one or more certificate templates resulting from the translation process. Storage for the structure and certificate templates is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_TUPLEGROUP_POINTER
CSSMERR_TP_INVALID_TUPLEGROUP
CSSMERR_TP_INVALID_TUPLE

SEE ALSO

For the CSSM API:
CSSM_TP_CertGroupToTupleGroup( )
CSSM_AC_AuthCompute( )

For the TP SPI:
TP_CertGroupToTupleGroup( )
AC_AuthCompute( )

8.9 Extensibility Functions

The man-page definition for the *CSSM_TP_PassThrough*( ) Extensibility function is presented in this section.

NAME

CSSM_TP_PassThrough for the CSSM API
TP_PassThrough for the TP SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_TP_PassThrough
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST *DBList,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)

SPI:
CSSM_RETURN CSSMTPI TP_PassThrough
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST *DBList,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)
```

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.

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

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store, identifying certificate databases containing certificates (and possibly other security objects) that may be used by the *pass-through* function. 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/optional)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested TP module.

OutputParams (output/optional)
A pointer to a module, implementation-specific structure containing the output data. The service provider allocates the memory for sub-structures. The application must free the memory for the sub-structures.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_TP_INVALID_CL_HANDLE
CSSMERR_TP_INVALID_CONTEXT_HANDLE
CSSMERR_TP_INVALID_DL_HANDLE
CSSMERR_TP_INVALID_DB_HANDLE
CSSMERR_TP_INVALID_DB_LIST_POINTER
CSSMERR_TP_INVALID_DB_LIST
CSSMERR_TP_INVALID_PASSTHROUGH_ID

Technical Standard

Part 5:

Authorization Computation (AC) Services

The Open Group

Chapter 9

Authorization Computation Services

9.1 Overview

An application or service that applies access control is typically faced with a request of the form:

"I am subject S. Do X for me."

The code performing the access control needs to answer two questions before performing this action:

1. Is this subject S?
2. Is S allowed to do X?

The first is called **authentication**. The second is called **authorization**.

There are many forms of authentication. These forms and the mechanisms that support them are covered in other sections of this specification.

9.2 Authorization, Certificates, and Credentials

No authorization decision can be traced back to a universal truth. Instead, it must be based on some assumption(s). Each assumption is formalized in an Access Control List (ACL) structure. An ACL is a list of subjects allowed to have some particular access to some resource. A requester presents an exhibit corresponding to the subject of an ACL. The exhibit can be matched with the subject of the ACL, which grants rights to the subject. Certificates, exhibits, other forms of credentials, and ACLs (which are non-signed credentials) can all be involved in the authorization decision.

9.2.1 Classes of Certificates and Other Credentials

Loren Kohnfelder defined a certificate, in 1978, as a digitally signed data record binding a name to a public key. Since then, other forms of certificate have been defined. We define three categories of data that may be contained in a certificate:

- Names
- Keys
- Permissions (or authorizations, attributes, etc.)

A permission, X, records a keyholder's permission or authority to perform some action on a given resource and potentially also the permission to delegate that permission to others. The key, S, is the means by which the keyholder authenticates himself to the entity doing the verification. The name, N, is some text string that people find useful in referring to the keyholder. These three items of information can be related in pairs. There are credentials of various kinds that express each of these pairs, as shown in Figure 9-1.

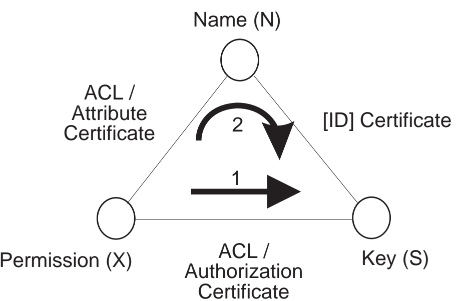


Figure 9-1 Credential Classes

To answer the authorization question:

"Is S allowed to do X?"

we need the relation found on the base of the triangle. That is, we need the permission, X, to be associated with a key, S. The permission speaks to the question being asked and the key is the mechanism by which a remote entity is verified. There are two ways to connect S and X, shown in Figure 9-1 as arrows: (1) directly, and (2) via names.

Direct authorization

Direct authorization is a mechanism found in a few credential types. There are both ACL and certificate forms for these credentials, depending on whether the credential is retained in a trusted memory at the verifier or must be protected from tampering while being held elsewhere, respectively.

Direct authorization (X.S) is used in the certificate definitions of SPKI, SET, and KeyNote. It is also found in the ACL definitions of SSH, AADS and X9.59. It is also assumed, implicitly, in SSL.

Authorization via Name

When using a name certificate, the permission, X, flows around the top of the triangle shown in through an ACL entry to empower a name, N, and then down an ID certificate to a key, S. This method of operation remains possible.

In the early 1990's, the X9 community defined an attribute certificate, which maps some attribute, X, to a name, N. The X.509 community is now adopting attribute certificates as well. With an attribute certificate, the permission flow is X.N.S. This means there are two certificates to be validated, not just one.

9.2.2 Credential Format Options

For each of the three sides in the triangle shown in Figure 9-1, it is possible to define both ACL and certificate mechanisms. An ACL differs from a certificate, in that it is not digitally signed and has no issuer.

In general, the data items useful in building credentials are (see Figure 9-1):

I Issuer - the signer of a certificate

N Name - the name assigned by the issuer to a keyholder, when that name is being defined in the credential, rather than used as a subject

S Subject - the subject of the credential: a public key or a name for a public key. Other subjects are possible but need not be considered for the purposes of this specification

D Delegation - the permission to delegate the permissions contained in the credential

X Permission - the permission granted by this credential

V Validity conditions - the conditions under which the credential is valid, typically not-before and not-after dates, but also possibly instructions to perform some on-line test such as a CRL.

Using these data items, different types of credentials can be constructed. Different credential types are fit for different purposes. The following is a non-exhaustive list of useful credential types. The types are defined by the data items they contain <I,N,S,D,X,V>. If a credential does not contain an issuer, then it is assumed that it is an ACL form.

X→N ACL (-,-,S,D,X,-)

This describes a traditional ACL entry. The subject, S, is assumed here to be a name for a key that will be used for authentication or verification. There is no standard for constructing an ACL. In general, ACL formats do not need to be standardized since an ACL is not transmitted between machines. This specification defines a structure for ACL entries so they can be passed across the CDSA interfaces on a single system.

X→N Attribute Certificate (I,-,S,D,X,V)

This describes an attribute certificate as standardized in the X9 community and now in X.509. There is also an attribute certificate standard under SPKI. In all of these cases, the subject, S, is a name (expected to be the name of some key that will be used in verification).

N→S ID Certificate (I,N,S,-,-,V)

This describes the standard ID certificate. Examples include:

- X.509, and its derivative forms
- PGP
- SDSI 1.0"
- SPKI (after the merger with SDSI)"

X→S ACL (-,-,S,D,X,-)

This describes a direct authorization ACL. Examples include:

- SSH, in the file ˜/.ssh/authorized_keys
- AADS
- X9.59
- The list of root keys for SSL, maintained in any browser capable of SSL

X→S Authorization Certificate (I,-,S,D,X,V)

This describes direct authorization certificates. Examples include:

- SPKI (as originally designed)
- KeyNote
- X.509 SSL - in which the permission is implicitly "(*)" - it is not possible to delegate a subset of the permissions.
- X.509 SET

9.2.3 Logic of Authorization

There are two methods for computing an authorization decision:

- Direct authorization based on ACLs and certificates
- Indirect authorization via names

Direct, Delegated Authorization

An authorization must be made with reference to at least one credential, an ACL entry. The ACL serves as an anchor for the authorization. It is required since there is no absolute truth or absolutely powerful public key on which to base a decision. Rather, there are only assumptions, sometimes called executable policies, on the basis of which logical inferences can be made.

All authorizations can be made by ACL, but the resulting ACL might get very large. Alternatively, the ACL can grant delegation permission to a key that can then issue certificates to cover what would have been a large set of ACL entries.

The authorization certificates thus generated may carry implicit or explicit permission. The intermediate form (TUPLE) used in *AuthCompute* carries explicit permission. If permission is implicit, then it must have been made explicit in translation to TUPLE.

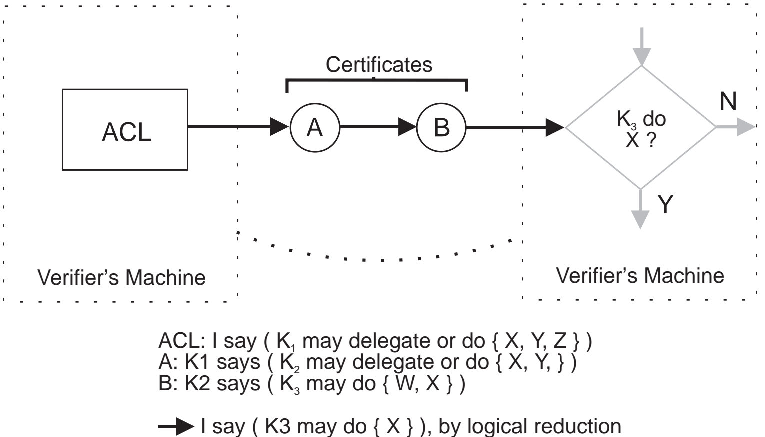


Figure 9-2 Logic of Authorization

In the example, the ACL grants three kinds of permission ("X, Y, Z") to public key K1. Each permission is expressed as a list of parameters, each being a byte string or a sublist. Most ACLs grant only a single permission, but multiple permissions are possible.

The ACL states that key K1 has those three permissions plus the permission to delegate them. K1 delegates permissions X and Y to key K2 by issuing a certificate A with those rights to the key K2. In turn, K2 issues a certificate, B, granting permissions W and X to key K3. Permission W is not relevant to this computation, but is shown here to point out that no intermediate issuer can get away with granting extra permissions. The certificate B does not give permission to delegate, therefore certificate B must always be right-most in a chain of certificates. The logical reduction of this chain of certificates is that the verifier (the agent running the access control test and holding the ACL) concludes that it knows that K3 is permitted to do X. Since that is what the condition in Figure 9-2 on page 410 is testing, this authorization test succeeds.

The authorization computation function *CSSM_AC_AuthCompute*( ) provide resource managers with a basic authorization computation enginer for making authorization decisions.

Authorization via Names

An authorization computation can also be performed by traversing path 2 in the triangle. This requires a pair of credentials:

- An attribute certificate that binds a permission to a name
- An ID certificate that binds a name to a key

This pair can exist as two certificates or as an ACL entry (giving permission to a name) and an ID certificate.

For use in *AuthCompute*, names are assumed to be fully qualified: that is, they are made globally unique by inclusion of the public key (or its hash) of the namespace that issues the ID certificates defining that name. As an example, name K1 "Fred Jones", where K1 is the hash of a public key, is a fully qualified name.

9.2.4 Authorization Reduction Process

The first step in performing an authorization computation is to replace name references with public keys. This is achieved by applying all name definitions with keys for subjects to all intermediate forms with that name. For example, if there is a TUPLE with a subject of the form (name K_1 "Fred Jones") and there is a name certificate TUPLE <K_1, "Fred Jones #53486", K_2, -, -, V>, then that subject is replaced by K_2. The validity date ranges of the two TUPLES involved are also intersected.

The name reduction process produces an authorization 5-tuples, mapping permissions to keys. The reduction rule for combining all of the input credentials is:

<I1,S1,D1,X1,V1> <I2,S2,D2,X2,V2> → <I1,S2,D2,auth_intersect(X1,X2),date_intersect(V1,V2)>

provided D1=true, S1=I2, and the two intersections are non-empty.

Date intersection is based on comparing not-before and not-after dates and times expressed as ASCII strings of the form 1999-04-24_23:57:39. If the not-before date is greater than the not-after date, then the intersection is empty.

The authorization intersection operation is rigidly defined. The authorizations X and Y are non-empty lists such that the first element is a byte string, called the name or type of the list. The intersection of two lists is computed by pairwise-intersecting each element in the list. List elements can be either byte strings or non-empty sublists. Byte strings intersect only if they are equal. Sublists are intersected by applying the rules for lists to each sublist. If any intersection is empty, then the pair of tuples under consideration can not be reduced. If two lists are of different length but intersect in a non-null set of common elements, then the intersection includes all the elements of the longer list.

Authorization computation based on reductions is defined only when using a set of CSSM_TUPLE structures. Other forms of credentials, such as X.509 certificates, PGP certificates, etc. can be converted to a set of CSSM_TUPLE structures. Translations are performed by using the appropriate Trust Policy service provider and invoking the function *CSSM_TP_CertGroupToTupleGroup*(). The resulting tuples can be used as input to the *AuthCompute* service. The inverse translation is provided by the appropriate Trust Policy service provider by invoking the function *CSSM_TP_TupleGroupToCertGroup*().

9.2.5 Example Authorization Request

Consider a web server that needs to restrict access to its contents. The server would manage one or more ACLs to define the base authorizations of the system. For instance, the server might have an ACL that says that Bob can access all data in the subtree http://www.bob.com/sensitiveData. Furthermore, Bob can delegate this authorization to other users by signing authorization certificates for them. Thus, the base authorization ACL held by the server might look something like this.

```
(
   (subject Key-Bob)
   (propagate)
   (tag (http (* prefix http://www.bob.com/sensitiveData)))
)
```

Bob delegates his permission to Alice by creating and signing an authorization certificate for her, as follows:

```
(
   (issuer Key-Bob)
   (subject Key-Alice)
   (tag (http (* prefix http://www.bob.com/sensitiveData/forAlice)))
)
```

Now when Alice wants to access the data on Bob's web server, she submits the certificate that was issued by Bob. The server challenges her to prove who she is by, for example, sending a large random number for her to digitally sign with her private key (matching Key-Alice) and return to the server. The server checks that signature, using Key-Alice, and if the signature is correct, the server knows that the entity making the request has control over the private key for Key-Alice.

The server then calls *AuthCompute* to determine whether Alice's request to access the data on Bob's web server should be honored. It provides the ACL controlling that resource (the base authorization), the certificates (credentials) Alice provided (if any) and a description of the current request. That description has three parts:

1. Requestors:
   the authentications that were performed - in this case, Key-Alice - in the form of one or more subject fields
2. RequestedAuthorizationPeriod:
   the current time or a time range for which the request is being made
3. RequestedAuthorization:
   the access being desired,
   e.g. (http http://www.bob.com/sensitiveData/forAlice/index.html)

The output of *AuthCompute* will be a set of TUPLEs, giving the intersection of ACL, certificates and requested authorization information, assuming all of the requestors have been

authenticated. These TUPLEs are in the form of ACL entries and can be:

- Examined to see if it is null - in which case the request was not authorized
- Added to the caller's ACL (if not null) so that the caller can make future requests without presenting certificates;
- Turned into a certificate and given back to the requesting party, if this one authorization certificate is smaller than the set of certificates originally provided.

9.3 Add-In Module

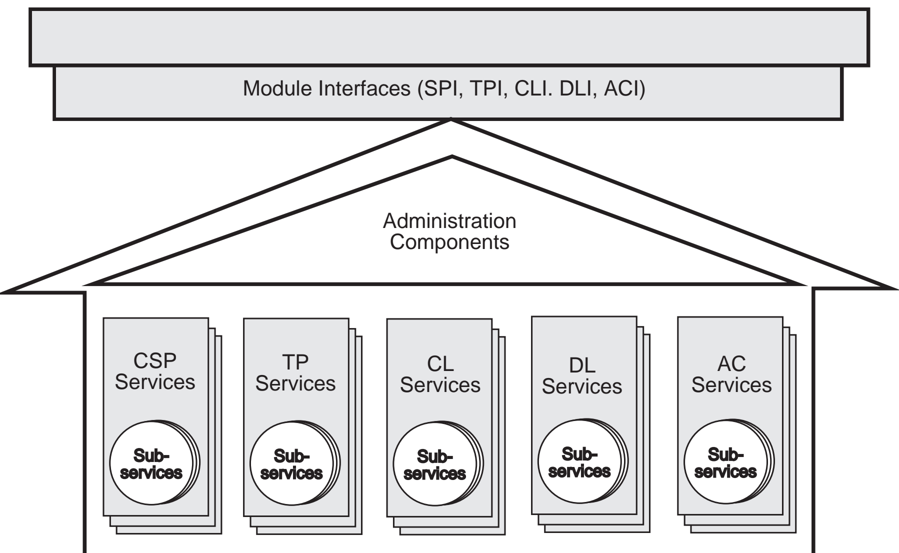


Figure 9-3 CDSA Add-In Module Structure

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. In this case it provides Authorization Computation (AC) services.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions that 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 **Part 14** of this Technical Standard.

9.4 Data Structures

9.4.1 CSSM_AC_HANDLE

This data structure represents the auth compute module handle. The handle value is a unique pairing between an auth compute module and an application that has attached that module. AC handles can be returned to an application as a result of the *CSSM_ModuleAttach*() function.

```
typedef CSSM_MODULE_HANDLE CSSM_AC_HANDLE
```

9.5 Error Codes and Error Values

The Error Values that can be returned by AC functions can be either derived from the Common Error Codes defined in or from a set of Errors that more than one AC function can return, or they are specific to an AC function.

9.5.1 AC Error Values Derived from Common Error Codes

```
#define CSSMERR_AC_INTERNAL_ERROR \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)
#define CSSMERR_AC_MEMORY_ERROR  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)
#define CSSMERR_AC_MDS_ERROR  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)
#define CSSMERR_AC_INVALID_POINTER  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)
#define CSSMERR_AC_INVALID_INPUT_POINTER  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)
#define CSSMERR_AC_INVALID_OUTPUT_POINTER  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)
#define CSSMERR_AC_FUNCTION_NOT_IMPLEMENTED  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)
#define CSSMERR_AC_SELF_CHECK_FAILED  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)
#define CSSMERR_AC_OS_ACCESS_DENIED  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)
#define CSSMERR_AC_FUNCTION_FAILED  \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)
#define CSSMERR_AC_INVALID_CONTEXT_HANDLE \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_CONTEXT_HANDLE)
#define CSSMERR_AC_INVALID_DATA \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_DATA)
#define CSSMERR_AC_INVALID_DB_LIST \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_LIST)
#define CSSMERR_AC_INVALID_PASSTHROUGH_ID \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_PASSTHROUGH_ID)
#define CSSMERR_AC_INVALID_DL_HANDLE \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_DL_HANDLE)
#define CSSMERR_AC_INVALID_CL_HANDLE \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_CL_HANDLE)
#define CSSMERR_AC_INVALID_TP_HANDLE \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_TP_HANDLE)
#define CSSMERR_AC_INVALID_DB_HANDLE \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_HANDLE)
#define CSSMERR_AC_INVALID_DB_LIST_POINTER \
    (CSSM_AC_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_LIST_POINTER)
```

9.5.2 AC Error Values

Values that can be returned by one or more AC APIs.

```
#define CSSM_AC_BASE_AC_ERROR \
    (CSSM_AC_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT)
```

```
#define CSSMERR_AC_INVALID_BASE_ACLS (CSSM_AC_BASE_AC_ERROR+1)
```

One (or more) of the base ACLs was not in proper ACL format

```
#define CSSMERR_AC_INVALID_TUPLE_CREDENTIALS (CSSM_AC_BASE_AC_ERROR+2)
```

One (or more) of the tuple credentials was not in proper CSSM_TUPLE format

```
#define CSSMERR_AC_INVALID_ENCODING (CSSM_AC_BASE_AC_ERROR+3)
```

One (or more) of the input parameters was not properly encoded

```
#define CSSMERR_AC_INVALID_VALIDITY_PERIOD (CSSM_AC_BASE_AC_ERROR+4)
```

The date and time descriptions provided were not properly encoded or did not specify a valid interval

```
#define CSSMERR_AC_INVALID_REQUESTOR (CSSM_AC_BASE_AC_ERROR+5)
```

One (or more) of the Requestors was not properly encoded

```
#define CSSMERR_AC_INVALID_REQUEST_DESCRIPTOR (CSSM_AC_BASE_AC_ERROR+6)
```

One (or more) of the base ACLs was not in proper ACL format

9.6 Authorization Computation Operations

The man-page definitions for Authorization Computation operations are presented in this section.

NAME

CSSM_AC_AuthCompute
AC_AuthCompute

SYNOPSIS

```
API:
CSSM_RETURN CSSMACI CSSM_AC_AuthCompute
    (CSSM_AC_HANDLE ACHandle,
    const CSSM_TUPLEGROUP *BaseAuthorizations,
    const CSSM_TUPLEGROUP *Credentials,
    uint32 NumberOfRequestors,
    const CSSM_LIST *Requestors,
    const CSSM_LIST *RequestedAuthorizationPeriod,
    const CSSM_LIST *RequestedAuthorization,
    CSSM_TUPLEGROUP_PTR AuthorizationResult)

SPI:
CSSM_RETURN CSSMACI AC_AuthCompute
    (CSSM_AC_HANDLE ACHandle,
    const CSSM_TUPLEGROUP *BaseAuthorizations,
    const CSSM_TUPLEGROUP *Credentials,
    uint32 NumberOfRequestors,
    const CSSM_LIST *Requestors,
    const CSSM_LIST *RequestedAuthorizationPeriod,
    const CSSM_LIST *RequestedAuthorization,
    CSSM_TUPLEGROUP_PTR AuthorizationResult)
```

DESCRIPTION

This function performs an authorization computation and returns the results as a group of tuple-certificates. The computation is based on five input values:

- *Requestors* - one or more items that identify the requestor. These items are matched against subject fields in the *BaseAuthorizations* or *Credentials*. These will be of any form that occurs in an ACL or certificate, and the class of entries is extensible. *AuthCompute* uses these fields to compare against *Subject* fields of TUPLES but does not interpret them, so it does not need to be aware of these extensions. *Requestors*, taken together with *RequestedAuthorization* and *RequestedAuthorizationPeriod*, form request tuples of the form "who requests what, when." *Requestors* can be public keys that verify some signed request, hashes of objects submitted for proof of permission, etc. In general, there will be only one *Requestor*, typically the public key of some keyholder signing a request or authenticating a connection.
- *RequestedAuthorization* - the authorization against which the *Requestors* are being tested in this computation.
- *RequestedAuthorizationPeriod* - the time range of an authorization computation.
- *BaseAuthorizations* - the group of ACL entries (unsigned certificates) provided as the basis for this computation.
- *Credentials* - a group of tuple-certificates used with the *BaseAuthorizations* to grant authorizations to the *Requestors*.

KIND OF SUBJECT	EXAMPLE REQUESTOR
Public key	(public-key (rsa-pkcs1-sha1 (e #03#) (n ##)))
Hash of object, key, template, etc.	(hash md5 #900150983cd24fb0d6963f7d28e17f72#)

The most likely *Requestor* is a public key that signs a request. In common practice there will be one Requestor per computation, but it is possible for an ACL or certificate to require multiple signatures or other forms of identification before an action is authorized. In that case, there must be multiple *Requestors*. This function can be used in two modes:

- To verify the authorization of a specific request, backed up by specific Requestors
- To compute the set of authorizations that a particular set of *Requestors* has been granted by the *BaseAuthorizations* and *Credentials*.

When using this function to verify an authorization, the *RequestedAuthorization* is the specific authorization being requested and the *RequestedAuthorizationPeriod* gives the date and time of that request (typically the current date and time) using both NOT_BEFORE and NOT_AFTER dates. The result, if any, should be an ACL entry with the same authorization that was requested. If such an ACL entry is produced by the computation, then the request is authorized.

EXAMPLE REQUESTED AUTHORIZATIONS
(http http://private.jf.intel.com/local-data.html)
(ftp ftp://private.jf.intel.com/users/cme/private/test.txt write)
EXAMPLE REQUESTED AUTHORIZATION PERIOD
(valid (not-before "1999-07-28_17:00:44") (not-after "1999-07-28_17:00:44"))

When using this function to compute the full set of possible authorizations from a set of credentials, rather than to verify a specific access request, the inputs should be of the following form:

- *RequestedAuthorizationPeriod* is either an empty list or the list"*valid*", indicating "all time".
- *RequestedAuthorization* is the list "*", indicating all possible authorizations.

The result of this computation, if any, will be one or more ACL entries representing all the granted authorizations for the indicated Requestor(s).

The scope of ACLs output from this function is limited to the local system. Each ACL should be interpreted to mean: "for this machine, under these base authorization ACLs and the provided certificates, relative to the specified requestors, the following authorizations have been deduced". Those authorizations are available only on the current platform (and possibly only for the application providing the ACL), and are therefore in the form of an ACL. They are not intended to be used by any other machine or application instance. However, the resulting ACLs can be transferred and used outside of the local scope by an entity with authority in the target scope/environment. The transfer and use is a three step process:

- Convert the ACL into one or more certificates - the certificates must be signed by some private key with appropriate authority in the target scope/environment.
- Transfer the certificates to the target environment.
- Use the signed certificates as input *Credentials* to this function in the target scope/environment.

If the function is successful, check *(*AuthorizationResult)→NumCerts* to determine the precise number of authorizations granted by this computation. If 0, then the *requestors* were not authorized.

PARAMETERS

ACHandle (input)
The handle that describes the authorization computation module used to perform this function.

BaseAuthorizations (input)
A pointer to a CSSM_TUPLEGROUP containing at least one ACL certificate, specifying the authorization granted to certain root keys, named entities or combinations thereof. A NULL group of *BaseAuthorizations* always results in a NULL *AuthorizationResult*.

Credentials (input/optional)
A pointer to a CSSM_TUPLEGROUP containing a group of certificates, in TUPLE form. The tuple-certificates define the delegation of authorizations from the *BaseAuthorizations* to the *Requestors*. If no additional authorization-granting tuples are provided, then this value is NULL and the *BaseAuthorizations* are the only source of trusted authorizations used as input to the authorization computation.

NumberOfRequestors (input)
The number of entries in the *Requestors* array.

Requestors (input)
A pointer to a list of requestors that define the "who" portion of the request. The list can be of type CSSM_LIST_TYPE_SEXPR. Typical exhibits include:

- Public keys
- Hashes of keys
- Hashes of other objects offered for proof.

RequestedAuthorizationPeriod (input/optional)
A list defining a validity period or NULL (implying "all time"). This is the "when" portion of the request.

If the list of is type CSSM_LIST_TYPE_SEXPR, then the validity interval is specified as a two-element list containing the values ((not-before <date1>)(not-after <date2>)). Note that each element is a two-element sublist. The <date> is represented by an ASCII byte-string, in the format (for example) "1998-11-24_15:06:16" and is assumed to be GMT. Open-ended time intervals are specified by omitting either of the interval ends. For example, ((not-before 1997-1-1_00:00:0)) specifies all dates and times beginning on January 1, 1997 going forward indefinitely. For programming convenience, when testing for authorization at a single point in time, the date is represented by a one-element list containing (<date>).

RequestedAuthorization (input)
A list defining the "what" portion of the authorization being requested.

If the list is of type CSSM_LIST_TYPE_SEXPR, then the list presents an authorization request in SPKI format. If a specific authorization is being requested, then this input is a two-element SEXPR list containing (tag <req>). The valid values for <req> are application-specific. If this is a request to derive all possible authorizations based on the *BaseAuthorizations*, *Credentials*, and *Requestors*, then this input value must be the two-element list containing (tag (*)). This list corresponds to "all authorizations". With this input, the function tests the provided ACL and certificates against the *Requestors* (and possibly *RequestedAuthorizationPeriod*) to yield all authorizations for which the provided *Exhibits* qualify.

AuthorizationResult (output)
A CSSM_TUPLEGROUP structure, giving the result of the authorization computation.

Typically there will be one result, but there could be as many as there are entries in the *BaseAuthorizations*. Each of these results says, in effect: "for this machine, under this ACL and the provided certificates, relative to the specified *Requestors*, the following authorizations have been deduced". Those authorizations are available only on the current platform (and possibly only for the application providing the ACL), and are therefore in the form of an ACL. They are not intended to be used by any other machine or application instance, necessarily, and need to be converted into certificates signed by some private key available to the caller if they are to be so used.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_AC_INVALID_BASE_ACLS
CSSMERR_AC_INVALID_ENCODING
CSSMERR_AC_INVALID_REQUESTOR
CSSMERR_AC_INVALID_REQUEST_DESCRIPTOR
CSSMERR_AC_INVALID_TUPLE_CREDENTIALS
CSSMERR_AC_INVALID_VALIDITY_PERIOD

SEE ALSO

For the CSSM API:
CSSM_TP_CertGroupToTupleGroup( )
CSSM_TP_TupleGroupToCertGroup( )

For the AC SPI:
TP_CertGroupToTupleGroup( )
TP_TupleGroupToCertGroup( )

9.7 Extensibility Functions

The man-page definition for the Authorization Computation extensibility function is presented in this section.

NAME

CSSM_AC_PassThrough for the CSSM API
AC_PassThrough for the AC SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_AC_PassThrough
    (CSSM_AC_HANDLE ACHandle,
    CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST *DBList,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)

SPI:
CSSM_RETURN CSSMACI AC_PassThrough
    (CSSM_AC_HANDLE ACHandle,
    CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST *DBList,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)
```

DESCRIPTION

This function allows applications to call authorization computation module-specific operations that have been exported. Such operations may include queries or services specific to the domain represented by the AC module.

PARAMETERS

ACHandle (input)
The handle that describes the authorization computation module used to perform this function.

TPHandle (input/optional)
The handle that describes the trust policy module that can be used by the authorization computation service to implement this function. If no trust policy module is specified, the AC module uses an assumed TP module, if required.

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 AC module uses an assumed CL module, if required.

CCHandle (input/optional)
The handle that describes the cryptographic context containing a 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 cryptographic context is specified, the AC module uses an assumed cryptographic context and 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, CRLs, and policy objects for use by the AC module. If no DL and DB handle pairs are specified, the AC module can use an assumed DL module and an assumed data store for this operation.

PassThroughId (input)
An identifier assigned by the AC 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 AC module. If the *passthrough* function requires access to a private key located in the CSP referenced by *CSPHandle*, then *InputParams* should contain a passphrase, or a callback or cryptographic context that can be used to obtain the passphrase.

OutputParams (output/optional)
A pointer to a module, implementation-specific structure containing the output data. The service provider will allocate the memory for this structure. The application must free the memory for the structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_AC_INVALID_CL_HANDLE
CSSMERR_AC_INVALID_CONTEXT_HANDLE
CSSMERR_AC_INVALID_DBLIST_POINTER
CSSMERR_AC_INVALID_DB_LIST
CSSMERR_AC_INVALID_DB_HANDLE
CSSMERR_AC_INVALID_DL_HANDLE
CSSMERR_AC_INVALID_PASSTHROUGH_ID
CSSMERR_AC_INVALID_TP_HANDLE

Technical Standard

Part 6:

Certificate Library (CL) Services

The Open Group

Chapter 10

Certificate Library Services

10.1 Overview

10.1.1 Certificates and CRLs

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:

- Verifying the signatures on certificates and CRLs
- Extracting field values from certificates and CRLs
- Searching CRLs for specified certificates

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.

10.1.2 Application and Certificate Library Interaction

An application determines the availability and basic capabilities of a Certificate Library by querying the Module Directory Services (MDS) records describing the CL module.

When a new CL is installed on a system, the certificate types and certificate fields that it supports are registered with MDS. 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.

10.1.3 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**

  — 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. Field values can be retrieve in their encoded representation or in a type-specific parsed format. Certificates and CRLs can be cached by the certificate library module for efficient processing of repeated access requests for field values.

  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.

- **Certificate revocation list operations**
  Operations on certificate revocation lists comprise 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. The contents of the CRL can be queried for all of its revocation records, specific certificates, or individual CRL fields.

- **Extensibility functions**
  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.

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.

The certificate library developer may choose to implement some or all of these CL interface 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.

10.1.4 Add-In Module

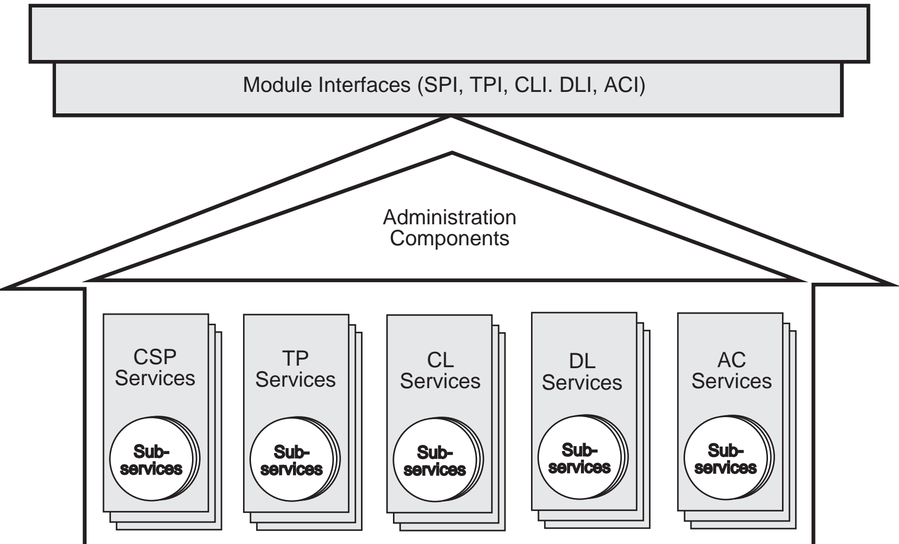


Figure 10-1 CDSA Add-In Module Structure

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, in this case the Certificate Library (CL) services. Each security service contains one or more implementation instances, called sub-services. 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.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions that 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 **Part 14** of this Technical Standard.

10.1.5 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 10-2. 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

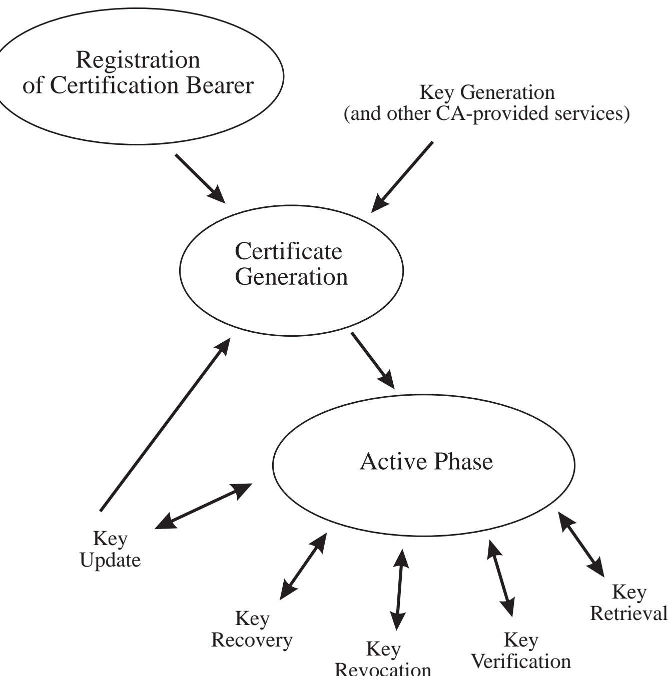


Figure 10-2 Certificate Life Cycle States and Actions

10.2 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.

10.2.1 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 CSSM_MODULE_HANDLE CSSM_CL_HANDLE
```

10.2.2 CSSM_CL_TEMPLATE_TYPE

This type defines an initial set of certificate template types. A certificate template is a buffer containing selected, encoded field values that are currently or will become values in a signed certificate. The template type defines the following aspects of the set of certificate fields:

- The mandatory fields and the optional fields
- Any ordering constraints on the selected fields
- The encoding applied to the selected fields

Each Certificate Library Module should support one default template type.

```
typedef uint32 CSSM_CL_TEMPLATE_TYPE

#define CSSM_CL_TEMPLATE_INTERMEDIATE_CERT   1
    /* for X509 certificates, a fully-formed
       encoded certificate with empty signature field */
#define CSSM_CL_TEMPLATE_PKIX_CERTTEMPLATE   2
    /* as defined in RFC2511, section 5 CertTemplate */
```

10.2.3 CSSM_CERT_BUNDLE_TYPE

This enumerated type lists the industry-defined formats for aggregating certificates and possibly CRLs and other security objects. This class of aggregates is called a certificate bundle. Typically bundles include cryptographic hashes and/or digital signatures of some or all objects in the bundle. Bundles are often used as the representation for exchanging sets of certificates among computing systems. A certificate bundle differs from a CSSM_CERTGROUP in that a bundle is a single encoded object; a CSSM certificate group views each certificate as an independent, encoded object. The CSSM_CERTGROUP data structure is an array of references to individual certificates. Certificates contained in a bundle are located by repeatedly searching the single bundle object.

```
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_SPKI_SEQUENCE = 0x06,
    CSSM_CERT_BUNDLE_PGP_KEYRING = 0x07,
    CSSM_CERT_BUNDLE_LAST = 0x7FFF
} CSSM_CERT_BUNDLE_TYPE;

/* Applications wishing to define their own custom certificate
 * bundle type should define and publicly document a uint32
 * value greater than CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE */

#define CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE 0x8000
```

10.2.4 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_SEXPR = 0x04,
    CSSM_CERT_BUNDLE_ENCODING_PGP = 0x05,
} CSSM_CERT_BUNDLE_ENCODING;
```

10.2.5 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.

10.2.6 CSSM_CERT_BUNDLE

This structure defines a certificate bundle, which consists of a descriptive header and a pointer to the opaque bundle. The bundle is an opaque aggregation of certificates and possibly other security-related objects.

```
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
An opaque aggregation of certificates and possibly other security-related objects.

10.2.7 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
```

10.2.8 CSSM_CRL_TYPE

This structure represents the type of 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_SPKI = 0x03,
    CSSM_CRL_TYPE_MULTIPLE = 0x7FFE,
} CSSM_CRL_TYPE, *CSSM_CRL_TYPE_PTR;
```

10.2.9 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 = 0x04,
    CSSM_CRL_ENCODING_SEXPR = 0x05,
    CSSM_CRL_ENCODING_MULTIPLE = 0x7FFE,
} CSSM_CRL_ENCODING, *CSSM_CRL_ENCODING_PTR;
```

10.2.10 CSSM_ENCODED_CRL

This structure contains a pointer to a certificate revocation list (CRL) in its encoded representation. The CRL is stored as a single contiguous byte array referenced by *CrlBlob*. The length of the byte array is contained in the *Length* subfield of the *CrlBlob*. The type and encoding of the CRL format are also contained in the structure.

```
typedef struct cssm_encoded_crl {
    CSSM_CRL_TYPE CrlType;          /* type of CRL */
    CSSM_CRL_ENCODING CrlEncoding;  /* encoding for this packed CRL */
    CSSM_DATA CrlBlob ;             /* packed CRL */
} CSSM_ENCODED_CRL, *CSSM_ENCODED_CRL_PTR ;
```

Definition

CrlType
Indicates the type of the certificate revocation list referenced by *CrlBlob*.

CrlEncoding
Indicates the encoding of the certificate revocation list referenced by *CrlBlob*.

CrlBlob
A two field structure containing a reference to a CRL in its opaque data blob format and the length of the byte array that contains the CRL blob.

10.2.11 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.

10.2.12 CSSM_FIELDVALUE_COMPLEX_DATA_TYPE

The value to which the Length component of a CSSM_FIELD, *FieldValue* is set to indicate that the FieldValue Data pointer points to a complex data type.

```
#define CSSM_FIELDVALUE_COMPLEX_DATA_TYPE (0xFFFFFFFF)
```

10.3 Error Codes and Error Values

This section defines Error Values that can be returned by CL operations.

The Error Values that can be returned by CL functions can be either derived from the Common Error Codes defined in Appendix A on page 925, or from a set of Errors that more than one CL function can return, or they are specific to a CL function.

10.3.1 CL Error Values Derived from Common Error Codes

```
#define CSSMERR_CL_INTERNAL_ERROR \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)
#define CSSMERR_CL_MEMORY_ERROR  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)
#define CSSMERR_CL_MDS_ERROR  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)
#define CSSMERR_CL_INVALID_POINTER  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)
#define CSSMERR_CL_INVALID_INPUT_POINTER  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)
#define CSSMERR_CL_INVALID_OUTPUT_POINTER  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)
#define CSSMERR_CL_FUNCTION_NOT_IMPLEMENTED  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)
#define CSSMERR_CL_SELF_CHECK_FAILED  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)
#define CSSMERR_CL_OS_ACCESS_DENIED  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)
#define CSSMERR_CL_FUNCTION_FAILED  \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)
#define CSSMERR_CL_INVALID_CONTEXT_HANDLE \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_CONTEXT_HANDLE)
#define CSSMERR_CL_INVALID_CERTGROUP_POINTER \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_CERTGROUP_POINTER)
#define CSSMERR_CL_INVALID_CERT_POINTER \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_CERT_POINTER)
#define CSSMERR_CL_INVALID_CRL_POINTER \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_CRL_POINTER)
#define CSSMERR_CL_INVALID_FIELD_POINTER \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_FIELD_POINTER)
#define CSSMERR_CL_INVALID_DATA \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_DATA)
#define CSSMERR_CL_CRL_ALREADY_SIGNED \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_CRL_ALREADY_SIGNED)
#define CSSMERR_CL_INVALID_NUMBER_OF_FIELDS \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_NUMBER_OF_FIELDS)
#define CSSMERR_CL_VERIFICATION_FAILURE \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_VERIFICATION_FAILURE)
#define CSSMERR_CL_UNKNOWN_FORMAT \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_UNKNOWN_FORMAT)
#define CSSMERR_CL_UNKNOWN_TAG \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_UNKNOWN_TAG)
#define CSSMERR_CL_INVALID_PASSTHROUGH_ID \
    (CSSM_CL_BASE_ERROR+CSSM_ERRCODE_INVALID_PASSTHROUGH_ID)
```

10.3.2 CL Error Values

These codes can be returned by one or more CL APIs.

```
#define CSSM_CL_BASE_CL_ERROR \
    (CSSM_CL_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT)
```

```
#define CSSMERR_CL_INVALID_BUNDLE_POINTER (CSSM_CL_BASE_CL_ERROR+1)
```

Invalid pointer for certificate bundle

```
#define CSSMERR_CL_INVALID_CACHE_HANDLE (CSSM_CL_BASE_CL_ERROR+2)
```

Invalid certificate or CRL cache handle

```
#define CSSMERR_CL_INVALID_RESULTS_HANDLE (CSSM_CL_BASE_CL_ERROR+3)
```

Invalid handle for results of a certificate or CRL query

```
#define CSSMERR_CL_INVALID_BUNDLE_INFO (CSSM_CL_BASE_CL_ERROR+4)
```

Unknown type or encoding for certificate bundle

```
#define CSSMERR_CL_INVALID_CRL_INDEX (CSSM_CL_BASE_CL_ERROR+5)
```

Invalid index for revocation record in CRL

```
#define CSSMERR_CL_INVALID_SCOPE (CSSM_CL_BASE_CL_ERROR+6)
```

Invalid sign or verify scope (function dependent)

```
#define CSSMERR_CL_NO_FIELD_VALUES (CSSM_CL_BASE_CL_ERROR+7)
```

No field matched the specified certificate or CRL field OID

```
#define CSSMERR_CL_SCOPE_NOT_SUPPORTED (CSSM_CL_BASE_CL_ERROR+8)
```

Field signing or verifying is not supported by this module

10.4 Certificate Operations

This section presents the man-page definitions for the functions 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

CSSM_CL_CertCreateTemplate for the CSSM API
CL_CertCreateTemplate for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertCreateTemplate
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CertFields,
    CSSM_DATA_PTR CertTemplate)

SPI:
CSSM_RETURN CSSMCLI CL_CertCreateTemplate
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CertFields,
    CSSM_DATA_PTR CertTemplate)
```

DESCRIPTION

This function allocates and initializes memory for an encoded certificate template output in *CertTemplate→Data*. The template values are specified by the input OID/value pairs contained in *CertFields*. The initialization process includes encoding all certificate field values according to the certificate type and certificate encoding supported by the certificate library module.

The memory for *CertTemplate→Data* is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

NumberOfFields (input)
The number of certificate field values specified in the *CertFields*.

CertFields (input)
A pointer to an array of OID/value pairs that identify the field values to initialize a new certificate.

CertTemplate (output)
A pointer to a CSSM_DATA structure that will contain the unsigned certificate template as a result of this function.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetAllTemplateFields( )
CSSM_CL_CertSign( )

For the CLI SPI:
CL_CertGetAllTemplateFields( )
CL_CertSign( )

NAME

CSSM_CL_CertGetAllTemplateFields for the CSSM API
CL_CertGetAllTemplateFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetAllTemplateFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *CertTemplate,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *CertFields)

SPI:
CSSM_RETURN CSSMCLI CL_CertGetAllTemplateFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *CertTemplate,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *CertFields)
```

DESCRIPTION

This function extracts and returns a copy of the values stored in the encoded *CertTemplate*. The memory for the *CertFields* output is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory by calling *CSSM_CL_FreeFields*( ) (CSSM API) or *CL_FreeFields*( ) (CL SPI).

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

CertTemplate (input)
A pointer to the CSSM_DATA structure containing the packed, encoded certificate template.

NumberOfFields (output)
The length of the output array of fields.

CertFields (output)
A pointer to an array of CSSM_FIELD structures which contains the OIDs and values of the fields of the input certificate template.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_FreeFields( )
CSSM_CL_CertCreateTemplate( )

For the CLI SPI:
CL_FreeFields( )
CL_CertCreateTemplate( )

NAME

CSSM_CL_CertSign for the CSSM API
CL_CertSign for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertTemplate,
    const CSSM_FIELD *SignScope,
    uint32 ScopeSize,
    CSSM_DATA_PTR SignedCert)

SPI:
CSSM_RETURN CSSMCLI CL_CertSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertTemplate,
    const CSSM_FIELD *SignScope,
    uint32 ScopeSize,
    CSSM_DATA_PTR SignedCert)
```

DESCRIPTION

This function signs a certificate using the private key and signing algorithm specified in the *CCHandle*. The result is a signed, encoded certificate in *SignedCert*. The certificate field values are specified in the input certificate template. The template is constructed using *CSSM_CL_CertCreateTemplate*() (CSSM API) or *CL_CertCreateTemplate*() (CL SPI). The template is in the default format for this CL.

The *CCHandle* must be a signature context created using the function *CSSM_CSP_CreateSignatureContext*(). (CSSM API) or *CSP_CreateSignatureContext*(). (SPI). The context must specify the Cryptographic Services Provider module, the signing algorithm, and the signing key that must be used to perform this operation. The context must also provide the passphrase or a callback function to obtain the passphrase required to access and use the private key.

The fields included in the signing operation are identified by the OIDs in the optional *SignScope* array.

The memory for the *SignedCert→Data* output is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this operation.

CCHandle (input)
A signature context defining the CSP, signing algorithm, and private key that must be used to perform the operation. The passphrase for the private key is also provided.

CertTemplate (input)
A pointer to a CSSM_DATA structure containing a certificate template in the default format supported by this CL. The template contains values that are currently contained in or will be contained in a signed certificate.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID/value pairs of the fields to be signed. A null input signs all the fields provided by *CertTemplate*.

ScopeSize (input)
The number of entries in the *SignScope* list. If the sign scope is not specified, the input value for scope size must be zero.

SignedCert (output)
A pointer to the CSSM_DATA structure containing the signed certificate.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_SCOPE
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_SCOPE_NOT_SUPPORTED

SEE ALSO

For the CSSM API:
CSSM_CL_CertVerify()
CSSM_CL_CertCreateTemplate()

For the CLI SPI:
CL_CertVerify()
CL_CertCreateTemplate()

NAME

CSSM_CL_CertVerify for the CSSM API
CL_CertVerify for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertToBeVerified,
    const CSSM_DATA *SignerCert,
    const CSSM_FIELD *VerifyScope,
    uint32 ScopeSize)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertToBeVerified,
    const CSSM_DATA *SignerCert,
    const CSSM_FIELD *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.

The caller can specify a cryptographic service provider and verification algorithm that the CL can use to perform the verification. The handle for the CSP is contained in the cryptographic context identified by *CCHandle*.

The verification process requires that the caller must specify the necessary verification algorithm parameters. These parameter values are specified in one of two locations:

- As a field value in the *SignerCert*
- As a set of algorithm parameters contained in the cryptographic context identified by *CCHandle*

If both of the above arguments are supplied, a consistency check is performed to ensure that they result in the same verification algorithm parameters. If they are not consistent, an error is returned. If only one of the above arguments is supplied, that argument is used to generate the verification algorithm parameters. If no algorithm parameters are found, the certificate can not be verified and the operation fails.

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 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/optional)
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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_SCOPE
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_SCOPE_NOT_SUPPORTED
CSSMERR_CL_VERIFICATION_FAILURE

SEE ALSO

For the CSSM API:
CSSM_CL_CertSign( )

For the CLI SPI:
CL_CertSign( )

NAME

CSSM_CL_CertVerifyWithKey for the CSSM API
CL_CertVerifyWithKey for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertVerifyWithKey
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertToBeVerified)

SPI:
CSSM_RETURN CSSMCLI CL_CertVerifyWithKey
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CertToBeVerified)
```

DESCRIPTION

This function verifies that the *CertToBeVerified* was signed using a specific private key and that the certificate has not been altered since it was signed using that private key. The public key corresponding to the private signing key is used in the verification process.

The *CCHandle*, must be a signature verification context created using the function *CSSM_CSP_CreateSignatureContext*(). (CSSM API) or *CSP_CreateSignatureContext*(). (SPI). The context must specify the Cryptographic Services Provider module, the verification algorithm, and the public verification key that must be used to perform this operation.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library service module used to perform this function.

CCHandle (input)
A signature verification context defining the CSP, verification algorithm, and public key that must be used to perform the operation.

CertToBeVerified (input)
A signed certificate whose signature is to be verified.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_VERIFICATION_FAILURE

SEE ALSO

For the CSSM API:
CSSM_CL_CertVerify()
CSSM_CL_CrlVerify()

For the CLI SPI:
CL_CertVerify( )
CL_CrlVerify( )

NAME

CSSM_CL_CertGetFirstFieldValue for the CSSM API
CL_CertGetFirstFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_OID *CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *Value)

SPI:
CSSM_RETURN CSSMCLI CL_CertGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_OID *CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *Value)
```

DESCRIPTION

This function returns the value of the certificate field designated by the CSSM_OID *CertField*. The OID also identifies the data format for the field value returned to the caller. If multiple OIDs name the same certificate field, then each such OID defines a distinct data format for the returned field value. The function *CSSM_CL_CertDescribeFormat*( ) (CSSM API) or *CL_CertDescribeFormat*( ) (CL SPI). provides a list of all CSSM_OID values supported by a certificate library module for naming fields of a certificate.

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.

The set of matching fields is determined by this function. The number of matching fields and the field values do not change between this function and subsequent calls to *CSSM_CL_CertGetNextFieldValue*( ) (CSSM API) or *CL_CertGetNextFieldValue*( ) (CL SPI).

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 total number of fields that match the *CertField* OID. This count includes the first match, which was returned by this function.

Value (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*Value)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **Value* and *(*Value)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetNextFieldValue( )
CSSM_CL_CertAbortQuery( )
CSSM_CL_CertGetAllField( )
CSSM_CL_FreeFieldValue( )
CSSM_CL_CertDescribeFormat( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CertGetNextFieldValue( )
CL_CertAbortQuery( )
CL_CertGetAllField( )
CL_FreeFieldValue( )
CL_CertDescribeFormat( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CertGetNextFieldValue for the CSSM API
CL_CertGetNextFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *Value)

SPI:
CSSM_RETURN CSSMCLI CL_CertGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *Value)
```

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*( ) (CSSM API) or *CL_CertGetFirstFieldValue*( ) (CL SPI). returns a results handle identifying the size and values contained in the result set. The *CSSM_CL_CertGetNextFieldValue*( ) (CSSM API) or *CL_CertGetNextFieldValue*( ) (CL SPI) function can be called repeatedly to obtain these values, one at a time. The result set does not change in size or value between calls to this function.

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.

Value (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*Value)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **Value* and *(*Value)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstFieldValue( )
CSSM_CL_CertAbortQuery( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CertGetFirstFieldValue()
CL_CertAbortQuery()
CL_FreeFieldValue()

NAME

CSSM_CL_CertAbortQuery for the CSSM API
CL_CertAbortQuery for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates a results handle used to access multiple certificate fields identified by a single OID. The *ResultsHandle* was created and returned by *CSSM_CL_CertGetFirstFieldValue*() (CSSM API) or *CL_CertGetFirstFieldValue*() (CL SPI), or by *CSSM_CL_CertGetFirstCachedFieldValue*(). (CSSM API) or *CL_CertGetFirstCachedFieldValue*(). (CL SPI). The CL releases all intermediate state information associated with the repeating-value query. Once this function has been invoked, the results handle is invalid.

Applications must invoke this function to terminate the *ResultsHandle*. Using *CSSM_CL_CertGetNextFieldValue*() (CSSM API) or *CL_CertGetNextFieldValue*() (CL SPI), or *CSSM_CL_CertGetNextCachedFieldValue*() (CSSM API) or *CL_CertGetNextCachedFieldValue*() (CL SPI), to access all of the attributes named by a single OID does not terminate the *ResultsHandle*. This function can be invoked to terminate the results handle without accessing all of the values identified by the single OID.

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*() (CSSM API) or *CL_GetFieldValue*() (CLI SPI) request.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstFieldValue()
CSSM_CL_CertGetNextFieldValue()
CSSM_CL_CertGetFirstCachedFieldValue()
CSSM_CL_CertGetNextCachedFieldValue()

For the CLI SPI:
CL_CertGetFirstFieldValue()

CL_CertGetNextFieldValue( )
CL_CertGetFirstCachedFieldValue( )
CL_CertGetNextCachedFieldValue( )

NAME

CSSM_CL_CertGetKeyInfo for the CSSM API
CL_CertGetKeyInfo for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetKeyInfo
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_KEY_PTR *Key)

SPI:
CSSM_RETURN CSSMCLI CL_CertGetKeyInfo
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_KEY_PTR *Key)
```

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.

Key (output)
A pointer to the CSSM_KEY structure containing the public key and possibly other key information. The CSSM_KEY structure and its sub-structures are allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstFieldValue( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CertGetFirstFieldValue( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CertGetAllFields for the CSSM API
CL_CertGetAllFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *FieldList)

SPI:
CSSM_RETURN CSSMCLI CL_CertGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *FieldList)
```

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.

FieldList (output)
A pointer to an array of CSSM_FIELD structures that contain the values of all of the fields of the input certificate. The field list is allocated by the service provider and must be de-allocated by the application by calling *CSSM_CL_FreeFields*( ) (CSSM API) or *CL_FreeFields*( ) (CL SPI).

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstFieldValue( )
CSSM_CL_CertDescribeFormat( )
CSSM_CL_FreeFields( )

For the CLI SPI:
CL_CertGetFirstFieldValue( )
CL_CertDescribeFormat( )
CL_FreeFields( )

NAME

CSSM_CL_FreeFields for the CSSM API
CL_FreeFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_FreeFields
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    CSSM_FIELD_PTR *FieldArray)

SPI:
CSSM_RETURN CSSMCLI CL_FreeFields
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    CSSM_FIELD_PTR *FieldArray)
```

DEFINITIONS

This function frees the fields in the *FieldArray* by freeing the Data pointers for both the *FieldOid* and *FieldValue* fields. It also frees the top level *FieldArray* pointer.

This function should only be used to free CSSM_FIELD_PTR values returned from calls
CSSM_TP_CertGetAllTemplateFields( )
CSSM_CL_CertGetAllTemplateFields( )
CSSM_CL_CertGetAllFields( )
CSSM_CL_CrlGetAllFields( )
CSSM_CL_CrlGetAllCachedRecordFields( )
or their SPI equivalent calls.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (input)
The length of the array of fields in *FieldArray*

FieldArray (input)
A pointer to an array of CSSM_FIELD structures that need to be deallocated.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

NAME

CSSM_CL_FreeFieldValue for the CSSM API
CL_FreeFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_FreeFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_OID *CertOrCrlOid,
    CSSM_DATA_PTR Value)

SPI:
CSSM_RETURN CSSMCLI CL_FreeFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_OID *CertOrCrlOid,
    CSSM_DATA_PTR Value)
```

DESCRIPTION

This function frees the data specified by *Value* and *Value→Data*. *CertOrCrlOid* indicates the type of the data in *Value*.

This function should only be used to free CSSM_DATA values returned from calls
CSSM_CL_CertGetFirstFieldValue( )
CSSM_CL_CertGetNextFieldValue( )
CSSM_CL_CertGetFirstCachedFieldValue( )
CSSM_CL_CertGetNextCachedFieldValue( )
CSSM_CL_CrlGetFirstFieldValue( )
CSSM_CL_CrlGetNextFieldValue( )
CSSM_CL_CrlGetFirstCachedFieldValue( )
CSSM_CL_CrlGetNextCachedFieldValue( )
or their CLI SPI equivalents.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CertOrCrlOid (input)
A pointer to the CSSM_OID structure describing the type of the *Value* to be freed.

Value (input)
A pointer to the CSSM_DATA structure containing the Data to be freed.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_UNKNOWN_TAG

NAME

CSSM_CL_CertCache for the CSSM API
CL_CertCache for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertCache
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_HANDLE_PTR CertHandle)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertCache
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_HANDLE_PTR CertHandle)
```

DESCRIPTION

This function caches a copy of a certificate for subsequent accesses using the functions *CSSM_CL_CertGetFirstCachedFieldValue*() (CSSM API) or *CL_CertGetFirstCachedFieldValue*() (CL SPI), and *CSSM_CL_CertGetNextCachedFieldValue*() (CSSM API) or *CL_CertGetNextCachedFieldValue*() (CL SPI).

The input certificate must be in an encoded representation. The Certificate Library module can cache the certificate in any appropriate internal representation. Parsed or incrementally parsed representations are common. The selected representation is opaque to the caller.

The application must call *CSSM_CL_CertAbortCache*() (CSSM API) or *CL_CertAbortCache*() (CL SPI), to remove the cached copy when additional *get* operations will not be performed on the cached certificate.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the encoded certificate.

CertHandle (output)
A pointer to the CSSM_HANDLE that should be used in all future references to the cached certificate.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstCachedFieldValue()
CSSM_CL_CertGetNextCachedFieldValue()
CSSM_CL_CertAbortQuery()

CSSM_CL_CertAbortCache( )

For the CLI SPI:
CL_CertGetFirstCachedFieldValue( )
CL_CertGetNextCachedFieldValue( )
CL_CertAbortQuery( )
CL_CertAbortCache( )

NAME

CSSM_CL_CertGetFirstCachedFieldValue for the CSSM API
CL_CertGetFirstCachedFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetFirstCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CertHandle,
    const CSSM_OID *CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *FieldValue)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetFirstCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CertHandle,
    const CSSM_OID *CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *FieldValue)
```

DESCRIPTION

This function returns a single structure containing a set of field values from the cached certificate identified by *CertHandle*. The selected fields are designated by the CSSM_OID CertField and returned in the output parameter *FieldValue*. The OID also identifies the data format of the values returned to the caller. If multiple OIDs designate the same certificate field, then each such OID defines a distinct data format for the returned values. The function *CSSM_CL_CertDescribeFormat*( ) (CSSM API) or *CL_CertDescribeFormat*( ) (CL SPI) provides a list of all CSSM_OID values supported by a certificate library module for naming fields of a certificate.

The *CertField* OID can identify a single occurrence of a set of certificate fields, or multiple occurrences of a set of certificate fields. If the *CertField* OID matches more than one occurrence, this function outputs the total number of matches and a *ResultsHandle* to be used as input to *CSSM_CertGetNextCachedFieldValue*( ) (CSM API) or *CertGetNextCachedFieldValue*( ) (CL SPI) to retrieve the remaining matches. The first match is returned as the return value of this function.

This function determines the complete set of matches. The number of matches and the selected field values do not change between this function and subsequent calls to *CSSM_CL_CertGetNextCachedFieldValue*( ). (CSSM API) or *CL_CertGetNextCachedFieldValue*( ). (CL SPI).

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CertHandle (input)
A handle identifying a certificate that the application has temporarily cached with the Certificate Library module. The referenced certificate is searched for the field value named by *CertField*.

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 total number of fields that match the *CertField* OID. This count includes the first match, which was returned by this function.

FieldValue (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*FieldValue)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate *FieldValue* and *(*FieldValue)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CACHE_HANDLE
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetNextCachedFieldValue( )
CSSM_CL_CertAbortCache( )
CSSM_CL_CertAbortQuery( )
CSSM_CL_CertGetAllFields( )
CSSM_CL_CertDescribeFormat( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CertGetNextCachedFieldValue( )
CL_CertAbortCache( )
CL_CertAbortQuery( )
CL_CertGetAllFields( )
CL_CertDescribeFormat( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CertGetNextCachedFieldValue for the CSSM API
CL_CertGetNextCachedFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetNextCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *FieldValue)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertGetNextCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *FieldValue)
```

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_CertGetFirstCachedFieldValue*( ) (CSSM API) or *CL_CertGetFirstCachedFieldValue*( ) (CL SPI) returns a *ResultsHandle* identifying the size and values contained in the result set. The *CSSM_CL_CertGetNextCachedFieldValue*( ) (CSSMAPI) or *CL_CertGetNextCachedFieldValue*( ) (CL SPI) function can be called repeatedly to obtain these values, one at a time. The result set does not change in size or value between calls to this function.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

FieldValue (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*FieldValue)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **FieldValue* and *(*FieldValue)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetFirstCachedFieldValue( )
CSSM_CL_CertAbortCache( )
CSSM_CL_CertAbortQuery( )
CSSM_CL_CertGetAllFields( )

CSSM_CL_CertDescribeFormat( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CertGetFirstCachedFieldValue( )
CL_CertAbortCache( )
CL_CertAbortQuery( )
CL_CertGetAllFields( )
CL_CertDescribeFormat( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CertAbortCache for the CSSM API
CL_CertAbortCache for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertAbortCache
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CertHandle)
SPI:

CSSM_RETURN CSSMAPI CSSM_CL_CertAbortCache
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CertHandle)
```

DECRIPTION

This function terminates a certificate cache handle created and returned by the function *CSSM_CL_CertCache*( ) (CSSM API) or *CL_CertCache*( ) (CL SPI). The Certificate Library module releases all cache space and state information associated with the cached certificate.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

CertHandle (input)
The handle that identifies the cached certificate.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CACHE_HANDLE

SEE ALSO

For the CSSM API:
CSSM_CL_CertCache( )

For the CLI SPI:
CL_CertCache( )

NAME

CSSM_CL_CertGroupToSignedBundle for the CSSM API
CL_CertGroupToSignedBundle for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGroupToSignedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERTGROUP *CertGroupToBundle,
    const CSSM_CERT_BUNDLE_HEADER *BundleInfo,
    CSSM_DATA_PTR SignedBundle)

SPI:
CSSM_RETURN CSSMCLI CL_CertGroupToSignedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERTGROUP *CertGroupToBundle,
    const CSSM_CERT_BUNDLE_HEADER *BundleInfo,
    CSSM_DATA_PTR SignedBundle)
```

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. 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/optional)
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.

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.

SignedBundle (output)
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 type defined by the library module if the *BundleInfo* was not specified by the caller. The *SignedBundle*→*Data* is allocated by the service provider and must be deallocated

by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CERTGROUP_POINTER
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_BUNDLE_POINTER
CSSMERR_CL_INVALID_BUNDLE_INFO

SEE ALSO

For the CSSM API:
CSSM_CL_CertGroupFromVerifiedBundle()

For the CLI SPI:
CL_CertGroupFromVerifiedBundle()

NAME

CSSM_CL_CertGroupFromVerifiedBundle for the CSSM API
CL_CertGroupFromVerifiedBundle for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertGroupFromVerifiedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERT_BUNDLE *CertBundle,
    const CSSM_DATA *SignerCert,
    CSSM_CERTGROUP_PTR *CertGroup)

SPI:
CSSM_RETURN CSSMCLI CL_CertGroupFromVerifiedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERT_BUNDLE *CertBundle,
    const CSSM_DATA *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/optional)
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 certificate group and CSSM_CERTGROUP substructures are allocated by the serivce provider and must be deallocated by the application. The group contains all of the certificates contained in the certificate bundle.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_BUNDLE_POINTER
CSSMERR_CL_INVALID_BUNDLE_INFO
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_INVALID_CERTGROUP_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_CertGroupToSignedBundle( )

For the CLI SPI:
CL_CertGroupToSignedBundle( )

NAME

CSSM_CL_CertDescribeFormat for the CSSM API
CL_CertDescribeFormat for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CertDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfOids,
    CSSM_OID_PTR *OidList)

SPI:
CSSM_RETURN CSSMAPI CSSM_CL_CertDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfOids,
    CSSM_OID_PTR *OidList)
```

DESCRIPTION

This function returns a list of the CSSM_OID values this certificate library module uses to name and reference fields of a certificate. Multiple CSSM_OID values can correspond to a single field. These OIDs can be provided as input to *CSSM_CL_CertGetFirstFieldValue*( ) (CSSM API) or *CL_CertGetFirstFieldValue*( ) (CL SPI) to retrieve field values from the certificate. The OID also implies the data format of the returned value. When multiple OIDs name the same field of a certificate, those OIDs have a different return data formats associated with them.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfOids (output)
The length of the returned array of OIDs.

OidList (output)
A pointer to the array of CSSM_OIDs which represent the supported certificate format. The OID list is allocated by the service provider and must be de-allocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_CL_CertGetAllFields( )
CSSM_CL_CertGetFirstFieldValue( )
CSSM_CL_CertGetFirstCachedFieldValue( )

For the CLI SPI:
CL_CertGetAllFields( )
CL_CertGetFirstFieldValue( )
CL_CertGetFirstCachedFieldValue( )

10.5 Certificate Revocation List Operations

This section presents the man-page definitions for the functions prototypes 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

CSSM_CL_CrlCreateTemplate for the CSSM API
CL_CrlCreateTemplate for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlCreateTemplate
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlTemplate,
    CSSM_DATA_PTR NewCrl)

SPI:
CSSM_RETURN CSSMCLI CL_CrlCreateTemplate
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlTemplate,
    CSSM_DATA_PTR NewCrl)
```

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*( ) call (CSSM API) or *CL_CrlSetFields*( ) call (CL SPI). 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.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

CrlTemplate (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL.

NewCrl (output)
A pointer to the CSSM_DATA structure containing the new CRL. The *NewCrl*→ *Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_INVALID_CRL_POINTER

SEE ALSO

For the CSSM API:
CSSM_CL_CrlSetFields( )
CSSM_CL_CrlAddCert( )
CSSM_CL_CrlSign( )
CSSM_CL_CertGetFirstFieldValue( )

For the CLI SPI:
CL_CrlSetFields( )
CL_CrlAddCert( )
CL_CrlSign( )
CL_CertGetFirstFieldValue( )

NAME

CSSM_CL_CrlSetFields for the CSSM API
CL_CrlSetFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlSetFields
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlTemplate,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR ModifiedCrl)
SPI:
CSSM_RETURN CSSMCLI CL_CrlSetFields
    (CSSM_CL_HANDLE CLHandle,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlTemplate,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR ModifiedCrl)
```

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.

This should be used to update any of the CRL field values. If a specified field was initialized by *CSSM_CL_CrlCreateTemplate*() (CSSM API) or *CL_CrlCreateTemplate*() (CL SPI), the field value is set to the new specified value. If a specified field was not initialized by the *CSSM_CL_CrlCreateTemplate*() (CSSM API) or *CL_CrlCreateTemplate*() (CL SPI), the field is set to the new specified value. *OldCrl* must be unsigned. Once a CRL has been signed using *CSSM_CL_CrlSign*() (CSSM API) or *CL_CrlSign*() (CL SPI), the signed CRL's field values can not be modified. Modification would invalidate the cryptographic signature of the CRL.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (input)
: The number of OID/value pairs specified in the CrlTemplate input parameter.

CrlTemplate (input)
: Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

OldCrl (input)
: The CRL to be updated with the new attribute values. The CRL must be unsigned and available for update.

ModifiedCrl (output)
: A pointer to the modified, unsigned CRL. The *ModifiedCrl→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_CRL_ALREADY_SIGNED

SEE ALSO

For the CSSM API:
CSSM_CL_CrlCreateTemplate( )
CSSM_CL_CrlAddCert( )
CSSM_CL_CrlSign( )
CSSM_CL_CertGetFirstFieldValue( )

For the CLI SPI:
CL_CrlCreateTemplate( )
CL_CrlAddCert( )
CL_CrlSign( )
CL_CertGetFirstFieldValue( )

NAME

CSSM_CL_CrlAddCert for the CSSM API
CL_CrlAddCert for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlAddCert
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Cert,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlEntryFields,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR NewCrl)

SPI:
CSSM_RETURN CSSMCLI CL_CrlAddCert
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *Cert,
    uint32 NumberOfFields,
    const CSSM_FIELD *CrlEntryFields,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR NewCrl)
```

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 new CRL entry is signed using the private key and signing algorithm specified in the *CCHandle*.

The *CCHandle* must be a context created using the function *CSSM_CSP_CreateSignatureContext*() (CSSM API) or *CSP_CreateSignatureContext*() (CL SPI). The context must specify the Cryptographic Services Provider module, the signing algorithm, and the signing key that must be used to perform this operation. The context must also provide the passphrase or a callback function to obtain the passphrase required to access and use the private key.

The operation is valid only if the CRL has not been closed by the process of signing the CRL, by calling *CSSM_CL_CrlSign*() (CSSM API) or *CL_CrlSign*() (CL SPI). 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.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlEntryFields input parameter.

CrlEntryFields (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL entry.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to which the newly-revoked certificate will be added.

NewCrl (output)
A pointer to the CSSM_DATA structure containing the updated CRL. The *NewCrl→Data* is allocated by the service provider and must be de-allocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_CRL_ALREADY_SIGNED

SEE ALSO

For the CSSM API:
CSSM_CL_CrlRemoveCert( )

For the CLI SPI:
CL_CrlRemoveCert( )

NAME

CSSM_CL_CrlRemoveCert for the CSSM API
CL_CrlRemoveCert for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlRemoveCert
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR NewCrl)

SPI:
CSSM_RETURN CSSMCLI CL_CrlRemoveCert
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_DATA *OldCrl,
    CSSM_DATA_PTR NewCrl)
```

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 *CSSM_CL_CrlSign*() (CSSM API) or *CL_CrlSign*() (CL SPI). 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.

NewCrl (output)
A pointer to the CSSM_DATA structure containing the updated CRL. The *NewCrl*→*Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_CRL_ALREADY_SIGNED

SEE ALSO

For the CSSM API:
CSSM_CL_CrlAddCert()

For the CLI SPI:
CL_CrlAddCert()

NAME

CSSM_CL_CrlSign for the CSSM API
CL_CrlSign for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *UnsignedCrl,
    const CSSM_FIELD *SignScope,
    uint32 ScopeSize,
    CSSM_DATA_PTR SignedCrl)

SPI:
CSSM_RETURN CSSMCLI CL_CrlSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *UnsignedCrl,
    const CSSM_FIELD *SignScope,
    uint32 ScopeSize,
    CSSM_DATA_PTR SignedCrl)
```

DESCRIPTION

This function signs a CRL using the private key and signing algorithm specified in the *CCHandle*. The result is a signed, encoded certificate revocation list in *SignedCrl*. The unsigned CRL is specified in the input *UnsignedCrl*. The *UnsignedCrl* is constructed using the *CSSM_CL_CrlCreateTemplate*( ), *CSSM_CL_CrlSetFields*( ), *CSSM_CL_CrlAddCert*( ), and *CSSM_CL_CrlRemoveCert*( ) functions (for the CSSM API) or their CL SPI equivalents.

The *CCHandle* must be a context created using the function *CSSM_CSP_CreateSignatureContext*( ) (CSSM API) or *CSP_CreateSignatureContext*( ) (SPI). The context must specify the Cryptographic Services Provider module, the signing algorithm, and the signing key that must be used to perform this operation. The context must also provide the passphrase or a callback function to obtain the passphrase required to access and use the private key.

The fields included in the signing operation are identified by the OIDs in the optional *SignScope* array.

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*( ) (or their CL SPI equivalent operations. A signed CRL can be verified, applied to a data store, and searched for values.

The memory for the *SignedCrl→Data* output is allocated by the service provider using the calling application's memory management routines. The application must deallocate the memory.

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.

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.

SignedCrl (output)
A pointer to the CSSM_DATA structure containing the signed CRL. The *SignedCrl→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_SCOPE
CSSMERR_CL_SCOPE_NOT_SUPPORTED
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_CRL_ALREADY_SIGNED

SEE ALSO

For the CSSM API:
CSSM_CL_CrlVerify( )
CSSM_CL_CrlVerifyWithKey( )

For the CLI SPI:
CL_CrlVerify( )
CL_CrlVerifyWithKey( )

NAME

CSSM_CL_CrlVerify for the CSSM API
CL_CrlVerify for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CrlToBeVerified,
    const CSSM_DATA *SignerCert,
    const CSSM_FIELD *VerifyScope,
    uint32 ScopeSize)

SPI:
CSSM_RETURN CSSMCLI CL_CrlVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CrlToBeVerified,
    const CSSM_DATA *SignerCert,
    const CSSM_FIELD *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/optional)
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/optional)
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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_FIELD_POINTER
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_INVALID_SCOPE
CSSMERR_CL_INVALID_NUMBER_OF_FIELDS
CSSMERR_CL_SCOPE_NOT_SUPPORTED
CSSMERR_CL_VERIFICATION_FAILURE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlSign( )

For the CLI SPI:
CL_CrlSign( )

NAME

CSSM_CL_CrlVerifyWithKey for the CSSM API
CL_CrlVerifyWithKey for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlVerifyWithKey
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CrlToBeVerified)

SPI:
CSSM_RETURN CSSMCLI CL_CrlVerifyWithKey
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA *CrlToBeVerified)
```

DESCRIPTION

This function verifies that the *CrlToBeVerified* was signed using a specific private key and that the certificate revocation list has not been altered since it was signed using that private key. The public key corresponding to the private signing key is used in the verification process.

The cryptographic context indicated by *CCHandle* must be a signature verification context created using the function *CSSM_CSP_CreateSignatureContext*( ) (CSSM API) or *CSP_CreateSignatureContext*( ) (CL SPI). The context must specify the Cryptographic Services Provider module, the verification algorithm, and the public verification key that must be used to perform this operation.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library service module used to perform this function.

CCHandle (input)
A signature verification context defining the CSP, verification algorithm, and public key that must be used to perform the operation.

CrlToBeVerified (input)
A signed certificate revocation list whose signature is to be verified.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_VERIFICATION_FAILURE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlVerify( )

For the CLI SPI:
CL_CrlVerify( )

NAME

CSSM_CL_IsCertInCrl for the CSSM API
CL_IsCertInCrl for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_IsCertInCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_DATA *Crl,
    CSSM_BOOL *CertFound)

SPI:
CSSM_RETURN CSSMCLI CL_IsCertInCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    const CSSM_DATA *Crl,
    CSSM_BOOL *CertFound)
```

DESCRIPTION

This function searches the CRL for a record corresponding to the certificate. The result of the search is returned in *CertFound*. The CRL and the records within the CRL must be digitally signed. This function does not verify either signature. The caller should use *CSSM_TP_CrlVerify*() or *CSSM_CL_CrlVerify*() (or their SPI equivalents) before invoking this function. Once the CRL has been verified, the caller can invoke this function repeatedly without repeating the verification process.

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.

CertFound (output)
A pointer to a CSSM_BOOL indicating success or failure in finding the specified certificate in the CRL. CSSM_TRUE signifies that the certificate was found in the CRL. CSSM_FALSE indicates that the certificate was not found in the CRL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

NAME

CSSM_CL_CrlGetFirstFieldValue for the CSSM API
CL_CrlGetFirstFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    const CSSM_OID *CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *Value)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    const CSSM_OID *CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *Value)
```

DESCRIPTION

This function returns the value of the CRL field designated by the CSSM_OID *CrlField*. The OID also identifies the data format for the field value returned to the caller. If multiple OIDs name the same CRL field, then each such OID defines a distinct data format for the returned field value. The function *CSSM_CL_CrlDescribeFormat*( ) (CSSM API) or *CL_CrlDescribeFormat*( ) (CL SPI) provides a list of all CSSM_OID values supported by a certificate library module for naming fields of a CRL.

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.

The set of matching fields is determined by this function. The number of matching fields and the field values do not change between this function and subsequent calls to *CSSM_CL_CrlGetNextFieldValue*( ) (CSSM API) or *CL_CrlGetNextFieldValue*( ) (CL SPI).

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 total number of fields that match the *CrlField* OID. This count includes the first match, which was returned by this function.

Value (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*Value)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **Value* and *(*Value)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetNextFieldValue( )
CSSM_CL_CrlAbortQuery( )
CSSM_CL_CrlGetAllFields( )

For the CLI SPI:
CL_CrlGetNextFieldValue( )
CL_CrlAbortQuery( )
CL_CrlGetAllFields( )

NAME

CSSM_CL_CrlGetNextFieldValue for the CSSM API
CL_CrlGetNextFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *Value)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *Value)
```

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*( ) (CSSM API) or *CL_CrlGetFirstFieldValue*( ) (CL SPI) initiates the process and returns a results handle identifying the size and values contained in the result set. The *CSSM_CL_CrlGetNextFieldValue*( ) (CSSM API) or *CL_CrlGetNextFieldValue*( ) (CL SPI) function can be called repeatedly to obtain these values, one at a time. The result set does not change in size or value between calls to this function.

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.

Value (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*Value)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **Value* and *(*Value)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstFieldValue( )
CSSM_CL_CrlAbortQuery( )

For the CLI SPI:
CL_CrlGetFirstFieldValue( )
CL_CrlAbortQuery( )

NAME

CSSM_CL_CrlAbortQuery for the CSSM API
CL_CrlAbortQuery for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)

SPI:
CSSM_RETURN CSSMCLI CL_CrlAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the query initiated by *CSSM_CL_CrlGetFirstFieldValue*( ) or *CSSM_CL_CrlGetFirstCachedFieldValue*( ) (or their CL SPI equivalents) and allows the CL to release all intermediate state information associated with the repeating-value *get* operation. Once this function has been invoked, the results handle is invalid.

Applications must invoke this function to terminate the *ResultsHandle*. Using *CSSM_CL_CrlGetNextFieldValue*( ) or *CSSM_CL_CrlGetNextCachedFieldValue*( ) (or their CL SPI equivalents) to access all of the attributes named by a single OID does not terminate the *ResultsHandle*. This function can be invoked to terminate the results handle without accessing all of the values identified by the single OID.

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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstFieldValue( )
CSSM_CL_CrlGetNextFieldValue( )
CSSM_CL_CrlGetFirstCachedFieldValue( )
CSSM_CL_CrlGetNextCachedFieldValue( )

For the CL SPI:
CL_CrlGetFirstFieldValue( )
CL_CrlGetNextFieldValue( )
CL_CrlGetFirstCachedFieldValue( )
CL_CrlGetNextCachedFieldValue( )

CL_CrlGetAllFields

NAME

CSSM_CL_CrlGetAllFields for the CSSM API
CL_CrlGetAllFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    uint32 *NumberOfCrlFields,
    CSSM_FIELD_PTR *CrlFields)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    uint32 *NumberOfCrlFields,
    CSSM_FIELD_PTR *CrlFields)
```

DESCRIPTION

This function returns one or more structures each containing a set of field values from the encoded, packed CRL contained in Crl. Each structure is returned in the *FieldValue* entry of the CSSM_FIELD structure *CrlFields*.. The parameter *NumberOfCrlFields* indicates the number of returned structures.

The CRL can be signed or unsigned. This function does not perform any signature verification on the CRL fields or the CRL-records. Each CRL-record can be digitally signed when it is added to the CRL using the function *CSSM_CL_CrlAddCert*( ) (CSSM API) or *CL_CrlAddCert*( ) (CL SPI).

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 encoded, packed, CRL from which field values are to be extracted.

NumberOfCrlFields (output)
The number of entries in the array *CrlFields*.

CrlFields (output)
A pointer to an array of OID-value pairs that describe the contents of the CRL. The extracted CRL fields are returned as the value portion of each OID-value pair. The field list is allocated by the service provider and must be deallocated by the application by calling *CSSM_CL_FreeFields*( ) (CSSM API) or *CL_FreeFields*( ) (CL SPI).

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_FreeFields( )

For the CLI SPI:
CL_FreeFields( )

NAME

CSSM_CL_CrlCache for the CSSM API
CL_CrlCache for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlCache
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    CSSM_HANDLE_PTR CrlHandle)

SPI:
CSSM_RETURN CSSMCLI CL_CrlCache
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Crl,
    CSSM_HANDLE_PTR CrlHandle)
```

DESCRIPTION

This function caches a copy of a *CertificateRevocationList* (CRL) for subsequent accesses using the functions *CSSM_CL_CrlGetFirstFieldValue*() and *CSSM_CL_CrlGetNextFieldValue*() (or their CL SPI equivalents).

The input CRL must be in an encoded representation. The Certificate Library module can cache the CRL in any appropriate internal represent. Parsed or incrementally parsed representations are common. The selected representation is opaque to the caller.

The application must call *CSSM_CL_CrlCacheAbort*() (CSSM API) or *CL_CrlCacheAbort*() (CL SPI) to remove the cached copy when additional *get* operations will not be performed on the cached CRL.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

Crl (input)
A pointer to the CSSM_DATA structure containing the encoded CRL.

CrlHandle (output)
A pointer to the CSSM_HANDLE that should be used in all future references to the cached CRL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CRL_POINTER
CSSMERR_CL_UNKNOWN_FORMAT

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstCachedFieldValue()
CSSM_CL_CrlGetNextCachedFieldValue()
CSSM_CL_IsCertInCachedCrl()
CSSM_CL_CrlAbortQuery()

CSSM_CL_CrlAbortCache( )

For the CLI SPI:
CL_CrlGetFirstCachedFieldValue( )
CL_CrlGetNextCachedFieldValue( )
CL_IsCertInCachedCrl( )
CL_CrlAbortQuery( )
CL_CrlAbortCache( )

NAME

CSSM_CL_IsCertInCachedCrl for the CSSM API
CL_IsCertInCachedCrl for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_IsCertInCachedCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_HANDLE CrlHandle,
    CSSM_BOOL *CertFound,
    CSSM_DATA_PTR CrlRecordIndex)

SPI:
CSSM_RETURN CSSMCLI CL_IsCertInCachedCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA *Cert,
    CSSM_HANDLE CrlHandle,
    CSSM_BOOL *CertFound,
    CSSM_DATA_PTR CrlRecordIndex)
```

DESCRIPTION

This function searches the cached CRL for a record corresponding to the certificate. The result of the search is returned in *CertFound*. The CRL and the records within the CRL must be digitally signed. This function does not verify either signature. The caller should use *CSSM_TP_CrlVerify*() or *CSSM_CL_CrlVerify*() (or their SPI equivalents) before invoking this function. Once the CRL has been verified, the caller can invoke this function repeatedly without repeating the verification process.

If the certificate is found in the CRL, the CL module returns an index descriptor *CrlRecordIndex* for use with other Certificate Library CRL functions. The index provides more direct access to the selected CRL record.

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 an encoded, packed certificate.

CrlHandle (input)
A handle identifying a CRL that the application has temporarily cached with the Certificate Library module. The referenced CRL is searched for a revocation record matching the specified Cert.

CertFound (output)
A pointer to a CSSM_BOOL indicating success or failure in finding the specified certificate in the CRL. CSSM_TRUE signifies that the certificate was found in the CRL. CSSM_FALSE indicates that the certificate was not found in the CRL.

CrlRecordIndex (output)
A pointer to a CSSM_DATA structure containing an index descriptor for direct access to the located CRL record. *CrlRecordIndex→Data* is allocated by the service provider and must be deallocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CERT_POINTER
CSSMERR_CL_UNKNOWN_FORMAT
CSSMERR_CL_INVALID_CACHE_HANDLE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstCachedFieldValue()
CSSM_CL_CrlGetNextCachedFieldValue()
CSSM_CL_CrlGetAllCachedRecordField()
CSSM_CL_CrlCache()
CSSM_CL_CrlAbortCache()

For the CLI SPI:
CL_CrlGetFirstCachedFieldValue()
CL_CrlGetNextCachedFieldValue()
CL_CrlGetAllCachedRecordField()
CL_CrlCache()
CL_CrlAbortCache()

NAME

CSSM_CL_CrlGetFirstCachedFieldValue for the CSSM API
CL_CrlGetFirstCachedFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetFirstCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle,
    const CSSM_DATA *CrlRecordIndex,
    const CSSM_OID *CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *FieldValue)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetFirstCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle,
    const CSSM_DATA *CrlRecordIndex,
    const CSSM_OID *CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields,
    CSSM_DATA_PTR *FieldValue)
```

DESCRIPTION

This function returns a single structure containing a set of field values from the cached CRL identified by *CrlHandle*.*The*selected*fields*are *CrlField* and returned in the output parameter *FieldValue*. The OID also identifies the data format of the values returned to the caller. If multiple OIDs designate the same CRL field, then each such OID defines a distinct data format for the returned values. The function *CSSM_CL_CrlDescribeFormat*( ) (CSSM API) or *CL_CrlDescribeFormat*( ) (CL SPI) provides a list of all CSSM_OID values supported by a CL module for naming fields of a CRL.

The search can be limited to a particular revocation record within the CRL. A single record is identified by the *CrlRecordIndex*, which is returned by the function *CSSM_CL_IsCertInCachedCrl*( ) (CSSM API) or *CL_IsCertInCachedCrl*( ) (CL SPI). If no record index is supplied, the search is initiated from the beginning of the CRL.

The CRL can be signed or unsigned. This function does not perform any signature verification on the CRL fields or the CRL-records. Each CRL-record may be digitally signed when it is added to the CRL using the function *CSSM_CL_CrlAddCert*( ) (CSSM API) or *CL_CrlAddCert*( ) (CL SPI). The caller can examine fields in the CRL and CRL-records at any time using this function.

The *CrlField* OID can identify a single occurrence of a set of CRL fields, or multiple occurrences of a set of CRL fields. If the *CrlField* OID matches more than one occurrence, this function outputs the total number of matches and a *ResultsHandle* to be used as input to *CSSM_CrlGetNextFieldValue*( ) (CSSM API) or *CrlGetNextFieldValue*( ) (CL SPI) to retrieve the remaining matches. The first match is returned as the return value of this function..

This function determines the complete set of matches. The number of matches and the selected field values do not change between this function and subsequent calls to *CSSM_CL_CrlGetNextFieldValue*( ) (CSSM API) or *CL_CrlGetNextFieldValue*( ) (CL SPI).

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlHandle (input)
A handle identifying a CRL that the application has temporarily cached with the Certificate Library module. The referenced CRL is searched for the field values identified by *CrlField*.

CrlRecordIndex (input/optional)
An index value identifying a particular revocation record in a cached CRL. If an index value is supplied, the scan for the field values identified by *CrlField* is limited to the pre-selected revocation record.

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 total number of fields that match the *CrlField* OID. This count includes the first match, which was returned by this function.

FieldValue (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*FieldValue)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **FieldValue* and *(*FieldValue)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CACHE_HANDLE
CSSMERR_CL_INVALID_CRL_INDEX
CSSMERR_CL_UNKNOWN_TAG
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetNextCachedFieldValue( )
CSSM_CL_IsCertInCachedCrl( )
CSSM_CL_CrlAbortQuery( )
CSSM_CL_CrlCache( )
CSSM_CL_CrlAbortCache( )
CSSM_CL_CrlDescribeFormat( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CrlGetNextCachedFieldValue( )
CL_IsCertInCachedCrl( )
CL_CrlAbortQuery( )

CL_CrlCache( )
CL_CrlAbortCache( )
CL_CrlDescribeFormat( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CrlGetNextCachedFieldValue for the CSSM API
CL_CrlGetNextCachedFieldValue for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetNextCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *FieldValue)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetNextCachedFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DATA_PTR *FieldValue)
```

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_CrlGetFirstCachedFieldValue*( ) (CSSM API) or *CL_CrlGetFirstCachedFieldValue*( ) (CL SPI) initiates the process and returns a *ResultsHandle* identifying the size and values contained in the result set. The *CSSM_CL_CrlGetNextCachedFieldValue*( ) (CSSM API) or *CL_CrlGetNextCachedFieldValue*( ) (CL SPI) function can be called repeatedly to obtain these values, one at a time. The result set does not change in size or value between calls to this function.

The results set selected by *CSSM_CL_CrlGetFirstCachedFieldValue*( ) (CSSM API) or *CL_CrlGetFirstCachedFieldValue*( ) (CL SPI) and identified by *ResultsHandle* can reference CRL fields repeated across multiple revocation records or within one revocation record. The scope of the scan was set by an optional *CrlRecordIndex* input to the function *CSSM_CL_CrlGetFirstCachedFieldValue*( ) (CSSM API) or *CL_CrlGetFirstCachedFieldValue*( ) (CL SPI). If the record index was specified, then the results set is to the revocation record identified by the index. If no record index was specified, then the results set can include repeated fields from multiple revocation records in a CRL.

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.

FieldValue (output)
A pointer to the structure containing the value of the requested field. The structure and the field at I "(*FiledValue)→Data" are allocated by the service provider. The *CSSM_CL_FreeFieldValue*( ) (CSSM API) or *CL_FreeFieldValue*( ) (CL SPI) function can be used to deallocate **FieldValue* and *(*FieldValue)→Data*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_RESULTS_HANDLE
CSSMERR_CL_NO_FIELD_VALUES

SEE ALSO

For the CSSM API:
CSSM_CL_CrlGetFirstCachedFieldValue( )
CSSM_CL_CrlAbortQuery( )
CSSM_CL_IsCertInCachedCrl( )
CSSM_CL_CrlCache( )
CSSM_CL_CrlAbortCache( )
CSSM_CL_FreeFieldValue( )

For the CLI SPI:
CL_CrlGetFirstCachedFieldValue( )
CL_CrlAbortQuery( )
CL_IsCertInCachedCrl( )
CL_CrlCache( )
CL_CrlAbortCache( )
CL_FreeFieldValue( )

NAME

CSSM_CL_CrlGetAllCachedRecordFields for the CSSM API
CL_CrlGetAllCachedRecordFields for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlGetAllCachedRecordFields
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle,
    const CSSM_DATA *CrlRecordIndex,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *Fields)

SPI:
CSSM_RETURN CSSMCLI CL_CrlGetAllCachedRecordFields
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle,
    const CSSM_DATA *CrlRecordIndex,
    uint32 *NumberOfFields,
    CSSM_FIELD_PTR *Fields)
```

DESCRIPTION

This function returns all field values from a pre-located, cached CRL record. The scanned CRL record is identified by *CrlRecordIndex*, which is returned by the function *CSSM_CL_IsCertInCachedCrl*( ) (CSSM API) or *CL_IsCertInCachedCrl*( ) (CL SPI).

Fields are returned as a set of OID-value pairs. The OID identifies the CRL record field and the data format of the value extracted from that field. The Certificate Library module defines and uses a preferred data format for returning field values in this function.

Each CRL record may be digitally signed when it is added to the CRL using the function *CSSM_CL_CrlAddCert*( ) (CSSM API) or *CL_CrlAddCert*( ) (CL SPI). This function does not perform any signature verification on the CRL record.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlHandle (input)
A handle identifying a CRL that the application has temporarily cached with the Certificate Library module. The referenced CRL must contain the CRL record identified by *CrlRecordIndex*.

CrlRecordIndex (input)
An index value identifying a particular revocation record in a cached CRL.

NumberOfFields (output)
The number of OID-value pairs returned by this function.

Fields (output)
A pointer to an array of CSSM_FIELD structures, describing the contents of the pre-selected CRL record using OID-value pairs. The field list is allocated by the service provider and must be deallocated by the application by calling *CSSM_CL_FreeFields*( ) (CSSM API) or. *CL_FreeFields*( ) (CL SPI).

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CACHE_HANDLE
CSSMERR_CL_INVALID_CRL_INDEX

SEE ALSO

For the CSSM API:
CSSM_CL_IsCertInCachedCrl()
CSSM_CL_CrlCache()
CSSM_CL_CrlAbortCache()
CSSM_CL_FreeFields()

For the CLI SPI:
CL_IsCertInCachedCrl()
CL_CrlCache()
CL_CrlAbortCache()
CL_FreeFields()

NAME

CSSM_CL_CrlAbortCache for the CSSM API
CL_CrlAbortCache for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlAbortCache
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle)

SPI:
CSSM_RETURN CSSMCLI CL_CrlAbortCache
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CrlHandle)
```

DESCRIPTION

This function terminates a CRL cache handle created and returned by the function *CSSM_CL_CrlCache*() (CSSM API) or *CL_CrlCache*() (CL SPI). The Certificate Library module releases all cache space and state information associated with the cached CRL.

PARAMETERS

CLHandle (input)
The handle that describes the certificate library module used to perform this function.

CrlHandle (input)
The handle that identifies the cached CRL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CACHE_HANDLE

SEE ALSO

For the CSSM API:
CSSM_CL_CrlCache()

For the CLI SPI:
CL_CrlCache()

NAME

CSSM_CL_CrlDescribeFormat for the CSSM API
CL_CrlDescribeFormat for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_CrlDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfOids,
    CSSM_OID_PTR *OidList)

SPI:
CSSM_RETURN CSSMCLI CL_CrlDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfOids,
    CSSM_OID_PTR *OidList)
```

DESCRIPTION

This function returns a list of the CSSM_OID values this certificate library module uses to name and reference fields of a CRL. Multiple CSSM_OID values can correspond to a single field. These OIDs can be provided as input to *CSSM_CL_CrlGetFirstFieldValue*( ) (CSSM API) or *CL_CrlGetFirstFieldValue*( ) (CL SPI) to retrieve field values from the CRL. The OID also implies the data format of the returned value. When multiple OIDs name the same field of a CRL, those OIDs have a different return data formats associated with them.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfOids (output)
The length of the returned array of OIDs.

OidList (output)
A pointer to the array of CSSM_OIDs which represent the supported CRL format. The OID list is allocated by the service provider and must be de-allocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

10.6 Extensibility Functions

The man-page definition for the Certificate Library *pass-through* extensibility function is presented in this section.

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.

CL_PassThrough

NAME

CSSM_CL_PassThrough for the CSSM API
CL_PassThrough for the CL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_CL_PassThrough
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)

SPI:
CSSM_RETURN CSSMCLI CL_PassThrough
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)
```

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/optional)
A pointer to a module implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested CL module.

OutputParams (output/optional)
A pointer to a module, implementation-specific structure containing the output data. The service provider allocates the memory for the structure and sub-structures. The application should free the memory for the structure and sub-structures.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_CL_INVALID_CONTEXT_HANDLE
CSSMERR_CL_INVALID_PASSTHROUGH_ID
CSSMERR_CL_INVALID_DATA

Technical Standard

Part 7:

Data Storage Library (DL) Services

The Open Group

Chapter 11

Data Storage Library Services

11.1 Introduction

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. The DL provides access to these data stores by translating calls from the DL interface (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.

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 single DL module can be tightly tied to a Certificate library (CL) and/or Trust Policy (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.

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.

11.2 CSSM API

CSSM stores and manages meta-information about a DL in the Module Directory Services (MDS). This information describes the storage and retrieval capabilities of a DL. Applications can query MDS to obtain information about the available DLs and attach to a DL that provides the needed services. Each DL service is responsible for acquiring and managing meta-data about each application-defined data store that it creates. Applications use the *CSSM_DL_GetDbNames*( ) function to obtain status information about each data store managed by the DL. The DL can store this meta-data using MDS or some other mechanism.

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 CL module can be used to parse the opaque certificate object. The DL module defines a default set of parsing functions.

11.3 DL SPI

11.3.1 Add-In Module

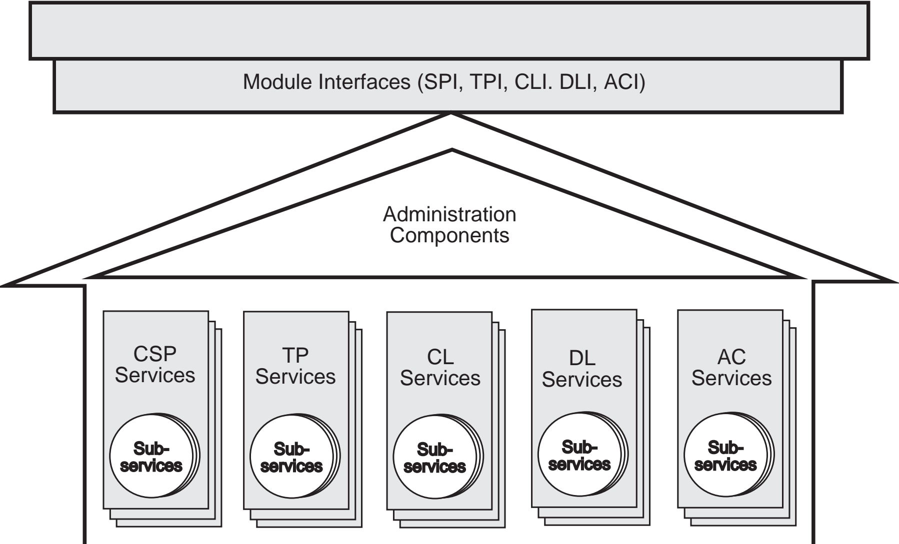


Figure 11-1 CDSA Add-In Module Structure

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, in this case the Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. 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.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions that 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 **Part 14** of this Technical Standard.

11.3.2 Operation

Applications are able to obtain information about the available DL services by using the Module Directory Services (MDS) 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 should provide information about the data stores that it has access to. Applications can obtain information about these data stores by using the *CSSM_GetDbInfo*() function. 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. The default add-in modules set by the data store creator are used for parsing.

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.

11.4 Interoperability

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.

11.5 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. Data store access can be protected by a passphrase, which can be changed by the owner of the old passphrase. 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.

11.6 Data Storage Data Structures

11.6.1 CSSM_DL_HANDLE

A unique identifier for an attached module that provide data storage library services.

```
typedef CSSM_MODULE_HANDLE CSSM_DL_HANDLE; /* data storage library Handle */
```

11.6.2 CSSM_DB_HANDLE

A unique identifier for an open data store.

```
typedef CSSM_MODULE_HANDLE CSSM_DB_HANDLE; /* Data storage Handle */
```

11.6.3 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.

11.6.4 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.

11.6.5 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_AS_INTEGER = 2,
} CSSM_DB_ATTRIBUTE_NAME_FORMAT, *CSSM_DB_ATTRIBUTE_NAME_FORMAT_PTR;
```

11.6.6 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.

It is important to note that the value returned from a database might not have the same binary value as the value inserted into the database. The value returned is only guaranteed to have the same value in the context of its attribute format. For instance, strings may acquire or loose NULL termination or the size of integers may change.

```
typedef enum cssm_db_attribute_format {
    CSSM_DB_ATTRIBUTE_FORMAT_STRING = 0,
    CSSM_DB_ATTRIBUTE_FORMAT_SINT32 = 1,
    CSSM_DB_ATTRIBUTE_FORMAT_UINT32 = 2,
    CSSM_DB_ATTRIBUTE_FORMAT_BIG_NUM = 3,
    CSSM_DB_ATTRIBUTE_FORMAT_REAL = 4,
    CSSM_DB_ATTRIBUTE_FORMAT_TIME_DATE = 5,
    CSSM_DB_ATTRIBUTE_FORMAT_BLOB = 6,
    CSSM_DB_ATTRIBUTE_FORMAT_MULTI_UINT32 = 7,
    CSSM_DB_ATTRIBUTE_FORMAT_COMPLEX = 8
} CSSM_DB_ATTRIBUTE_FORMAT, *CSSM_DB_ATTRIBUTE_FORMAT_PTR;
```

Definitions

STRING
: A string containing only printable characters, NULL terminator is optional. Greater than and less than operations are performed by comparing the binary value of each character (strcmp). The character format is platform specific.

SINT32
: A signed, 32-bit integer. Endian-ness is platform specific. 8-bit and 16-bit integers are converted to 32-bit integers with sign extension. All other sized integers are invalid.

UINT32
: An unsigned, 32-bit integer. Endian-ness is platform specific. 8-bit and 16-bit integers are converted to 32-bit integers without sign extension. All other sized integers are invalid.

BIG_NUM
: This is a sign-magnitude little-endian integer of arbitrary size. The first bit represents the sign of the number (0 == positive, 1 == negative, zero is non-deterministic), all other bits represent the magnitude. Padding zeros are allowed.

REAL
: A double precision IEEE floating point number (size = 8 bytes). Single precision IEEE floating point numbers (size = 4 bytes) are interpolated to doubles. Not a number (NaN) is

not a valid input.

TIME_DATE
: A representation of generalized time: A NULL terminated ASCII string representation of Zulu Time and Data of size 16 character and following the format: "YYYYMMDDhhmmssZ"

BLOB
: An opaque block of data. Greater than and less than operations are preformed by comparing the binary value of each byte.

MULTI_UINT32
: An array of uint32s. The length of this structure must be a multiple of four. Greater than and less than operations are performed by comparing the binary value of each uint32.

COMPLEX
: A non-standard or complex structure. The type is further defined by the attribute's name. (for example, if *AttributeName* = APP_DOMAIN_STRUCTURED_NAME, then the implied type is a application-defined structure containing a name). Use of this type is not recommended.

11.6.7 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. 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 AttributeOID;        /* e.g., CSSMOID_RECORDLABEL */
        uint32 AttributeID;
    } Label;
    CSSM_DB_ATTRIBUTE_FORMAT AttributeFormat;
} CSSM_DB_ATTRIBUTE_INFO, *CSSM_DB_ATTRIBUTE_INFO_PTR;
```

Definition

AttributeNameFormat
: Indicates which of the defined formats was selected to represent the attribute name.

Label
: A character string representation of the attribute name. or an OID representation of the attribute name as indicated by the value of *AttributeNameFormat.*

AttributeName
: A character string representation of the attribute name.

AttributeOID
: An OID representation of the attribute name.

AttributeID
: An integer 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.

11.6.8 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;
    uint32 NumberOfValues;
    CSSM_DATA_PTR 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.

NumberOfValues
An integer value indicating the number of individual values contained in Value. If Value is a multi-valued attribute, this value can be greater than one. For single-valued attributes, this integer must be 1.

Value
The data-present value(s) assigned to the attribute.

11.6.9 CSSM_DB_RECORDTYPE

These constants partition the space of record type names into three primary groups:

- Record types for schema management
- Record types recognized and documented in this specification for application use
- Record types defined independently by the industry at large for application use

All record types defined in this specification are defined in the Schema Management name space and the Open Group name space.

```
typedef uint32 CSSM_DB_RECORDTYPE;

/* Schema Management Name Space Range Definition*/
#define CSSM_DB_RECORDTYPE_SCHEMA_START          (0x00000000)
#define CSSM_DB_RECORDTYPE_SCHEMA_END
(CSSM_DB_RECORDTYPE_SCHEMA_START + 4)

/* Open Group Application Name Space Range Definition*/
#define CSSM_DB_RECORDTYPE_OPEN_GROUP_START      (0x0000000A)
#define CSSM_DB_RECORDTYPE_OPEN_GROUP_END
(CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 8)

/* Industry At Large Application Name Space Range Definition */
#define CSSM_DB_RECORDTYPE_APP_DEFINED_START     (0x80000000)
#define CSSM_DB_RECORDTYPE_APP_DEFINED_END       (0xffffffff)

/* Record Types defined in the Schema Management Name Space */
#define CSSM_DL_DB_SCHEMA_INFO
(CSSM_DB_RECORDTYPE_SCHEMA_START + 0)
```

```
#define CSSM_DL_DB_SCHEMA_INDEXES
(CSSM_DB_RECORDTYPE_SCHEMA_START + 1)
#define CSSM_DL_DB_SCHEMA_ATTRIBUTES
(CSSM_DB_RECORDTYPE_SCHEMA_START + 2)
#define CSSM_DL_DB_SCHEMA_PARSING_MODULE
(CSSM_DB_RECORDTYPE_SCHEMA_START + 3)

/* Record Types defined in the Open Group Application Name Space */
#define CSSM_DL_DB_RECORD_ANY
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 0)
#define CSSM_DL_DB_RECORD_CERT
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 1)
#define CSSM_DL_DB_RECORD_CRL
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 2)
#define CSSM_DL_DB_RECORD_POLICY
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 3)
#define CSSM_DL_DB_RECORD_GENERIC
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START +4)
#define CSSM_DL_DB_RECORD_PUBLIC_KEY
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 5)
#define CSSM_DL_DB_RECORD_PRIVATE_KEY
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 6)
#define CSSM_DL_DB_RECORD_SYMMETRIC_KEY
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 7)
#define CSSM_DL_DB_RECORD_ALL_KEYS
        (CSSM_DB_RECORDTYPE_OPEN_GROUP_START + 8)
```

Definitions for Schema Management Record Types

CSSM_DL_DB_SCHEMA_RELATIONS is a relation containing one record for each relation defined for the database. All fields are searchable. *RecordType* is the primary database key for this relation.

The schema relations can be queried by users and applications, but cannot be modified by users or applications.

	Field Name	Field Data Type	Comment
*	RelationID	UINT32	A unique integer value identifying a relation.
	RelationName	STRING	The relation name in ASCII text.

CSSM_DL_DB_SCHEMA_ATTRIBUTES is a relation containing one record for each attribute defined for the database. All fields are searchable. The starred(*) fields form the primary database key for this relation.

	Field Name	Field Data Type	Comment
*	RelationID	UINT32	A unique integer value identifying a relation.
*	AttributeID	UINT32	A number identifying an attribute in the relation identified by (RelationId)
	AttributeNameFormat	UINT32	Format of AttributeName
	AttributeName	STRING	Name of attribute
	AttributeNameID	BLOB	Name of attribute expressed as an infinite precision number (aka OID).

	AttributeFormat	UINT32	Data type of values associated with the attribute

CSSM_DL_DB_SCHEMA_INDEXES is a relation containing one record for each index defined for the database. All fields are searchable. The starred(*) fields form the primary database key for this relation.

	Field Name	Field Data Type	Comment
*	RelationID	UINT32	A unique integer value identifying a relation.
*	IndexID	UINT32	A number uniquely identifying an index. Unique indexes will use the same *IndexID* for each attribute (*AttributeID*) comprising the concatenated key of the unique index.
*	AttributeID	UINT32	An integer value uniquely identifying an attribute within the relation identified by RelationID.
	IndexType	UINT32	Type of index (part of the unique index or a non-unique index).
	IndexedDataLocation	UINT32	Source of the information used to create the index

CSSM_DL_DB_SCHEMA_PARSING_MODULE is a relation containing one record for each attribute in a relation in the database for attributes using parsing modules. All fields are searchable. The starred(*) fields form the primary database key for this relation. If no parsing modules are defined for an attribute, then no entry will be found in the table.

	Field Name	Field Data Type	Comment
*	RelationID	UINT32	A unique integer value identifying a relation.
*	AttributeID	UINT32	Attribute to which the parsing module is associated.
	ModuleID	BLOB	GUID of the module used to parse the data object
	AddinVersion	STRING	Version of the module used to parse the data object
	SSID	UINT32	SubserviceID of the subservice used to parse the data object
	SubserviceType	UINT32	Type of module used to parse the data object

Definitions for Open Group Application Record Types

The meaning of each record types is defined as follows:

Record Type	CSSM Type	Comment
CSSM_DL_DB_RECORD_CERT	CSSM_DATA	An opaque structure whose format is defined by the type of certificate stored in the structure.
CSSM_DL_DB_RECORD_CRL	CSSM_DATA	An opaque structure whose format is defined by the type of CRL stored in the structure.

CSSM_DL_DB_RECORD_POLICY	CSSM_DATA	An opaque structure whose format is defined by the type of policy stored in the structure.
CSSM_DL_DB_RECORD_GENERIC	CSSM_DATA	An opaque structure whose format is defined by agreement between the application and the service provider module
CSSM_DL_DB_RECORD_PUBLIC_KEY	CSSM_KEY	A CSSM_KEY structure with an instantiated CSSM_KEY_HEADER describing the attributes of the key.
CSSM_DL_DB_RECORD_PRIVATE_KEY	CSSM_KEY	A CSSM_KEY structure with an instantiated CSSM_KEY_HEADER describing the attributes of the key.
CSSM_DL_DB_RECORD_SYMMETRIC_KEY	CSSM_KEY	A CSSM_KEY structure with an instantiated CSSM_KEY_HEADER describing the attributes of the key.
CSSM_DL_DB_RECORD_ALL_KEYS	CSSM_KEY	A CSSM_KEY structure with an instantiated CSSM_KEY_HEADER describing the attributes of the key.

Schema for DL Records of Type CSSM_DL_DB_RECORD_CERT

The following schema defines the set of attributes for DL records of type CSSM_DL_DB_RECORD_CERT.

Attribute Name	**Attribute Type**	**Attribute Description**
CertType	CSSM_CERT_TYPE	One of the values defined for CSSM_CERT_TYPE.
CertEncoding	CSSM_CERT_ENCODING	One of the values defined for CSSM_CERT_ENCODING.
PrintName	CSSM_Data (max length 16 characters)	The *PrintName* attribute required in all DL-stored records.
Alias	CSSM_Data (max length 8 bytes)	The *Alias* attribute required in all DL-stored records.

Schema for DL Records of Type CSSM_DL_DB_RECORD_CRL

The following schema defines the set of attributes for DL records of type CSSM_DL_DB_RECORD_CRL.

Attribute Name	Attribute Type	Attribute Description
CrlType	CSSM_CRL_TYPE	One of the values defined for CSSM_CRL_TYPE.
CrlEncoding	CSSM_CRL_ENCODING	One of the values defined for CSSM_CRL_ENCODING.
PrintName	CSSM_Data (max length 16 characters)	The *PrintName* attribute required in all DL-stored records.
Alias	CSSM_Data (max length 8 bytes)	The *Alias* attribute required in all DL-stored records.

Schema for DL Records of Type CSSM_DL_DB_RECORD_POLICY

The following schema defines the set of attributes for DL records of type CSSM_DL_DB_RECORD_POLICY.

Attribute Name	Attribute Type	Attribute Description
PolicyName	CSSM_OID	One of the values defined by the policy domain.
PrintName	CSSM_Data (max length 16 characters)	The *PrintName* attribute required in all DL-stored records.
Alias	CSSM_Data (max length 8 bytes)	The *Alias* attribute required in all DL-stored records.

Schema for DL Records of Type CSSM_DL_DB_RECORD_GENERIC

The following schema defines the set of attributes for DL records of type CSSM_DL_DB_RECORD_GENERIC.

Attribute Name	Attribute Type	Attribute Description
PrintName	CSSM_Data (max length 16 characters)	The *PrintName* attribute required in all DL-stored records.
Alias	CSSM_Data (max length 8 bytes)	The *Alias* attribute required in all DL-stored records.

Schema for DL Records of Type KEY

The following schema defines the set of attributes for DL records of type CSSM_DL_DB_RECORD_PUBLIC_KEY, CSSM_DL_DB_RECORD_PRIVATE_KEY, and CSSM_DL_DB_RECORD_SYMMETRIC_KEY.

Attribute Name	**Attribute Type**	**Attribute Description**
KeyClass	CSSM_DB_RECORDTYPE	One of the following values: CSSM_DL_DB_RECORD_PUBLIC_KEY, CSSM_DL_DB_RECORD_PRIVATE_KEY, CSSM_DL_DB_RECORD_SYMMETRIC_KEY
PrintName	CSSM_Data (max length 16 characters)	The *PrintName* attribute required in all DL-stored records. This attribute could be replaced by the value of *Label* attribute.
Alias	CSSM_Data (max length 8 bytes)	The *Alias* attribute required in all DL-stored records. This attribute could be replaced by the value of *ApplicationTag* attribute.
Permanent	CSSM_BOOL	Indicates whether the key is stored permanently or temporarily in the device.
Private	CSSM_BOOL	Indicates whether user authentication is required to access the key.
Modifiable	CSSM_BOOL	Attributes describing the key can be modified
Label	CSSM_DATA	User-defined label assigned by the user who created the key.
ApplicationTag	CSSM_DATA	Application-defined string assigned by the application creating the key.
KeyCreator	CSSM_GUID	GUID of the CSP that created the key. This could be a virtual attribute for a multi-service provider with CSP and DL in one module.
KeyType	CSSM_ALGORITHMS	Algorithm Identifier for a key type
KeySizeInBits	uint32	Size of the key in bits
EffectiveKeySize	uint32	Effective size of the key. This could be a virtual attribute that is computed on demand.
StartDate	CSSM_DATE	Starting validity date
EndDate	CSSM_DATE	Ending validity date
Sensitive	CSSM_BOOL	Key can not be revealed outside of the device in an unwrapped state
AlwaysSensitive	CSSM_BOOL	The *Sensitive* attribute has always been CSSM_TRUE, and the key was generated by the CSP.
Extractable	CSSM_BOOL	Key can be removed from the token in wrapped or unwrapped form
NeverExtractable	CSSM_BOOL	The *Extractable* attribute has never been CSSM_TRUE
Encrypt	CSSM_BOOL	Key is usable for encryption
Decrypt	CSSM_BOOL	Key is usable for decryption
Derive	CSSM_BOOL	Key is usable for derivation
Sign	CSSM_BOOL	Key is usable for signature or MAC generation

Verify	CSSM_BOOL	Key is usable for signature verification
SignRecover	CSSM_BOOL	Key is usable to generate signatures with message recovery (private key encrypt)
VerifyRecover	CSSM_BOOL	Key is usable to verify signatures with message recovery (public key decrypt)
Wrap	CSSM_BOOL	Key can be used to wrap other keys
Unwrap	CSSM_BOOL	Key can be used to unwrap other keys

11.6.10 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	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

11.6.11 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.

```
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.

11.6.12 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.

11.6.13 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 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. If no parsing module is specified for this *RecordType*, then the *ModuleSubserviceUid* must be zero.

11.6.14 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;
```

11.6.15 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;
```

11.6.16 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. The primary key is formed by concatenating the attributes specifying the unique indexes. Non-unique indexes can only be on multiple single attributes.

```
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 set of one or more attributes that form the index.

11.6.17 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.

11.6.18 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.

11.6.19 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 */
```

11.6.20 CSSM_DB_MODIFY_MODE

Constants of this type define the types of modifications that can be performed on record attributes using the function *CSSM_DL_DataModify*().

```
typedef uint32 CSSM_DB_MODIFY_MODE;

#define CSSM_DB_MODIFY_ATTRIBUTE_NONE (0)
#define CSSM_DB_MODIFY_ATTRIBUTE_ADD
    (CSSM_DB_MODIFY_ATTRIBUTE_NONE + 1)
#define CSSM_DB_MODIFY_ATTRIBUTE_DELETE
    (CSSM_DB_MODIFY_ATTRIBUTE_NONE + 2)
#define CSSM_DB_MODIFY_ATTRIBUTE_REPLACE
```

```
(CSSM_DB_MODIFY_ATTRIBUTE_NONE + 3)
```

11.6.21 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.

```
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_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.

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.

11.6.22 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_LESS_THAN = 2,
    CSSM_DB_GREATER_THAN = 3,
    CSSM_DB_CONTAINS = 4,
    CSSM_DB_CONTAINS_INITIAL_SUBSTRING = 5,
    CSSM_DB_CONTAINS_FINAL_SUBSTRING = 6
} CSSM_DB_OPERATOR, *CSSM_DB_OPERATOR_PTR;
```

Some logical operator can only be applied to selected attribute types. The table below defines the valid attribute types for each logical operator.

Operator	**Description**	**CSSM_DB_ ATTRIBUTE_FORMAT**
CSSM_DB_EQUAL	**Is the predicate equal to the attribute**	**All**
CSSM_DB_NOT_EQUAL	Is the predicate not equal to the attribute	All
CSSM_DB_LESS_THAN	Is the predicate less than the attribute	All, except Multi-Uint32, Blob, and Complex
CSSM_DB_GREATER_THAN	Is the predicate greater than the attribute	All, except Multi-Uint32, Blob, and Complex
CSSM_DB_CONTAINS	Is the predicate contained in the attribute	String, Blob, Multi-Uint32
CSSM_DB_CONTAINS_INITIAL_SUBSTRING	Is the start of the attribute equal to the predicate	String, Blob, Multi-Uint32
CSSM_DB_CONTAINS_FINAL_SUBSTRING	Is the end of the attribute equal to the predicate	String, Blob, Multi-Uint32

11.6.23 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;
```

11.6.24 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.

11.6.25 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. 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.

11.6.26 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 (0x01)
```

Flag Identifier	Meaning
CSSM_QUERY_RETURN_DATA	Valid for key records only. If this flag is set, the DL will attempt to return a CSSM_KEY structure with the actual key material as plaintext. If it is not set, then the DL will return a CSSM_KEY structure with a key reference.

11.6.27 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.

11.6.28 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 */
    CSSM_DL_MEMORY = 6,
    CSSM_DL_REMOTEDIR = 7
} CSSM_DLTYPE, *CSSM_DLTYPE_PTR;
```

11.6.29 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 the PKCS#11 data store.

11.6.30 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)
```

11.6.31 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.

11.6.32 CSSM_DB_RETRIEVAL_MODES

This defines the retrieval modes for *CSSM_DL_DataGetFirst*() operations. The data storage module supports one of two retrieval models:

- Transactional
  All query results are determined at initial query evaluation. Results do not change during an incremental retrieval process.
- File System Scan
  Query results are selected during the incremental retrieval process. Records matching the query may be added to or deleted from the underlying data store during the iterative retrieval. The caller may receive the new matching records and not receive the deleted records.

```
typedef uint32 CSSM_DB_RETRIEVAL_MODES;

#define CSSM_DB_TRANSACTIONAL_MODE (0)
#define CSSM_DB_FILESYSTEMSCAN_MODE (1)
```

11.6.33 CSSM_DB_SCHEMA_ATTRIBUTE_INFO

```
typedef struct cssm_db_schema_attribute_info {
    uint32 AttributeId;
    char *AttributeName;
    CSSM_OID AttributeNameID;
    CSSM_DB_ATTRIBUTE_FORMAT DataType;
} CSSM_DB_SCHEMA_ATTRIBUTE_INFO, *CSSM_DB_SCHEMA_ATTRIBUTE_INFO_PTR;
```

11.6.34 CSSM_DB_SCHEMA_INDEX_INFO

```
typedef struct cssm_db_schema_index_info {
    uint32 AttributeId;
    uint32 IndexId;
    CSSM_DB_INDEX_TYPE IndexType;
    CSSM_DB_INDEXED_DATA_LOCATION IndexedDataLocation;
} CSSM_DB_SCHEMA_INDEX_INFO, *CSSM_DB_SCHEMA_INDEX_INFO_PTR;
```

11.7 Error Codes and Error Values

This section defines Error Values that can be returned by DL operations.

The Error Values that can be returned by DL functions can be either derived from the Common Error Codes defined in Appendix A on page 925, or from a Common set that more than one DL function can return, or they are specific to the DL function.

The DL functions defined in this section list all the DL Error Values in the Common set, plus any Error Values that are specific to the function.

11.7.1 DL Error Values Derived from Common Error Codes

```
#define CSSMERR_DL_INTERNAL_ERROR \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INTERNAL_ERROR)
#define CSSMERR_DL_MEMORY_ERROR  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_MEMORY_ERROR)
#define CSSMERR_DL_MDS_ERROR  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_MDS_ERROR)
#define CSSMERR_DL_INVALID_POINTER  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_POINTER)
#define CSSMERR_DL_INVALID_INPUT_POINTER  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_INPUT_POINTER)
#define CSSMERR_DL_INVALID_OUTPUT_POINTER  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_OUTPUT_POINTER)
#define CSSMERR_DL_FUNCTION_NOT_IMPLEMENTED  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED)
#define CSSMERR_DL_SELF_CHECK_FAILED  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_SELF_CHECK_FAILED)
#define CSSMERR_DL_OS_ACCESS_DENIED  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OS_ACCESS_DENIED)
#define CSSMERR_DL_FUNCTION_FAILED  \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_FUNCTION_FAILED)
#define CSSMERR_DL_INVALID_DL_HANDLE \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_DL_HANDLE)
```

11.7.2 DL Error Values Derived from ACL-based Error Codes

This section lists DL error values derived from convertible ACL-based error codes.

```
#define CSSMERR_DL_OPERATION_AUTH_DENIED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OPERATION_AUTH_DENIED)
#define CSSMERR_DL_OBJECT_USE_AUTH_DENIED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OBJECT_USE_AUTH_DENIED)
#define CSSMERR_DL_OBJECT_MANIP_AUTH_DENIED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OBJECT_MANIP_AUTH_DENIED)
#define CSSMERR_DL_OBJECT_ACL_NOT_SUPPORTED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OBJECT_ACL_NOT_SUPPORTED)
#define CSSMERR_DL_OBJECT_ACL_REQUIRED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_OBJECT_ACL_REQUIRED)
#define CSSMERR_DL_INVALID_ACCESS_CREDENTIALS \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACCESS_CREDENTIALS)
#define CSSMERR_DL_INVALID_ACL_BASE_CERTS \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_BASE_CERTS)
#define CSSMERR_DL_ACL_BASE_CERTS_NOT_SUPPORTED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_BASE_CERTS_NOT_SUPPORTED)
#define CSSMERR_DL_INVALID_SAMPLE_VALUE \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_SAMPLE_VALUE)
#define CSSMERR_DL_SAMPLE_VALUE_NOT_SUPPORTED \
```

```
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_SAMPLE_VALUE_NOT_SUPPORTED)
#define CSSMERR_DL_INVALID_ACL_SUBJECT_VALUE \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_SUBJECT_VALUE)
#define CSSMERR_DL_ACL_SUBJECT_TYPE_NOT_SUPPORTED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_SUBJECT_TYPE_NOT_SUPPORTED)
#define CSSMERR_DL_INVALID_ACL_CHALLENGE_CALLBACK \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_CHALLENGE_CALLBACK)
#define CSSMERR_DL_ACL_CHALLENGE_CALLBACK_FAILED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_CHALLENGE_CALLBACK_FAILED)
#define CSSMERR_DL_INVALID_ACL_ENTRY_TAG \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_ENTRY_TAG)
#define CSSMERR_DL_ACL_ENTRY_TAG_NOT_FOUND \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_ENTRY_TAG_NOT_FOUND)
#define CSSMERR_DL_INVALID_ACL_EDIT_MODE \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_ACL_EDIT_MODE)
#define CSSMERR_DL_ACL_CHANGE_FAILED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_CHANGE_FAILED)
#define CSSMERR_DL_INVALID_NEW_ACL_ENTRY \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_NEW_ACL_ENTRY)
#define CSSMERR_DL_INVALID_NEW_ACL_OWNER \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_NEW_ACL_OWNER)
#define CSSMERR_DL_ACL_DELETE_FAILED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_DELETE_FAILED)
#define CSSMERR_DL_ACL_REPLACE_FAILED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_REPLACE_FAILED)
#define CSSMERR_DL_ACL_ADD_FAILED \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_ACL_ADD_FAILED)
```

11.7.3 DL Error Values for Specific Data Types

This section lists DL error values derived from Common Error Codes for specific data types.

```
#define CSSMERR_DL_INVALID_DB_HANDLE \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_DB_HANDLE)
#define CSSMERR_DL_INVALID_PASSTHROUGH_ID \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_PASSTHROUGH_ID)
#define CSSMERR_DL_INVALID_NETWORK_ADDR \
    (CSSM_DL_BASE_ERROR+CSSM_ERRCODE_INVALID_NETWORK_ADDR)
```

11.7.4 General DL Error Values

These values can be returned from any DL function.

```
#define CSSM_DL_BASE_DL_ERROR \
    (CSSM_DL_BASE_ERROR+CSSM_ERRORCODE_COMMON_EXTENT)

#define CSSMERR_DL_DATABASE_CORRUPT (CSSM_DL_BASE_DL_ERROR+1)
```

The database file or other data form is corrupt

11.7.5 DL Specific Error Values

```
#define CSSMERR_DL_INVALID_RECORD_INDEX (CSSM_DL_BASE_DL_ERROR+8)
```

A record index in the DbInfo structure is invalid

```
#define CSSMERR_DL_INVALID_RECORDTYPE (CSSM_DL_BASE_DL_ERROR+9)
```

Record types from DbInfo's arrays do not match

```
#define CSSMERR_DL_INVALID_FIELD_NAME (CSSM_DL_BASE_DL_ERROR+10)
```

Attribute or index name is an illegal name

```
#define CSSMERR_DL_UNSUPPORTED_FIELD_FORMAT (CSSM_DL_BASE_DL_ERROR+11)
```

A field's format (data type) is not supported

```
#define CSSMERR_DL_UNSUPPORTED_INDEX_INFO (CSSM_DL_BASE_DL_ERROR+12)
```

Requested IndexInfo struct is not supported

```
#define CSSMERR_DL_UNSUPPORTED_LOCALITY (CSSM_DL_BASE_DL_ERROR+13)
```

The value of DbInfo->IsLocal is not supported

```
#define CSSMERR_DL_UNSUPPORTED_NUM_ATTRIBUTES (CSSM_DL_BASE_DL_ERROR+14)
```

Unsupported number of attributes specified

```
#define CSSMERR_DL_UNSUPPORTED_NUM_INDEXES (CSSM_DL_BASE_DL_ERROR+15)
```

Unsupported number of indexes

```
#define CSSMERR_DL_UNSUPPORTED_NUM_RECORDTYPES (CSSM_DL_BASE_DL_ERROR+16)
```

Unsupported number of record types

```
#define CSSMERR_DL_UNSUPPORTED_RECORDTYPE (CSSM_DL_BASE_DL_ERROR+17)
```

Requested record type is not supported

```
#define CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE (CSSM_DL_BASE_DL_ERROR+18)
```

A record attribute or index was specified multiple times with differing information

```
#define CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT (CSSM_DL_BASE_DL_ERROR+19)
```

The field format specified is different from the field format of that attribute

```
#define CSSMERR_DL_INVALID_PARSING_MODULE (CSSM_DL_BASE_DL_ERROR+20)
```

A parsing module in the DB Info is invalid

```
#define CSSMERR_DL_INVALID_DB_NAME (CSSM_DL_BASE_DL_ERROR+22)
```

The Database name is invalid.

```
#define CSSMERR_DL_DATASTORE_DOESNOT_EXIST (CSSM_DL_BASE_DL_ERROR+23)
```

The specified datastore does not exists

```
#define CSSMERR_DL_DATASTORE_ALREADY_EXISTS (CSSM_DL_BASE_DL_ERROR+24)
```

The specified datastore already exists

```
#define CSSMERR_DL_DB_LOCKED (CSSM_DL_BASE_DL_ERROR+25)
```

Database is currently locked for exclusive update

```
#define CSSMERR_DL_DATASTORE_IS_OPEN (CSSM_DL_BASE_DL_ERROR+26)
```

The database is currently open

```
#define CSSMERR_DL_RECORD_NOT_FOUND (CSSM_DL_BASE_DL_ERROR+27)
```

The record does not exist

```
#define CSSMERR_DL_MISSING_VALUE (CSSM_DL_BASE_DL_ERROR+28)
```

Missing needed attribute or data value

```
#define CSSMERR_DL_UNSUPPORTED_QUERY (CSSM_DL_BASE_DL_ERROR+29)
```

An unsupported query was specified

```
#define CSSMERR_DL_UNSUPPORTED_QUERY_LIMITS (CSSM_DL_BASE_DL_ERROR+30)
```

The requested query limits are not supported

```
#define CSSMERR_DL_UNSUPPORTED_NUM_SELECTION_PREDS \
    (CSSM_DL_BASE_DL_ERROR+31)
```

The number of selection predicates is not supported

```
#define CSSMERR_DL_UNSUPPORTED_OPERATOR (CSSM_DL_BASE_DL_ERROR+33)
```

An unsupported operator was requested

```
#define CSSMERR_DL_INVALID_RESULTS_HANDLE (CSSM_DL_BASE_DL_ERROR+34)
```

Invalid results handle

```
#define CSSMERR_DL_INVALID_DB_LOCATION (CSSM_DL_BASE_DL_ERROR+35)
```

The Database Location is not valid

```
#define CSSMERR_DL_INVALID_ACCESS_REQUEST (CSSM_DL_BASE_DL_ERROR+36)
```

Unrecognized access type

```
#define CSSMERR_DL_INVALID_INDEX_INFO (CSSM_DL_BASE_DL_ERROR+37)
```

Invalid index information passed

```
#define CSSMERR_DL_INVALID_SELECTION_TAG (CSSM_DL_BASE_DL_ERROR+38)
```

Invalid selection tag

```
#define CSSMERR_DL_INVALID_NEW_OWNER (CSSM_DL_BASE_DL_ERROR+39)
```

Owner definition is invalid

```
#define CSSMERR_DL_INVALID_RECORD_UID (CSSM_DL_BASE_DL_ERROR+40)
```

The data inside the unique record identifier is not valid

```
#define CSSMERR_DL_INVALID_UNIQUE_INDEX_DATA (CSSM_DL_BASE_DL_ERROR+41)
```

The modification would have caused the new primary key to have a value that is already in use

```
#define CSSMERR_DL_INVALID_MODIFY_MODE (CSSM_DL_BASE_DL_ERROR+42)
```

The specified modification mode is undefined or could not be applied to the record attributes identified for modification.

```
#define CSSMERR_DL_INVALID_OPEN_PARAMETERS (CSSM_DL_BASE_DL_ERROR+43)
```

The open parameters are not valid

```
#define CSSMERR_DL_RECORD_MODIFIED (CSSM_DL_BASE_DL_ERROR+44)
```

The record was changed by someone since the last time it was retrieved from the DL. The attributes and data that you requested were successfully retrieved; however consider retrieving all attributes and data again.

```
#define CSSMERR_DL_ENDOFDATA (CSSM_DL_BASE_DL_ERROR+45)
```

There are no more records satisfying the query.

```
#define CSSMERR_DL_INVALID_QUERY (CSSM_DL_BASE_DL_ERROR+46)
```

The specified CSSM_QUERY was not valid (possibly because a selection predicate it contains has an *Attribute* with *NumberOfValues* not equal to 1).

```
#define CSSMERR_DL_INVALID_VALUE (CSSM_DL_BASE_DL_ERROR+47)
```

A value specified for an attribute was not of the correct form.

```
#define CSSMERR_DL_MULTIPLE_VALUES_UNSUPPORTED (CSSM_DL_BASE_DL_ERROR+48)
```

This DL does not support multiple values per attribute.

```
#define CSSMERR_DL_STALE_UNIQUE_RECORD (CSSM_DL_BASE_DL_ERROR+49)
```

The record was changed by someone since the last time it was retrived from the DL. Call *CSSM_DL_DataGetFromUniqueRecordId*( ) to update the *UniqueRecordId*.

11.8 Data Storage Library Operations

The man-page definitions for Data Storage Library Operations are presented in this section.

NAME

CSSM_DL_Authenticate, for the CSSM API
DL_Authenticate, for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_Authenticate
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_ACCESS_CREDENTIALS *AccessCred)

SPI:
CSSM_RETURN CSSMDLI DL_Authenticate
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_ACCESS_CREDENTIALS *AccessCred)
```

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.

AccessCred (input)
A pointer to the set of one or more credentials being presented for authentication by the caller. The credentials can apply to the DL module in general or to a particular data store managed by this service module. The credentials required for creating new data stores is defined by the DL and recorded in a record in the MDS Primary DL relation. The required set of credentials to access a particular data store is defined by the *DbInfo* record containing meta-data for the specified data store.

The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function the DL can use to obtain one or more credentials.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_ACCESS_REQUEST
CSSMERR_DL_INVALID_DB_HANDLE

NAME

CSSM_DL_GetDbAcl for the CSSM API
DL_GetDbAcl for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_GetDbAcl
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_STRING *SelectionTag,
    uint32 *NumberOfAclInfos,
    CSSM_ACL_ENTRY_INFO_PTR *AclInfos)

SPI:
CSSM_RETURN CSSMDLI DL_GetDbAcl
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_STRING *SelectionTag,
    uint32 *NumberOfAclInfos,
    CSSM_ACL_ENTRY_INFO_PTR *AclInfos)
```

DESCRIPTION

This function returns a description of zero or more ACL entries managed by the data storage service provider module and associated with the target database identified by *DLDBHandle.DBHandle*. The optional input *SelectionTag* restricts the returned descriptions to those ACL entries with a matching *EntryTag* value. If a *SelectionTag* value is specified and no matches are found, zero descriptions are returned. If no *SelectionTag* is specified, a description of all ACL entries associated with the target data base are returned by this function.

Each *AclInfo* structure contains:

- Public contents of an ACL entry
- ACL *EntryHandle*, which is a unique value defined and managed by the service provider

The public ACL entry information returned by this function includes:

- The subject type - A CSSM_LIST structure containing one element identifying the type of subject stored in the ACL entry.
- Delegation flag - A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in Authorization
- Authorization array - A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period - A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A CSSM_STRING containing a user-defined value associated with the ACL entry.

PARAMETERS

DLDBHandle (input)
The handle pair that identifies the Data Storage service provider to perform this operation and the target data store whose associated ACL entries are scanned and returned.

SelectionTag (input/optional)
A CSSM_STRING value matching the user-defined tag value associated with one or more ACL entries for the target data base. To retrieve a description of all ACL entries for the

target data base, this parameter must be NULL.

NumberOfAclInfos (output)
The number of entries in the *AclInfos* array. If no ACL entry descriptions are returned, this value is zero.

AclInfos (output)
An array of CSSM_ACL_ENTRY_INFO structures. The unique handle contained in each structure can be used during the current attach session to reference the ACL entry for editing. The structure is allocated by the service provider and must be released by the caller when the structure is no longer needed. If no ACL entry descriptions are returned, this value is NULL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE

SEE ALSO

For the CSSM API: *CSSM_DL_ChangeDbAcl*()

For the DL SPI: *DL_ChangeDbAcl*()

NAME

CSSM_DL_ChangeDbAcl for the CSSM API
DL_ChangeDbAcl for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_ChangeDbAcl
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_EDIT *AclEdit)

SPI:
CSSM_RETURN CSSMDLI DL_ChangeDbAcl
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_EDIT *AclEdit)
```

DESCRIPTION

This function edits the stored ACL associated with the target data base identified by *DLDBHandle.DBHandle*. The ACL is modified according to the edit mode and information provided in *AclEdit*.

The caller must be authorized to modify the target ACL. Caller authentication and authorization to edit the ACL is determined based on the caller-provided *AccessCred*.

The caller must be authorized to add, delete or replace the ACL entries associated with the target data base. When adding or replacing an ACL entry, the service provider must reject the creation of duplicate ACL entries.

When adding a new ACL entry to an ACL, the caller must provide a complete ACL entry prototype. All ACL entry items, except the ACL entry *TypedSubject* must be provided as an immediate value in *AclEdit→NewEntry*. The ACL entry Subject can be provided as an immediate value, from a verifier with a protected data path, from an external authentication or authorization service, or through a callback function specified in *AclEdit→NewEntry→Callback*.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the data storage library module to be used to perform this function, and the open data store whose associated ACL entries are to be updated.

AccessCred (input)
A pointer to the set of one or more credentials used to authenticate and validate the caller's authorization to modify the ACL associated with the target data base. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

AclEdit (input)
A structure containing information that defines the edit operation. Valid operations include adding, replacing and deleting entries in the set of ACL entries managed by the service provider. The *AclEdit* can contain information for a new ACL entry and a unique handle identifying an existing ACL entry. The information controls the edit operation as follows:

Value of AclEdit.EditMode	Use of AclEdit.NewEntry and AclEdit.OldEntryHandle
CSSM_ACL_EDIT_MODE_ADD	Adds a new ACL entry to the set of ACL entries associated with the specified data base. The new ACL entry is created from the prototype ACL entry contained in *NewEntry*. *OldEntryHandle* is ignored for this *EditMode*.
CSSM_ACL_EDIT_MODE_DELETE	Deletes the ACL entry identified by *OldEntryHandle* and associated with the specified data base. *NewEntry* is ignored for this *EditMode*.
CSSM_ACL_EDIT_MODE_REPLACE	Replaces the ACL entry identified by *OldEntryHandle* and associated with the specified data base. The existing ACL is replaced based on the ACL entry prototype contained in *NewEntry*.

When replacing an existing ACL entry, the caller must replace all of the items in an ACL entry. The replacement prototype includes:

- Subject type and value - A CSSM_LIST structure containing a typed Subject. The Subject identifies the entity authorized by this ACL entry.
- Delegation flag - A CSSM_BOOL value indicating whether the subject can delegate the permissions recorded in the authorization array.
- Authorization array - A CSSM_AUTHORIZATIONGROUP structure defining the set of operations for which permission is granted to the Subject.
- Validity period - A CSSM_ACL_VALIDITY_PERIOD structure containing two elements, the start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A CSSM_STRING containing a user-defined value associated with the ACL entry.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE

SEE ALSO

For the CSSM API: *CSSM_DL_GetDbAcl*( )

For the DL SPI: *DL_GetDbAcl*( )

NAME

CSSM_DL_GetDbOwner for the CSSM API
DL_GetDbOwner for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_GetDbOwner
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_ACL_OWNER_PROTOTYPE_PTR Owner)

SPI:
CSSM_RETURN CSSMDLI DL_GetDbOwner
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_ACL_OWNER_PROTOTYPE_PTR Owner)
```

DESCRIPTION

This function returns a CSSM_ACL_OWNER_PROTOTYPE describing the current Owner of the Data Base.

PARAMETER

DLDBHandle (input)
The handle pair that describes the data storage library module to be used to perform this function, and the open data store whose associated Owner is to be retrieved.

Owner (output)
A CSSM_ACL_OWNER_PROTOTYPE describing the current Owner of the Data Base.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE

SEE ALSO

For the CSSM API: *CSSM_DL_ChangeDbOwner*( )

For the DL SPI: *DL_ChangeDbOwner*( )

NAME

CSSM_DL_ChangeDbOwner for the CSSM API
DL_ChangeDbOwner for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_ChangeDbOwner
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_OWNER_PROTOTYPE *NewOwner)

SPI:
CSSM_RETURN CSSMDLI DL_ChangeDbOwner
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const CSSM_ACL_OWNER_PROTOTYPE *NewOwner)
```

DESCRIPTION

This function takes a CSSM_ACL_OWNER_PROTOTYPE defining the new Owner of the Data Base.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the data storage library module to be used to perform this function, and the open data store whose associated Owner is to be updated.

AccessCred (input)
A pointer to the set of one or more credentials used to prove the caller is the current Owner of the Data Base. Required credentials can include zero or more certificates, zero or more caller names, and one or more samples. If certificates and/or caller names are provided as input these must be provided as immediate values in this structure. The samples can be provided as immediate values or can be obtained through a callback function included in the *AccessCred* structure.

NewOwner (Input)
A CSSM_ACL_OWNER_PROTOTYPE defining the new Owner of the Data Base.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_NEW_OWNER

SEE ALSO

For the CSSM API: *CSSM_DL_GetDbOwner*()

For the DL SPI: *DL_GetDbOwner*()

11.9 Data Storage Operations

The man-page definitions for Data Storage operations are presented in this section.

NAME

CSSM_DL_DbOpen for the CSSM API
DL_DbOpen for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DbOpen
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const void *OpenParameters,
    CSSM_DB_HANDLE *DbHandle)

SPI:
CSSM_RETURN CSSMDLI DL_DbOpen
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_ACCESS_CREDENTIALS *AccessCred,
    const void *OpenParameters,
    CSSM_DB_HANDLE *DbHandle)
```

DESCRIPTION

This function opens the data store with the specified logical name under the specified access mode. 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, 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.

AccessCred (input/optional)
A pointer to the set of one or more credentials being presented for authentication by the caller. These credentials are required to obtain access to the specified data store. The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN,

certificate, or template of user-specific data, or the caller can specify a callback function the DL can use to obtain one or more credentials. The required set of credentials to access a particular data store is defined by the *DbInfo* record containing meta-data for the specified data store. If credentials are not required to access the specified data store, then this field can be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

DbHandle (output)
The handle to the opened data store. The value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_DB_LOCKED
CSSMERR_DL_INVALID_ACCESS_REQUEST
CSSMERR_DL_INVALID_DB_LOCATION
CSSMERR_DL_INVALID_DB_NAME
CSSMERR_DL_DATASTORE_DOESNOT_EXIST
CSSMERR_DL_INVALID_PARSING_MODULE
CSSMERR_DL_INVALID_OPEN_PARAMETERS

SEE ALSO

For the CSSM API: *CSSM_DL_DbClose*()

For the DL SPI: *DL_DbClose*()

NAME

CSSM_DL_DbClose for the CSSM API
DL_DbClose for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DbClose
    (CSSM_DL_DB_HANDLE DLDBHandle)

SPI:
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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE

SEE ALSO

For the CSSM API:
CSSM_DL_DbOpen( )

For the DL SPI:
DL_DbOpen( )

NAME

CSSM_DL_DbCreate for the CSSM API
DL_DbCreate for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DbCreate
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    const CSSM_DBINFO *DBInfo,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    const void *OpenParameters,
    CSSM_DB_HANDLE *DbHandle)

SPI:
CSSM_RETURN CSSMDLI DL_DbCreate
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    const CSSM_DBINFO *DBInfo,
    CSSM_DB_ACCESS_TYPE AccessRequest,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    const void *OpenParameters,
    CSSM_DB_HANDLE *DbHandle)
```

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. If any *RecordType* defined in the DBINFO structure does not have an associated parsing module, then the *ModuleSubserviceUid* specified for that record type must be zero.

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 *OpenParameters*. If user authentication credentials are required, they must be provided.

Authorization policy can restrict the set of callers who can create a new resource. In this case, the caller must present a set of access credentials for authorization. Upon successfully authenticating the credentials, the template that verified the presented samples identifies the ACL entry that will be used in the authorization computation. If the caller is authorized, the new resource is created.

The caller must provide an initial ACL entry to be associated with the newly created resource. This entry is used to control future access to the new resource and (since the subject is deemed to be the "Owner") exercise control over its associated ACL. The caller can specify the following items for initializing an ACL entry:

- Subject - A CSSM_LIST structure, containing the type of the subject and a template value that can be used to verify samples that are presented in credentials when resource access is requested.
- Delegation flag - A value indicating whether the Subject can delegate the permissions recorded in the *AuthorizationTag*. (This item only applies to public key subjects).

- Authorization tag - The set of permissions that are granted to the Subject.
- Validity period - The start time and the stop time for which the ACL entry is valid.
- ACL entry tag - A user-defined string value associated with the ACL entry.

The service provider can modify the caller-provided initial ACL entry to conform to any innate resource-access policy that the service provider may be required to enforce. If the initial ACL entry provided by the caller contains values or permissions that are not supported by the service provider, then the service provider can modify the initial ACL appropriately or can fail the request to create the new resource. Service providers list their supported *AuthorizationTag* values in their Module Directory Services primary record.

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 assume a default storage service process location. If the *DbName* does not distinguish the storage service process, 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.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read-write.

CredAndAclEntry (input/optional)
A structure containing one or more credentials authorized for creating a data base and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values or callback functions can be provided for use by the DL to acquire the credentials and/or the ACL entry interactively. If the DL provides public access for creating a data base, then the credentials can be NULL. If the DL defines a default initial ACL entry for the new data base, then the ACL entry prototype can be an empty list.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

DbHandle (output)
The handle to the newly created and open data store. The value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_DATASTORE_ALREADY_EXISTS
CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_ACCESS_REQUEST
CSSMERR_DL_INVALID_DB_LOCATION
CSSMERR_DL_INVALID_DB_NAME
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_OPEN_PARAMETERS
CSSMERR_DL_INVALID_PARSING_MODULE
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_INDEX
CSSMERR_DL_UNSUPPORTED_FIELD_FORMAT
CSSMERR_DL_UNSUPPORTED_INDEX_INFO
CSSMERR_DL_UNSUPPORTED_LOCALITY
CSSMERR_DL_UNSUPPORTED_NUM_ATTRIBUTES
CSSMERR_DL_UNSUPPORTED_NUM_INDEXES
CSSMERR_DL_UNSUPPORTED_NUM_RECORDTYPES
CSSMERR_DL_UNSUPPORTED_RECORDTYPE

SEE ALSO

For the CSSM API:
CSSM_DL_DbOpen( )
CSSM_DL_DbClose( )
CSSM_DL_DbDelete( )

For the DL SPI:
DL_DbOpen( )
DL_DbClose( )
DL_DbDelete( )

NAME

CSSM_DL_DbDelete for the CSSM API
DL_DbDelete for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DbDelete
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    const CSSM_ACCESS_CREDENTIALS *AccessCred)

SPI:
CSSM_RETURN CSSMDLI DL_DbDelete
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS *DbLocation,
    const CSSM_ACCESS_CREDENTIALS *AccessCred)
```

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 assume a default storage service process location. If the *DbName* does not distinguish the storage service process, the service cannot be performed and the operation fails.

AccessCred (input/optional)
A pointer to the set of one or more credentials being presented for authentication by the caller. These credentials are required to obtain access to the specified data store. The credentials structure can contain multiple types of credentials, as required for multi-factor authentication. The credential data can be an immediate value, such as a passphrase, PIN, certificate, or template of user-specific data, or the caller can specify a callback function the DL can use to obtain one or more credentials. The required set of credentials to access a particular data store is defined by the *DbInfo* record containing meta-data for the specified data store. If credentials are not required to access the specified data store, then this field can be NULL.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_DATASTORE_DOESNOT_EXIST
CSSMERR_DL_DATASTORE_IS_OPEN
CSSMERR_DL_INVALID_DB_LOCATION
CSSMERR_DL_INVALID_DB_NAME

SEE ALSO

For the CSSM API:
CSSM_DL_DbCreate( )
CSSM_DL_DbOpen( )
CSSM_DL_DbClose( )

For the DL SPI:
DL_DbCreate( )
DL_DbOpen( )
DL_DbClose( )

NAME

CSSM_DL_CreateRelation for the CSSM API
DL_CreateRelation for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_CreateRelation
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RelationID,
    const char *RelationName,
    uint32 NumberOfAttributes,
    const CSSM_DB_SCHEMA_ATTRIBUTE_INFO *pAttributeInfo,
    uint32 NumberOfIndexes,
    const CSSM_DB_SCHEMA_INDEX_INFO *pIndexInfo)

SPI:
CSSM_RETURN CSSMDLI DL_CreateRelation
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RelationID,
    const char *RelationName,
    uint32 NumberOfAttributes,
    const CSSM_DB_SCHEMA_ATTRIBUTE_INFO *pAttributeInfo,
    uint32 NumberOfIndexes,
    const CSSM_DB_SCHEMA_INDEX_INFO *pIndexInfo)
```

DESCRIPTION

This function creates a new persistent relation of the specified type by inserting it into the specified data store. The *pAttributeInfo* and *pIndexInfo* specify the values contained in the new relation 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 relation record. The database should be opened in administrative mode using the CSSM_DB_ACCESS_PRIVILEGED flag.

RelationID (input)
: Indicates the type of relation record being added to the data store.

RelationName (input)
: Indicates the name of the relation being added to the data store.

NumberOfAttributes (input)
: Indicates the number of attributes specified in *pAttributeInfo*.

pAttributeInfo (input)
: A list of structures containing the meta information (schema) describing the attributes for the relation being added to the specified data store. The list contains at most one entry per attribute in the specified record type.

NumberOfIndexes (input)
: Indicates the number of indexes specified in *pIndexInfo*.

pIndexInfo (input)
: A list of structures containing the meta information (schema) describing the indexes for the

relation being added to the specified data store. The list contains at most one entry per index in the specified record type.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INVALID_ATTRIBUTE_INFO
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_INDEX_INFO
CSSMERR_DL_INVALID_RECORDTYPE

SEE ALSO

For the CSSM API: *CSSM_DL_DestroyRelation*( )

For the DL SPI: *DL_DestroyRelation*( )

NAME

CSSM_DL_DestroyRelation for the CSSM API
DL_DestroyRelation for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DestroyRelation
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RelationID)

SPI:
CSSM_RETURN CSSMDLI DL_DestroyRelation
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RelationID)
```

DESCRIPTION

This function destroys an existing relation of the specified type by removing its entry 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 relation record.

RelationID (input)
Indicates the type of relation record being deleted from the data store.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_RECORDTYPE

SEE ALSO

For the CSSM API: *CSSM_DL_CreateRelation*()

For the DL SPI: *DL_CreateRelation*()

NAME

CSSM_DL_GetDbNames for the CSSM API
DL_GetDbNames for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_GetDbNames
    (CSSM_DL_HANDLE DLHandle,
    CSSM_NAME_LIST_PTR *NameList)

SPI:
CSSM_RETURN CSSMDLI DL_GetDbNames
    (CSSM_DL_HANDLE DLHandle,
    CSSM_NAME_LIST_PTR *NameList)
```

DESCRIPTION

This function returns the list of logical data store names for all data stores that are known by and accessible to the specified DL module. This list also includes the number of data store names in the return list.

The *CSSM_DL_FreeNameList*( ) function must be called to de-allocate memory containing the list.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

NameList (output)
Returns a list of data store names in a CSSM_NAME_LIST_PTR structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_DL_GetDbNameFromHandle( )
CSSM_DL_FreeNameList( )

For the DL SPI:
DL_GetDbNameFromHandle( )
DL_FreeNameList( )

NAME

CSSM_DL_GetDbNameFromHandle for the CSSM API
DL_GetDbNameFromHandle for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_GetDbNameFromHandle
    (CSSM_DL_DB_HANDLE DLDBHandle,
    char **DbName)

SPI:
CSSM_RETURN CSSMDLI DL_GetDbNameFromHandle
    (CSSM_DL_DB_HANDLE DLDBHandle,
    char **DbName)
```

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.

DbName (output)
Returns a zero terminated string which contains a data store name. The memory is allocated by the service provider and must be de-allocated by the application.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE

SEE ALSO

For the CSSM API:
CSSM_DL_GetDbNames( )

For the DL SPI:
DL_GetDbNames( )

NAME

CSSM_DL_FreeNameList for the CSSM API
DL_FreeNameList for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_FreeNameList
    (CSSM_DL_HANDLE DLHandle,
    CSSM_NAME_LIST_PTR NameList)

SPI:
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 *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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

None specific to this call. See the **Error Codes and Error Values** section earlier in this Chapter.

SEE ALSO

For the CSSM API:
CSSM_DL_GetDbNames( )

For the DL SPI:
DL_GetDbNames( )

11.10 Data Record Operations

The man-page definitions for Data Record operations are presented in this section.

NAME

CSSM_DL_DataInsert for the CSSM API
DL_DataInsert for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataInsert
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA *Attributes,
    const CSSM_DATA *Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)

SPI:
CSSM_RETURN CSSMDLI DL_DataInsert
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA *Attributes,
    const CSSM_DATA *Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)
```

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 *Attributes* parameter also specifies a record type. This type must be the same as the type specified by the *RecordType* input parameter. 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.

If a primary key (concatination of all unique indexes in the relation) exists, the error CSSMERR_DL_INVALID_UNIQUE_INDEX_DATA is returned. The client should call *CSSM_DL_DataGetFirst*() followed by *CSSM_DL_DataModify*() to change an existing 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 specified *AttributeFormat* for each attribute must match that of the database schema, otherwise the error CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT is returned. If an attribute is of type CSSM_DB_ATTRIBUTE_FORMAT_STRING and the value specified for that string includes a null-terminator, then the length count in the CSSM_DATA structure containing the input string should include the terminating character. (If null-terminators are used, they should be used consistently when storing, searching, and retrieving the string value, otherwise selection predicates will not locate expected matches.) For those attributes that are not assigned values by the caller, the DL module may assume the values to be the empty set, or

assume default values, or 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.

UniqueId (output)
A pointer to a CSSM_DB_UNIQUE_RECORD_PTR containing a unique identifier associated with the new record. This unique identifier structure can be used in future references to this record during the current open data base session. The pointer will be set to NULL if the function fails. The *CSSM_DL_FreeUniqueRecord*( ) function must be used to deallocate this structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_PARSING_MODULE
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_UID
CSSMERR_DL_INVALID_UNIQUE_INDEX_DATA
CSSMERR_DL_INVALID_VALUE
CSSMERR_DL_MISSING_VALUE

SEE ALSO

For the CSSM API:
CSSM_DL_DataDelete( )

For the DL SPI:
DL_DataDelete( )

NAME

CSSM_DL_DataDelete for the CSSM API
DL_DataDelete for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataDelete
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD *UniqueRecordIdentifier)

SPI:
CSSM_RETURN CSSMDLI DL_DataDelete
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD *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, except as an argument to *DL_FreeUniqueRecord*( ), which must still be called.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_RECORD_UID
CSSMERR_DL_RECORD_NOT_FOUND

SEE ALSO

For the CSSM API:
CSSM_DL_DataInsert( )

For the DL SPI:
DL_DataInsert( )

NAME

CSSM_DL_DataModify for the CSSM API
DL_DataModify for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataModify
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RecordType,
    CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA *AttributesToBeModified,
    const CSSM_DATA *DataToBeModified,
    CSSM_DB_MODIFY_MODE ModifyMode)

SPI:
CSSM_RETURN CSSMDLI DL_DataModify
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_RECORDTYPE RecordType,
    CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA *AttributesToBeModified,
    const CSSM_DATA *DataToBeModified,
    CSSM_DB_MODIFY_MODE ModifyMode)
```

DESCRIPTION

This function modifies the persistent data record identified by the *UniqueRecordIdentifier*. The modifications are specified by the *Attributes* and *Data* parameters. The *ModifyMode* indicates how the attributes are to be updated. The *ModifyMode* has no affect on updating the data blob contained in the record. If the data blob is the only record attribute being updated by this function call, then the modification mode must be 0. The current modification modes behave as follows:

ModifyMode Value	Function Behavior
CSSM_DB_MODIFY_ATTRIBUTE_NONE	No Attributes are being updated.
CSSM_DB_MODIFY_ATTRIBUTE_ADD	The specified values are added to the set of current values for each attribute. If 0 values are specified then the error CSSMERR_DL_INVALID_MODIFY_MODE is returned. If a DL does not support multiple values per attribute, the error CSSMERR_DL_MULTIPLE_VALUES_UNSUPPORTED is returned.
CSSM_DB_MODIFY_ATTRIBUTE_DELETE	The specified values are removed from the set of current values for each attribute. If 0 values are specified then all values are deleted or the attributes value is replaced with the default for this attribute. If a DL does not support multiple values per attribute, the error CSSMERR_DL_MULTIPLE_VALUES_UNSUPPORTED is returned.
CSSM_DB_MODIFY_ATTRIBUTE_REPLACE	The values for each attribute are replaced with the specified set of values for each attribute. If no values are specified then all values are deleted or the attributes value is replaced with the default for

	this attribute. If a DL does not support multiple values per attribute, the error CSSMERR_DL_MULTIPLE_VALUES_UNSUPPORTED is returned when more than 1 value is specified.

If the attribute lists specifies an attribute that is not defined in the database's meta-information, an error condition is returned. For each attribute-value pair, the value replaces the corresponding attribute value in the record. If a data value is specified, the record's data value is replaced with the specified value. A record's data value or attribute values can be set to NULL or zero to represent deletion or the lack of a known value.

If the record referenced by *UniqueRecordIdentifier* has been modified since the last time it was updated, the error CSSMERR_DL_STALE_UNIQUE_RECORD is returned and no modification takes place.

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/output)
A pointer to a CSSM_DB_UNIQUE_RECORD containing a unique identifier associated with the record to modify. If the modification succeeds, the *UniqueRecordIdentifier* points to a CSSM_DB_UNIQUE_RECORD containing a unique identifier associated with the updated record. If the modification fails, the *UniqueRecordIdentifier* is not modified.

AttributesToBeModified (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 specified *AttributeFormat* for each attribute must match that of the database schema, otherwise the error CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT is returned. If an attribute is of type CSSM_DB_ATTRIBUTE_FORMAT_STRING and the value specified for that string includes a null-terminator, then the length count in the CSSM_DATA structure containing the input string should include the terminating character. (If null-terminators are used, they should be used consistently when storing, searching, and retrieving the string value, otherwise selection predicates will not locate expected matches.) Each attribute specified is modified according to the value of *ModifyMode* (see table in the **DESCRIPTION** section of this definition). Those attributes that are not specified as part of this parameter remain unchanged. If the *AttributesToBeModified* 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.

ModifyMode (input)
A CSSM_DB_MODIFY_MODE value indicating the type of modification to be performed on the record attributes identified by *AttributesToBeModified*. If no attributes are specified, then this value must be CSSM_DB_MODIFY_ATTRIBUTE_NONE.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_MODIFY_MODE
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_UID
CSSMERR_DL_INVALID_UNIQUE_INDEX_DATA
CSSMERR_DL_INVALID_VALUE
CSSMERR_DL_MULTIPLE_VALUES_UNSUPPORTED
CSSMERR_DL_STALE_UNIQUE_RECORD

SEE ALSO

For the CSSM API:
CSSM_DL_DataInsert( )
CSSM_DL_DataDelete( )

For the DL SPI:
DL_DataInsert( )
DL_DataDelete( )

NAME

CSSM_DL_DataGetFirst for the CSSM API
DL_DataGetFirst for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataGetFirst
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_QUERY *Query,
    CSSM_HANDLE_PTR ResultsHandle,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)

SPI:
CSSM_RETURN CSSMDLI DL_DataGetFirst
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_QUERY *Query,
    CSSM_HANDLE_PTR ResultsHandle,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)
```

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. If the Query specifies an attribute that is not defined in the database's meta-information, an error condition is returned. 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. The output buffers for the retrieved record are allocated by this function using the memory management functions provided during the module attach operation. This function also returns a results handle to be used when retrieving subsequent records satisfying the query.

Additional matching records are iteratively retrieved using the function *CSSM_DL_DataGetNext*(). The data storage module supports one of two retrieval models:

- Transactional - all query results are determined at initial query evaluation. Results do not change during an incremental retrieval process.
- File System Scan - query results are selected during the incremental retrieval process. Records matching the query may be added to or deleted from the underlying data store during the iterative retrieval. The caller may receive the new matching records and not received the deleted records.

The caller can determine which retrieval model is supported by examining the encapsulated product description for this data storage module.

If the query selection criteria also specifies time for space limits for executing the query, those limits also apply ro 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. Once a user has finished using a certain query, it must call *CSSM_DataAbortQuery*() for releasing resources that CSSM uses. If all records satisfying the query have been retrieved, then query is automatically terminated.

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 field of this Query structure. If a search attribute is of type CSSM_DB_ATTRIBUTE_FORMAT_STRING and the search value specified for that string includes a null-terminator, then the length count for that string should include the terminating character. (If null-terminators are used they should be used consistently, storing the terminator as part of the string in the data store, otherwise selection predicates will not locate expected matches.) The Query structure attributes also identify the particular attributes to be searched by this query. If no query is specified, the DL module can return the first record in the data store, performing sequential retrieval, or return an error. If no selection predicates are specified, the DL module can return the first record in the data store, performing sequential retrieval, or return an error (CSSM_DL_UNSUPPORTED_NUM_SELECTION_PREDS). When selection predicates are specified, the *NumberOfValues* of the *Attribute* of each selection predicate must be 1. If any selection predicate does not satisfy this requirement, the error CSSMERR_DL_INVALID_QUERY is returned.

ResultsHandle (output)
This handle should be used to retrieve subsequent records that satisfied this query.

Attributes (optional-input/output)
If the *Attributes* structure pointer is NULL, no values are returned.

Otherwise, the *DataRecordType*, *NumberOfAttributes* and *AttributeData* fields are read. *AttributeData* must be an array of *NumberOfAttributes* CSSM_DB_RECORD_ATTRIBUTE elements. Only the *Info* field of each element is used on input. The *AttributeFormat* field of the *Info* field is ignored on input.

On output, a CSSM_DB_RECORD_ATTRIBUTE structure containing a list of all or the requested attribute values (subset) from the retrieved record. The *SemanticInformation* field is set. For each CSSM_DB_ATTRIBUTE_DATA contained in the *AttributeData* array, the *NumberOfValues* field is set to reflect the size of the *Value* array which is allocated by the DL using the application specified allocators. Each CSSM_DATA in the *Value* array will have it's *Data* field as a pointer to data allocated using the application specified allocators containing the attributes value, and have it's *Length* set to the length of the value.

All values for an attribute are returned (this could be 0). All fields in the *Info* field of the CSSM_DB_ATTRIBUTE_DATA are left unchanged except for the *AttributeFormat* field, which is set to reflect the schema.

Data (optional-input/output)
Data values contained in the referenced memory are ignored during processing and are overwritten with the retrieved opaque object. On output, a CSSM_DATA structure containing the opaque object stored in the retrieved record.

UniqueId (output)
If successful and (at least) a record satisfying the query has been found, then this parameter returns a pointer to a CSSM_UNIQUE_RECORD_PTR 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 there are no records satisfying the query, then this pointer is NULL and *CSSM_DL_DataGetFirst*( ) must return CSSM_DL_ENDOFDATA; in this case a normal termination condition has occurred. The *CSSM_DL_FreeUniqueRecord*( ) must be used to de-allocate this structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_ENDOFDATA
CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_PARSING_MODULE
CSSMERR_DL_INVALID_QUERY
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_UID
CSSMERR_DL_UNSUPPORTED_FIELD_FORMAT
CSSMERR_DL_UNSUPPORTED_NUM_SELECTION_PREDS
CSSMERR_DL_UNSUPPORTED_OPERATOR
CSSMERR_DL_UNSUPPORTED_QUERY
CSSMERR_DL_UNSUPPORTED_QUERY_LIMITS

SEE ALSO

For the CSSM API:
CSSM_DL_DataGetNext( )
CSSM_DL_DataAbortQuery( )

For the DL SPI:
DL_DataGetNext( )
DL_DataAbortQuery( )

NAME

CSSM_DL_DataGetNext for the CSSM API
DL_DataGetNext for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataGetNext
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)

SPI:
CSSM_RETURN CSSMDLI DL_DataGetNext
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data,
    CSSM_DB_UNIQUE_RECORD_PTR *UniqueId)
```

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 Attributes parameter can specify a subset of the attributes to be returned. If *Attributes* specifies an attribute that is not defined in the database's meta-information, an error condition is returned. The record values are returned in the *Attributes* and *Data* parameters. The output buffers for the retrieved record are allocated by this function using the memory management functions provided during the module attach operation. The function also returns a unique record identifier for the return record.

The data storage module supports one of two retrieval models: transactional or file system scan. The transactional model freezes the set of records to be retrieved at query initiation. The file system scan model selects from a potentially changing set of records during the retrieval process. The *EndOfDataStore*() indicates when all matching records have been retrieved. The caller can determine which retrieval model is supported by examining the encapsulated product description for this data storage module. Once a user has finished using a certain query, it must call *CSSM_DataAbortQuery*() for releasing resources that CSSM uses. If all records satisfying the query have been retrieved, then query is automatically terminated.

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 handle identifying a set of records retrieved by a query executed by the *CSSM_DL_DataGetFirst*() function.

Attributes (optional-input/output)
: If the *Attributes* structure pointer is NULL, no values are returned.

Otherwise, the *DataRecordType*, *NumberOfAttributes* and *AttributeData* fields are read. *AttributeData* must be an array of *NumberOfAttributes* CSSM_DB_RECORD_ATTRIBUTE

elements. Only the *Info* field of each element is used on input. The *AttributeFormat* field of the *Info* field is ignored on input.

On output, a CSSM_DB_RECORD_ATTRIBUTE structure containing a list of all or the requested attribute values (subset) from the retrieved record. The *SemanticInformation* field is set. For each CSSM_DB_ATTRIBUTE_DATA contained in the *AttributeData* array, the *NumberOfValues* field is set to reflect the size of the *Value* array which is allocated by the DL using the application specified allocators. Each CSSM_DATA in the *Value* array will have it's *Data* field as a pointer to data allocated using the application specified allocators containing the attributes value, and have it's *Length* set to the length of the value.

All values for an attribute are returned (this could be 0). All fields in the *Info* field of the CSSM_DB_ATTRIBUTE_DATA are left unchanged except for the *AttributeFormat* field, which is set to reflect the schema.

Data (optional-input/output)
Data values contained in the referenced memory are ignored during processing and are overwritten with the retrieved opaque object. On output, a CSSM_DATA structure containing the opaque object stored in the retrieved record. If the pointer is data structure pointer is NULL, the opaque object is not returned.

UniqueId (output)
If successful and (at least) a record satisfying the query has been found, then this parameter returns a pointer to a CSSM_UNIQUE_RECORD_PTR 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 there are no more records satisfying the query, then this pointer is NULL and *CSSM_DL_DataGetNext*( ) must return CSSM_DL_ENDOFDATA; in this case a normal termination condition has occurred. The *CSSM_DL_FreeUniqueRecord*( ) must be used to de-allocate this structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_ENDOFDATA
CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_UID
CSSMERR_DL_INVALID_RESULTS_HANDLE

SEE ALSO

For the CSSM API:
CSSM_DL_DataGetFirst( )
CSSM_DL_DataAbortQuery( )

For the DL SPI:
DL_DataGetFirst( )
DL_DataAbortQuery( )

NAME

CSSM_DL_DataAbortQuery for the CSSM API
DL_DataAbortQuery for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_DataAbortQuery
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle)

SPI:
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, and release any locks on the resource. The user/application must call *CSSM_DL_DataAbortQuery*( ) at the termination.

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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_RESULTS_HANDLE

SEE ALSO

For the CSSM API:
CSSM_DL_DataGetFirst( )
CSSM_DL_DataGetNext( )

For the DL SPI:
DL_DataGetFirst( )
dL_DataGetNext( )

NAME

CSSM_DL_DataGetFromUniqueRecordId for the CSSM API
DL_DataGetFromUniqueRecordId for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_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)

SPI:
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 *Attributes* parameter can specify a subset of the attributes to be returned. If *Attributes* specifies an attribute that is not defined in the database's meta-information, an error condition is returned. The output buffers for the retrieved record are allocated by this function using the memory management functions provided during the module attach operation. The DL module can use an indexing structure identified in the *UniqueRecordId* to enhance the performance of the retrieval operation.

The DL should assume that the value of CSSM_QUERY_FLAGS is 0 when performing this operation. In particular this means that if the data of a key record is being retrieved, the DL will return a CSSM_KEY structure with a key reference.

If the record referenced by *UniqueRecordIdentifier* has been modified since the last time it was retrieved, the error (warning) CSSMERR_DL_RECORD_MODIFIED is returned but the requested attributes and data of the new record is returned. The caller should be advised that other attributes (or the data) might have changed that were not fetched from the DL with this call.

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 (optional-input/output)
If the *Attributes* structure pointer is NULL, no values are returned.

Otherwise, the *DataRecordType*, *NumberOfAttributes* and *AttributeData* fields are read. *AttributeData* must be an array of *NumberOfAttributes* CSSM_DB_RECORD_ATTRIBUTE elements. Only the *Info* field of each element is used on input. The *AttributeFormat* field of the *Info* field is ignored on input.

On output, a CSSM_DB_RECORD_ATTRIBUTE structure containing a list of all or the requested attribute values (subset) from the retrieved record. The *SemanticInformation* field is set. For each CSSM_DB_ATTRIBUTE_DATA contained in the *AttributeData* array, the *NumberOfValues* field is set to reflect the size of the *Value* array which is allocated by the DL using the application specified allocators. Each CSSM_DATA in the *Value* array will have it's *Data* field as a pointer to data allocated using the application specified allocators containing the attributes value, and have it's *Length* set to the length of the value.

All values for an attribute are returned (this could be 0). All fields in the *Info* field of the CSSM_DB_ATTRIBUTE_DATA are left unchanged except for the *AttributeFormat* field, which is set to reflect the schema.

Data (optional-input/output)

Data values contained in the referenced memory are ignored during processing and are overwritten with the retrieved opaque object. On output, a CSSM_DATA structure containing the opaque object stored in the retrieved record. If the pointer is data structure pointer is NULL, the opaque object is not returned.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_RECORD_UID

SEE ALSO

For the CSSM API:
CSSM_DL_DataInsert()
CSSM_DL_DataGetFirst()
CSSM_DL_DataGetNext()

For the DL SPI:
CSSM_DL_DataInsert()
CSSM_DL_DataGetFirst()
CSSM_DL_DataGetNext()

NAME

CSSM_DL_FreeUniqueRecord for the CSSM API
DL_FreeUniqueRecord for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_FreeUniqueRecord
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)

SPI:
CSSM_RETURN CSSMDLI 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

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_RECORD_UID

SEE ALSO

For the CSSM API:
CSSM_DL_DataInsert()
CSSM_DL_DataGetFirst()
CSSM_DL_DataGetNext()

For the DL SPI:
DL_DataInsert()
DL_DataGetFirst()
DL_DataGetNext()

11.11 Extensibility Operations

The man-page defining the *CSSM_DL_PassThrough*() extensibility function is presented in this section.

NAME

CSSM_DL_PassThrough for the CSSM API
DL_PassThrough for the DL SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_DL_PassThrough
    (CSSM_DL_DB_HANDLE DLDBHandle,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)

SPI:
CSSM_RETURN CSSMDLI DL_PassThrough
    (CSSM_DL_DB_HANDLE DLDBHandle,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)
```

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.

OutputParams (output)
A pointer to a module, implementation-specific structure containing the output data. The service provider will allocate the memory for this structure. The application should free the memory for the structure.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the **Error Codes and Error Values** section earlier in this Chapter.

CSSMERR_DL_INVALID_DB_HANDLE
CSSMERR_DL_INVALID_PASSTHROUGH_ID

Technical Standard

Part 8:

Module Directory Service (MDS)

The Open Group

Chapter 12

Introduction

The Module Directory Services (MDS) provide facilities to describe and locate executable objects and their associated signed manifest integrity credentials.

MDS is a database and access methods used primarily to support secure loading and use of software modules. It is a system-wide service available to all processes. MDS makes special accommodation for CDSA-defined modules, providing access to registration and capabilities information.

12.1 Common Data Security Architecture

MDS provided a locator service that is used extensively 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 18-1 shows the three basic layers of the CDSA:

- System Security Services
- The Common Security Services Manager (CSSM)
- Security Service Provider Modules

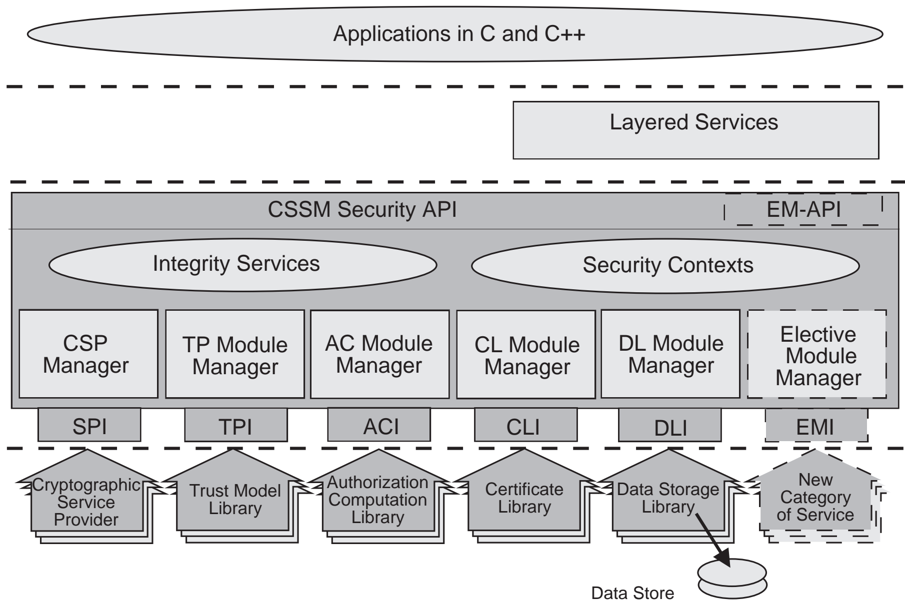


Figure 12-1 Common Data Security Architecture for all Platforms

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 security service modules.

CSSM:

- Defines the application programming interface for accessing security services.
- Defines the service provider 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. Security service modules perform the requested services. 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 security service modules that perform cryptographic operations, manipulate certificates, manage application-domain-specific trust policies, and perform new, elective categories of security services. Independent software and hardware vendors can provide security service modules as competitive products. Applications use CSSM module managers to direct their requests to modules from specific vendors or to any module that performs the required services. A single module can provide one or more categories of service. Modules implementing more than one category of service are called multi-service modules.

CSSM core services support:

- Module management
  The module management functions are used by applications and by modules to support dynamic module selection and module load.
- Security context management
  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 an intermediate state that must be managed.
- System integrity services.
  CSSM, service modules, elective module managers, and optionally applications verify the identity and integrity of dynamic components as they are added to the runtime environment.

CDSA components use MDS to locate executables for CDSA components and the integrity credentials associated with those components. CDSA components and credentials can be stored anywhere on a system (local or remote). MDS allows components to move as required by general system management, while retaining secured use of those components.

12.2 MDS in CDSA

Module Directory Services (MDS) is a platform-independent registry service designed to support secure loading and secure use of software modules. MDS is a system-wide service available to all processes. CDSA is a seminal user of MDS. MDS defines a basic Object Directory schema to name and locate software components and the signed manifest credentials associated with those software components. Each software component in the Object Directory is uniquely named by a GUID (Globally Unique ID). CDSA defines an additional set of schemas to store CDSA-specific security attributes of all CDSA components. CDSA components use the MDS-managed data to:

- Discover other available CDSA components
- Learn about the capabilities and properties of other CDSA components
- Locate the executables for CDSA components
- Locate the signed manifest credentials associated with a CDSA software component.

New schemas can be defined to store the properties and capabilities of Elective CDSA Modules as they are defined over time. CDSA applications can also define MDS schemas and use MDS services. CDSA components use MDS managed data to support CDSA's software authentication and integrity checking procedure, known as bilateral authentication.

MDS supports the CDSA Data Storage Library APIs to query the MDS database. These APIs are familiar to CDSA application developers. The database creation and schema creation interfaces are limited to authorized use by database administrators.

The CDSA-specific schema and data formats are well defined in this specification. Interoperability is achieved through these CDSA-specific schema and data formats. Using this schema and MDS query services, CDSA service providers and applications can interact while maintaining a high degree of autonomy. MDS enforces unique values for primary database keys. MDS can implement indexes over primary keys or dependent attributes to improve query performance.

While MDS is a general service available to all applications, the MDS-defined Object Directory schema and the MDS interfaces are defined here to support CDSA as its first user and for completeness of the CDSA specifications.

The general MDS architecture and its use by CDSA is shown in the following figure.

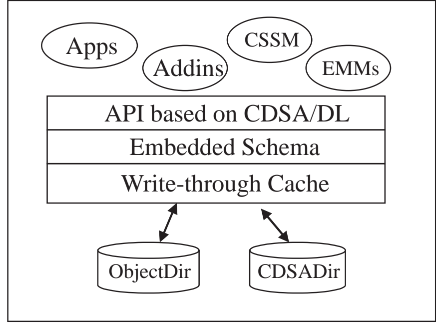


Figure 12-2 MDS Architecture

12.3 MDS Installation and Access

MDS is a single, system-wide service. When CDSA is installed on a system, the CDSA installation procedure must ensure that MDS is available on the system[1]. The CDSA schema is generated at MDS installation time. Updates to the schema are made by administrative tools, such as the installation program

MDS may be accessed through either static or dynamically loaded libraries. MDS interfaces are publicly accessible. MDS is a signed software service. Users of MDS can check the integrity of MDS prior to using MDS services. This ensures the caller that the MDS access library has not been previously compromised. This concern, however, is avoided if the library is statically bound to the caller and the caller has successfully performed an integrity self-check. MDS signed manifest credentials are stored in a well-known location outside of MDS to allow bootstrapping of the integrity services model. Once MDS is available to a process, MDS is the primary source of signed manifest credentials for module self-check and cross-check operations. The MDS Object Directory and the CDSA-defined Directory can be queried by all applications on the platform.

12.4 Using MDS in Integrity Verification Protocols

Software modules can verify the credentials and integrity of any software module registered in the MDS database. The module performing the cross-check must obtain some information about the module being checked, namely, the target module's credentials and installation location.

MDS is used to support the three-step integrity cross-check procedure shown in Figure 18-3. Module (A) needs to verify the authenticity and integrity of module (B). To accomplish this task, Module A must locate the on-disk executable image of module B, compute a cryptographic hash of B, and compare the result with a correct, known hash value. If modules A and B are manufactured separately, module A will not have apriori knowledge module B's correct hash value. The MDS service associates a module's GUID with the module's signed manifest credentials, and the module's installation location. The GUID is the unique database key for retrieving an Object Directory entry.

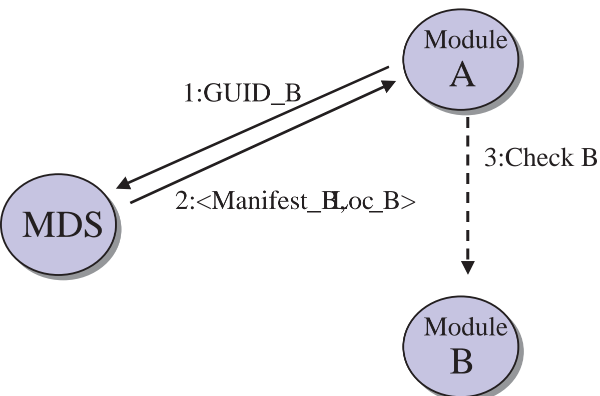


Figure 12-3 Software Module Cross-Check

1. The CDSA installation procedure may be required to install MDS also.

Module A uses MDS to support the cross-check procedure as follows:

1. Module A queries MDS for module B signed manifest credentials based on the GUID for module B. Module A can learn the GUID for module B at manufacturing time or from some runtime source. Typically the software development kit for module B includes the GUID for module B.
2. MDS retrieves the signed manifest and module B install path based on GUID B. The retrieved database entry was created by the installation program for module B.
3. Using the installation path information, module A locates the executable for module B and computes a cryptographic hash of the executable. The hash value for module B is included in the signed manifest for module B. Module A verifies the signature on the manifest and then compares the computed hash value with the hash value contained in the signed manifest. If the values are equal, then module A can trust module B. Trust means that module A can load and transfer execution control to module B.

MDS managed data can also be used to support the CDSA bilateral authentication protocol and authentication over arbitrary CDSA call graphs. Examples using MDS are provided in later sections that define these protocols.

12.5 Multi-User Access Model

MDS data is shared among all processes using the MDS service. MDS enforces data consistency by serializing access to MDS relations. Write operations wait for all read operations to complete before tables are updated. Relations may be read simultaneously by multiple threads and processes.

Users requiring write access to MDS schema or MDS databases must be appropriately authorized. Authorization is based on access privileges granted to software modules through their signed manifest credentials. Any software module (with or without credentials) is granted read-only access to MDS schema and MDS databases.

12.6 API Overview

The MDS API leverages CDSA DL interfaces but uses MDS specific interfaces for initialization and MDS installation. All users of MDS must initialize/terminate using:

- *MDS_Initialize*()
- *MDS_Terminate*()

Installation and configuration applications can use the following functions to define and modify MDS schema:

- *MDS_Install*()
- *MDS_Uninstall*()

Upon successful initialization MDS returns a table of function pointers for MDS services. The function interfaces are a consistent subset of those defined by the CDSA Data Storage Library Interface (DLI).

The DL interfaces supported by MDS are:

- *DbOpen*()

- *DbClose*( )
- *GetDbNames*( )
- *GetDbNameFromHandle*( )
- *FreeNameList*( )
- *DataInsert*( )
- *DataDelete*( )
- *DataModify*( )
- *DataGetFirst*( )
- *DataGetNext*( )
- *DataAbortQuery*( )
- *DataGetFromUniqueRecordId*( )
- *FreeUniqueRecord*( )
- *CreateRelation*( )
- *DestroyRelation*( )

The MDS Object Directory and CDSA Directories are defined in this specification. Creation of these databases is encapsulated in the *MDS_Install* operation. These schemas are well defined and do not require parsing functions to interpret contents. MDS controls access to MDS databases based on authentication credentials presented in the *DbOpen* operation.

Chapter 13

MDS Schema Definition

Module Directory Services standardize a set of schemas and programming interfaces that enable any software object to locate information about any other software object. A set of CDSA-specific schemas are also defined to support CDSA components discover the properties of other CDSA components.

13.1 Object Directory Database and the Object Relation

The root MDS Object Directory is a single relation in an MDS database named *MDS Object Directory*. The database location is registered at MDS installation time with the host platform registry services (for example, Windows registry or UNIX configuration file). The database contains a single relation identified by a relation ID.

The schema for the Object Directory relation is defined in the following table:

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying a collection of software objects (module)
	Manifest	BLOB	Signed-manifest describing the module
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	Path	STRING	Module search path in platform-specific format.
	ProductVersion	STRING	Product version string (in dotted high/low format - e.g. 2.0)

* Indicates the primary database key.

The Object relation contains one record for each instance of a CSSM. Each instance of a CSSM must create entries in the Object relation when CSSM is installed. Any new CDSA applications that have a manifest and need CSSM to do bilateral authentication must insert entries in the Object relation. The primary purpose of the object relation is to support bi-lateral authentication between applications and CSSM. Any other CDSA components added to the platform over time (such as service providers and Elective Module Managers) are not required to add a record to the Object Relation. These CDSA components must add records to other CDSA-specific relations defined later in this specification. The additional information stored in the CDSA-specific relations is required by other CDSA components for successful cross-check and use of CDSA service provider, EMMs, and applications. These records are added to CDSA-specific relations as part of the installation process for that CDSA component.

Typically each CDSA module has exactly one record in the Object Relation. If a third party redistributes a module, and the redistributed manifest signature differs but the object itself is unchanged, the signatures of the redistributed objects are appended to the manifest structure and the augmented manifest is inserted into the existing MDS record by the installation program for the module. The *ModuleID* remains the same and there is a single record for this module. If an object module changes such that the hash of that module changes, then a new *ModuleID* must be generated and a new additional record containing the new *ModuleID* and the new signed manifest must be added to the Object Directory relation by the module's installation program.

The Object Directory contains also relations defining its schema, namely MDS_SCHEMA_RELATIONS, MDS_SCHEMA_ATTRIBUTES, MDS_SCHEMA_INDEXES, which have the same form as those described at Section 19.19 on page 617. The schema relations can be queried by users and applications, but cannot be modified by users or applications.

13.2 CDSA Directory Database

MDS supports a CDSA-specific set of schemas in a database called *MDS CDSA Directory*. The database location is registered at MDS installation time with the platform specific registry service (for example, Windows registry, UNIX config file, etc.) The CDSA directory database file contains relations that are identified by relation Ids. The implemented CDSA relation Ids are registered with the platform-specific registry service.

The information stored in the CDSAA Directory database is provided to:

- Support intra-CDSA module authentication
- Provide access to public information about module properties and capabilities
- Support the addition of new types of CDSA modules.

The CDSA schemas focus on CDSA security service provider modules and CDSA Elective Module Managers. Each module type has one or more corresponding relations containing module-specific information. The CDSA-specific schemas are defined in the following sections.

13.3 CSSM Relation

This relation defines attributes of an instance of CSSM.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying a CSSM module.
	CDSAVersion	STRING	CSSM Version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	CSSM Vendor name in ASCII text.
	Desc	STRING	CSSM description in ASCII text.
	NativeServices**	uint32	Set of service managers which are native to CSSM. This information is a CSSM_SERVICE_MASK.

* Indicates the primary database key.

** In the *NativeServices*, CSSM can describe all services that are implemented natively. If a module service type is set in the Native services, then CSSM implements the service type. If not set, the module may be implemented as an EMM.

13.4 KRMM Relation

This relation defines attributes of an instance of KRMM.

	Field Name	Field Data Type	Comment
*	CSSMGuid	STRING	GUID (in string format) uniquely identifying a CSSM module.
*	PolicyType	unit32	Flag identifying the KR policy type.
	PolicyName	STRING	Human readable name of the module containing the policy information.
	PolicyPath	STRING	Module search path in platform-specific format.
	PolicyInfo	BLOB	Additional policy information, used by CSSM.
	PolicyManifest	BLOB	Signed maifest describing the module

13.5 EMM Relation

This relation defines attributes of a CDSA Elective Module Manager. This relation contains information that allows the CSSM to perform cross-check operations before loading the EMM. There is a single entry for each EMM installed on the system. The *CDSAVersion* is used by CSSM to identify EMMs that interoperate with this version of CDSA. The *EMMSpecVersion* is used by CSSM to identify EMMs that interoperate with EMM service providers.

	Field Name	**Field Data Type**	**Comment**
*	ModuleID	STRING	GUID (in string format) identifying EMM modules
	Manifest	BLOB	Signed-manifest describing the EMM module
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	Path	STRING	Library search path. Contains locations where EMM modules and EMM service provider modules are installed. Path is in platform specific format (Windows, UNIX, Mac).
	CDSAVersion	STRING	Highest compatible CDSA Version (in dotted high/low format - e.g. 2.0).
	EMMSpecVersion	STRING	Highest compatible EMM spec Version (in dotted high/low format - e.g. 2.0).
	Desc	STRING	Module description
	PolicyStmt	BLOB	Any policy statement defined and managed by this EMM.
	EmmVersion	STRING	EMM manufacturer version string (in dotted high/low format - e.g. 2.0).
	EmmVendor	STRING	EMM manufacturer/vendor name in ASCII text.
	EmmType	UINT32	Module service type supported by the EMM.

* Indicates the primary database key.

As new EMMs are defined, additional information can be required. New relations can be created in MDS to store this information. To add update existing relations or add new relations, the CDSA Directory database is closed for general use and re-opened for administrative use. A unique CDSA RelationType value must be defined and associated with each new MDS relation.

13.6 Primary EMM Service Provider Relation

This relation contains credentials that are introduced to the system by dynamic EMM service providers.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	identifying the service provider module
*	SSID	UINT32	4 byte Subservice ID
*	ServiceType	UINT32	Flag identifying the CSSM_SERVICE_TYPE of the service provider module. The integer corresponds to symbols of type CSSM_SERVICE_TYPE.
	Manifest	BLOB	Reserved foir future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProcuctVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	SampleTypes	MULTIUINT32	An array of 4-byte integers representing the sample types accepted by the service provider. The integers correspond to symbols of type CSSM_SAMPLE_TYPE
	Acl SubjectTypes	MULTIUINT32	An array of 4-byte integers representing the ACL subject types accepted by the service provider. The integers correspond to symbols of type CSSM_ACL_SUBJECT_TYPE
	AuthTags	MULTIUINT32	An array of 4-byte integers representing the Authorization tag values defined by the service provider. The integers correspond to symbols of type CSSM_ACL_AUTHORIZATION_TAG
	EMMSpecVersion	STRING	Highest compatible EMM spec Version (in dotted high/low format - e.g. 2.0)

* Indicates the primary database key.

13.7 Common Relation

The *Common Relation* contains information common to all CDSA service provider modules. Information in this table uses the module GUID as a unique key. The Manifest element describes the library that performs event notification and exports CDSA interfaces. The *ModuleName* element is the name of the library file. Path is the search path used to find installed modules. Path also describes manifests that may be introduced to the system by dynamic service providers. See also the *Primary Relation* schema defines in this specification.

This relation will be updated during service provider installation and deinstallation.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
	Manifest	BLOB	Signed-manifest describing the module
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	Path	STRING	Module installation path in platform specific format (Windows, UNIX, Mac).
	CDSAVersion	STRING	Highest compatible CDSA Version (in dotted high/low format - e.g. 2.0).
	Desc	STRING	Module description
	DynamicFlag	UINT32	Module supports dynamic subservices
	MultiThreadFlag	UINT32	Module requires CSSM to serialize access. This flag will be deprecated in future, as it cannot be enforced in an environment with multiple CSSMs in the same process space (two independent CSSMs using the same *addin* will be unaware of the other's threadsafe mutex)
	ServiceMask	UINT32	Service Mask of all supported service types. The integer corresponds to symbols of type CSSM_SERVICE_MASK.

* Indicates the primary database key.

13.8 CSP Primary Relation

The CSP Primary Relation describes attributes of a cryptographic service provider. The *ModuleID* and *sub-service ID* (SSID) uniquely identify the CSP. The information in this relation can change each time a CSP is inserted or removed from the running system.

The USEE tag values in this table must be verified against the actual values in the module manifest before granting security-critical privileges.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) for service provider module
*	SSID	UINT32	4 byte Subservice ID
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProductVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	CspType	UINT32	Implementation type, e.g. software/hardware (4 bytes). The integers correspond to symbols of type CSSM_SERVICE_MASK.
	CspFlags	UINT32	Flags (4 bytes). The integers correspond to symbols of type CSSM_CSP_FLAGS.
	CspCustomFlags	UINT32	More Flags (4 bytes)
	UseeTags	MULTIUINT32	Array of 4-byte INTEGERS containing the USEE tag values supported by the Service provider module. The integers correspond to symbols of type CSSM_USEE_TAG
	SampleTypes	MULTIUINT32	An array of 4-byte integers representing the sample types accepted by the service provider. The integers correspond to symbols of type CSSM_SAMPLE_TYPE
	Acl SubjectTypes	MULTIUINT32	An array of 4-byte integers representing the ACL subject types accepted by the service provider. The integers correspond to symbols of type CSSM_ACL_SUBJECT_TYPE
	AuthTags	MULTIUINT32	An array of 4-byte integers representing the Authorization tag values defined by the service provider. The integers correspond to symbols of type CSSM_ACL_AUTHORIZATION_TAG

* Indicates the primary database key.

13.9 CSP Capabilities Relation

The Cryptographic Service Provider (CSP) Capabilities relation contains attribute information for cryptographic services. Each CSP can have multiple entries. All fields except *AttributeValue* and *Description* form the concatenated primary database key. The description string is a short (about 64 bytes) description of the algorithm identified by the *AlgType* field.

	Field Name	**Field Data Type**	**Comment**
*	ModuleID	STRING	GUID (in string format) uniquely identifying the CSP module
*	SSID	UINT32	4 byte Subservice ID
*	UseeTag	UINT32	4-byte USEE tag associated with the attribute values. The integer correspond to symbols of type CSSM_USEE_TAG.
*	ContextType	UINT32	Class of cryptographic information (4 bytes) The integers correspond to symbols of type CSSM_CONTEXT_TYPE
*	AlgType	UINT32	Cryptographic algorithm supported by CSP (4 bytes). The integers correspond to symbols of type CSSM_ALGORITHMS.
*	GroupId	UINT32	4-byte identifier grouping all of the attributes associated with a single AlgType
*	AttributeType	UINT32	CSP attribute tag to identify the attribute value. The integers correspond to symbols of type CSSM_ATTRIBUTE_TYPE.
	AttributeValue	MULTIUINT32	Array of 4-byte values having the same AttributeType.
	Description	STRING	Human readable description of the algorithm (AlgType)

* Indicates the primary database key.

The AttributeValue field contains a list of the attributes expressed as 4-byte scalars corresponding to namespace as controlled by the CDSA specification. Each array of values is relative to the fields comprising the database key.

The use exemption (USEE) tag value of none (0) is used to describe the base capabilities of the CSP. All non-restricted algorithms and attributes are populated under USEE tag none. Restricted algorithms and appropriate key sizes have entries suitable for other USEE tag values.

Several attribute types defined by CDSA do not describe capabilities of a cryptographic service provider. These attribute types are not recorded with MDS. They are:

- CSSM_ATTRIBUTE_KEY
- CSSM_ATTRIBUTE_INIT_VECTOR
- CSSM_ATTRIBUTE_SALT
- CSSM_ATTRIBUTE_RANDOM
- CSSM_ATTRIBUTE_SEED
- CSSM_ATTRIBUTE_PASSPHRASE
- CSSM_ATTRIBUTE_ALG_PARAMS

- CSSM_ATTRIBUTE_LABEL
- CSSM_ATTRIBUTE_START_DATE
- CSSM_ATTRIBUTE_END_DATE
- CSSM_ATTRIBUTE_PRIME
- CSSM_ATTRIBUTE_BASE
- CSSM_ATTRIBUTE_SUBPRIME

Conventions for expressing attribute values as a MULTIUINT32 data type are described in terms of the attribute data type tag:

- CSSM_ATTRIBUTE_DATA_UINT32
  Each word (UINT32) represents an instance of a list of possible attribute values.
- CSSM_ATTRIBUTE_DATA_DATE
  The date is expressed in Year, Month, Day format in a 3-word MULTIUINT32 value. Each component occupies one 4-byte element. The first word contains the year, one byte for each ordinal. For example, December 31, 1998 is expressed as (0x01, 0x09, 0x09, 0x08) for the year, (0x00, 0x00, 0x 01, 0x02) for the month, and (0x00, 0x00, 0x03, 0x01) for the day.
- CSSM_ATTRIBUTE_DATA_RANGE
  The range is expressed in Min, Max format in a 2 word MULTIUINT32 value. For example, a range from 15 to 1025 would be (0x0000000F) and (0x00000401).
- CSSM_ATTRIBUTE_DATA_VERSION
  The version is expressed in Major, Minor format in a 2 word MULTIUINT32 value. For example, a version of 3.0 would be (0x00000003) and (0x00000000).

The CSSM_ATTRIBUTE_CUSTOM type can be specified. The service provider vendor is responsible for defining conventions for interpreting each member of the MULTIUINT32 element.

13.10 CSP Encapsulated Products Relation

This relation defines the attributes describing an third party, encapsulated product that is used in the implementation of the Cryptographic Service Provider. Information in the *CSP Encapsulated Products* relation is optional. It is provided at installation time by the CSP vendor. There are three classes of information:

1. A software product used in the implementation of the CSP.
2. A standard to which the implementation complies.
3. A reader device used for removable tokens/smartcards.

CSP vendors define the format of STRING fields. The format for flags fields is specific to the encapsulated product.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	ProductDesc	STRING	ASCII text description of the product encapsulated by the implementation.
	ProductVendor	STRING	ASCII text description of a software product encapsulated by the implementation.
	ProductVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ProductFlags	UINT32	Flags (4 bytes)
	CustomFlags	UINT32	More flags (4 bytes)
	StandardDesc	STRING	String describing the standards complied to by the implementation (e.g. PKCS11)
	StandardVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ReaderDesc	STRING	ASCII text description of the token reader device.
	ReaderVendor	STRING	ASCII text description of the reader device vendor.
	ReaderVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ReaderFirmwareVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ReaderFlags	UINT32	Flags (4 bytes). Currently valid values include: - CSSM_CSP_RDR_EXISTS (0x00000002): Device is a reader with removable token - CSSM_CSP_RDR_HW (0x00000004): Reader is a physical device
	ReaderCustomFlags	UINT32	More flags (4 bytes)
	ReaderSerialNumber	STRING	Text representation of the token reader device serial number.

* Indicates the primary database key.

13.11 CSP SmartcardInfo Relation

The information in the CSP Smartcard relation is updated each time a smartcard is inserted or removed from a reader device. The information is optional. CSP vendors define the format for STRING fields. The format for flags fields are specific to the smartcard.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	ScDesc	STRING	ASCII text description of the smartcard.
	ScVendor	STRING	ASCII text description of a smartcard.
	ScVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ScFirmwareVersion	STRING	Version string (in dotted high/low format - e.g. 2.0).
	ScFlags	UINT32	Flags (4 bytes). Currently valid values include: - CSSM_CSP_TOK_RNG (0x00000001) Device has a hardware random number generator - CSSM_CSP_TOK_CLOCK_EXISTS (0x00000040) Device has built-in real-time clock
	ScCustomFlags	UINT32	More flags (4 bytes)
	ScSerialNumber	STRING	Text representation of the smartcard serial number.

* Indicates the primary database key.

13.12 DL Primary Relation

The DL Primary relation describes capabilities of Data Library modules. This table may be updated in conjunction with CSSM *Insert* and *Remove* events.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProductVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	DLType	UINT32	Flag describing the backend implementation approach. The integer correspond to symbols of type CSSM_DLTYPE.
	QueryLimitsFlag	UINT32	Flag indicating query limits will be performed. Valid values include CSSM_QUERY_TIMELIMIT_NONE and CSSM_QUERY_SIZELIMIT_NONE.
	SampleTypes	MULTIUINT32	An array of 4-byte integers representing the sample types accepted by the service provider. The integers correspond to symbols of type CSSM_SAMPLE_TYPE
	Acl SubjectTypes	MULTIUINT32	An array of 4-byte integers representing the ACL subject types accepted by the service provider. The integers correspond to symbols of type CSSM_ACL_SUBJECT_TYPE
	AuthTags	MULTIUINT32	An array of 4-byte integers representing the Authorization tag values defined by the service provider. The integers correspond to symbols of type CSSM_ACL_AUTHORIZATION_TAG
	ConjunctiveOps	MULTIUINT32	Array of 4-byte integers describing the supported conjunctive operators.
	RelationalOps	MULTIUINT32	Array of 4-byte integers describing the supported relational operators.

* Indicates the primary database key.

13.13 DL Encapsulated Products Relation

The DL Encapsulated Products Relation provides information about third party products that are used in the implementation of the DL services. This information is optional, but it can be useful to users of the DL service provider.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying a service provider module
*	SSID	UINT32	4 byte Subservice ID
	ProductDesc	STRING	ASCII text description of the commercial product encapsulated by the implementation. This string is typically a product name. Examples include "Oracle RDBMS*", "GemStone*", "Microsoft Access"
	ProductVendor	STRING	ASCII text providing the name of the vendor of the encapsulated product. This string is the commercial business name of the vendor who markets the encapsulated product.
	ProductVersion	STRING	ASCII string providing a version number for the product named in the Product Description. The version number is formatted in dotted high/low format - e.g. 2.0.
	ProductFlags	UINT32	Flags describing product-specific features provided by the encapsulated product and used by the DL service provider to provide service to users. Examples include: - support for and use of stored data base queries - support for and use of audit trails - support for and use of virtual data views (4 bytes).
	StandardDesc	STRING	ASCII string describing an industry standard supported by the encapsulated product. Examples include: - an encapsulated RDBMS can support industry standard SQL - an encapsulated OODBMS can support industry standard O-SQL. Multiple standards can be listed in a semicolon-separated list.
	StandardVersion	STRING	ASCII string providing a version number for the industry standard named in the Standard Description. The version number is formatted in dotted high/low format - e.g. 2.0.
	Protocol	UINT32	Identifies the CSSM_NET_PROTOCOL supported by the encapsulated product (if any). Examples include CSSM_NET_PROTO_LDAP, CSSM_NET_PROTO_NDS, CSSM_NET_PROTO_HTTP
	RetrievalMode	UINT32	Retrieval modes supported by the service provider. The integer corresponds to the symbols of type CSSM_DB_RETRIEVAL_MODES.

* Indicates the primary database key.

13.14 CL Primary Relation

The CL Primary relation describes capabilities of Certificate Library modules. This relation can be updated in conjunction with CSSM Insert and Remove events.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProductVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	CertTypeFormat	UINT32	Certificate standard (e.g. X.509) & format (e.g. BER/DER). High word (2 bytes) is type, low word (2 bytes) is format. The high word (type) corresponds to symbols of type CSSM_CERT_TYPE and low word (format) corresponds to symbols of type CSSM_CERT_ENCODING.
	CrlTypeFormat	UINT32	Certificate revocation record format. The standard (e.g. X.509) & encoding format (e.g. BER/DER) are in high word low word. High word (2 bytes) is type, low word (2 bytes) is format. The high word (type) corresponds to symbols of type CSSM_CERT_TYPE and low word (format) corresponds to symbols of type CSSM_CERT_ENCODING.
	CertFieldNames	BLOB	Encapsulated array of OIDs in <length> <OID> format. Length is a 4-byte length followed by OID. The length byte order is in platform specific format.
	BundleTypeFormat	MULTIUINT32	Encapsulated array of the supported standards for importing collections of certificates into CertGroups. Certificate collection standard (e.g. PKCS7) & encoding format (e.g. BER/DER) where high word (2 bytes) is Type, low word (2 bytes) is Format. The high word (type) corresponds to symbols of type CSSM_CERT_TYPE and low word (format) corresponds to symbols of type CSSM_CERT_ENCODING.
	XlationTypeFormat	MULTIUINT32	Encapsulated array of supported standards for translating certificate formats. The certificate standard is the high word (2 bytes) as Type (e.g. X.509) & the low word (2 bytes) as encoding format (e.g. BER/DER).
	DefaultTemplateType	UINT32	An integer identifying the default template type

	TemplateFieldNames	BLOB	Encapsulated array of OIDs in <length> <OID> format. Length is a 4-byte length followed by OID. The length byte order is in platform specific format.

* Indicates the primary database key.

13.15 CL Encapsulated Products Relation

This relation provides implementation-specific information that may be useful to callers of the CL service provider. This relation can be used to describe a commercial product that is used to implement the CL module. Examples include third party Certification Authorities, Registration Authorities, and Authorization Servers.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID in string form describing service provider modules
*	SSID	UINT32	4 byte Subservice ID
	ProductDesc	STRING	ASCII text description of the commercial certificate services product or a certificate encode-decode product encapsulated by the implementation. This string is typically a product name. An example is "Snappy-soft ASN.1 Compiler"
	ProductVendor	STRING	ASCII text providing the name of the vendor of the encapsulated product. This string is the commercial business name of the vendor who markets the encapsulated product.
	ProductVersion	STRING	ASCII string providing a version number for the product named in the Product Description. The version number is formatted in dotted high/low format - e.g. 2.0.
	ProductFlags	UINT32	Flags describing product-specific features provided by the encapsulated product and used by the CL service provider to provide service to users. Examples include BER encoding only, BER/DER encoding, S-expression parsing.
	StandardDesc	STRING	ASCII string describing an industry standard supported by the encapsulated product. Examples include: - ASN.1 standard RFC 209 - SPKI S-expression IETF Draft spki-cert-theory-04 (November 1998) Multiple standards can be listed in a semicolon-separated list.
	StandardVersion	STRING	ASCII string providing a version number for the industry standard named in the Standard Description. The version number is formatted in dotted high/low format - e.g. 2.0.

* Indicates the primary database key.

13.16 TP Primary Relation

The TP Primary relation describes capabilities of Trust Policy Library modules. This relation can be updated in conjunction with CSSM *Insert* and *Remove* events.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProductVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	CertTypeFormat	UINT32	Certificate standard (e.g. X.509) & format (e.g. BER/DER). High word (2 bytes) is type, low word (2 bytes) is format. The high word (type) corresponds to symbols of type CSSM_CERT_TYPE and low word (format) corresponds to symbols of type CSSM_CERT_ENCODING.
	SampleTypes	MULTIUINT32	An array of 4-byte integers representing the sample types accepted by the servic e provider. The integers correspond to symbols of type CSSM_SAMPLE_TYPE
	Acl SubjectTypes	MULTIUINT32	An array of 4-byte integers representing the ACL subject types accepted by the s ervice provider. The integers correspond to symbols of type CSSM_ACL_SUBJECT_TYP E
	AuthTags	MULTIUINT32	An array of 4-byte integers representing the Authorization tag values defined by the service provider. The integers correspond to symbols of type CSSM_ACL_AUTHO RIZATION_TAG

* Indicates the primary database key.

13.17 TP Policy-OIDS Relation

The Policy OIDS relation lists the policy object identifiers recognized by the service provider. Information in the table can change in conjunction with CSSM *Insert* and *Remove* events. The policy objects implemented by the service provider are contained in this relation. There is an entry for each OID and Value. The ModuleID, SSID and OID fields identify a unique record.

	Field Name	**Field Data Type**	**Comment**
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
*	OID	BLOB	Policy object identifier interpreted by module
	Value	BLOB	Accompanying information. Uses OID to interpret contents.

* Indicates the primary database key.

13.18 TP Encapsulated Products Relation

This relation provided information about any third party products used in the implementation of the Trust Policy services. This information is optional, but can be useful to users of the trust policy services.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying a service provider module
*	SSID	UINT32	4 byte Subservice ID
	ProductDesc	STRING	ASCII text description of the commercial certificate services product or a certificate encode-decode product encapsulated by the implementation. This string is typically a product name. Examples include World-authority Certificate Service, Corporate CA, Mini-server CA.
	ProductVendor	STRING	ASCII text providing the name of the vendor of the encapsulated product. This string is the commercial business name of the vendor who markets the encapsulated product.
	ProductVersion	STRING	ASCII string providing a version number for the product named in the Product Description. The version number is formatted in dotted high/low format - e.g. 2.0.
	ProductFlags	UINT32	Flags describing product-specific features provided by the encapsulated product and used by the CL service provider to provide service to users. Examples include: - Key archive and recovery service - Support for Registration Authorities - Online Certificate Revocation Services - Online verification services (4 bytes)
	AuthorityRequestType	MULTIUINT32	Encapsulated array of supported certification authority (TP) requests this service provider can submit to a TP process. Example request type values include: CSSM_TP_AUTHORITY_REQUEST_CERTISSUE CSSM_TP_AUTHORITY_REQUEST_CERTREVOKE CSSM_TP_AUTHORITY_REQUEST_CERTVERIFY CSSM_TP_AUTHORITY_REQUEST_CRLISSUE
	StandardDesc	STRING	ASCII string describing an industry standard supported by the encapsulated product. Examples include: - IETF's OCSP for certificate revocation - IETF's PKIX-3 for certificate requests and recovery - PKCS#10 for certificate requests Multiple standards can be listed in a semicolon-separated list.
	StandardVersion	STRING	ASCII string providing a version number for the industry standard named in the Standard Description. The version number is formatted in

			dotted high/low format - e.g. 2.0.
	ProtocolDesc	STRING	ASCII string describing the protocol used to perform certificate request. (e.g. CMP)
	ProtocolFlags	UINT32	Flags describing protocol-specific features provided by the encapsulated product and used by the CL service provider to provide service to users(4 bytes)
	CertClassName	STRING	ASCII string naming a class or category of certificates managed by the encapsulated product. A certificate authority can issue multiple classes of certificates.
	RootCertificate	BLOB	The certificate of the encapsulated product. For a CA product, this is the CA's certificate. The certificate contains the trusted public key of the encapsulated product. The public key can be used for verification. The certificate is encoded as specified by RootCertTypeFormat
	RootCertType Format	UINT32	Specifies the certificate form (e.g. X.509) & encoding (e.g. BER/DER). High word (2 bytes) is Type/Form, low word (2 bytes) is Encoding.

* Indicates the primary database key.

13.19 MDS Schema Relation

The following relations provide information about the schema definition of all the relations managed under MDS. Applications can query the relation to learn about the schema of the CDSA relations. Administrative application can update the relation when creating new relations in MDS or adding new attributes to existing relations in MDS.

The schema relations can be queried by users and applications, but cannot be modified by users or applications.

MDS_SCHEMA_RELATIONS is a relation containing one record for each relation defined for the database. All fields are searchable. *RelationID* is the primary database key for this relation.

	Field Name	**Field Data Type**	**Comment**
*	RelationID	UINT32	A unique integer value identifying a relation stored and managed by MDS. The CDSA relations are identified by integers in the range [CSSM_DB_RELATIONID_MDS_START, CSSM_DB_RELATIONID_MDS_END].
	RelationName	STRING	The relation name in ASCII text.

* Indicates the primary database key.

MDS_SCHEMA_ATTRIBUTES is a relation containing one record for each attribute defined for the MDS database. All fields are searchable. The starred(*) fields form the primary database key for this relation.

	Field Name	**Field Data Type**	**Comment**
*	RelationID	UINT32	A unique integer value identifying a relation stored and managed by MDS. The CDSA relations are identified by integers in the range [CSSM_DB_RELATIONID_MDS_START, CSSM_DB_RELATIONID_MDS_END].
*	AttributeID	UINT32	A number identifying an attribute in the relation identified by *RelationId*
	AttributeNameFormat	UINT32	Format of *AttributeName*
	AttributeName	STRING	Name of attribute
	AttributeNameID	BLOB	Name of attribute expressed as an infinite precision number (aka OID).
	AttributeFormat	UINT32	Data type of values associated with the attribute

* Indicates the primary database key.

MDS_SCHEMA_INDEXES is a relation containing one record for each index defined for the MDS database. All fields are searchable. The starred(*) fields form the primary database key for this relation.

	Field Name	Field Data Type	Comment
*	RelationID	UINT32	A unique integer value identifying a relation stored and managed by MDS. The CDSA relations are identified by integers in the range [CSSM_DB_RELATIONID_MDS_START, CSSM_DB_RELATIONID_MDS_END].
*	IndexID	UINT32	A number uniquely identifying an index. Unique indexes will use the same *IndexID* for each attribute (*AttributeID*) comprising the concatenated key of the unique index.
*	AttributeID	UINT32	An integer value uniquely identifying an attribute within the relation identified by *RelationID*.
	IndexType	UINT32	Type of index (part of the unique index or a non-unique index).
	IndexedDataLocation	UINT32	Source of the information used to create the index

* Indicates the primary database key.

13.20 AC Primary Relation

The AC Primary relation describes capabilities of Authorization Computation service provider modules. This relation can be updated in conjunction with CSSM *Insert* and *Remove* events.

	Field Name	**Field Data Type**	**Comment**
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Human readable name. This is the filename of the library that performs cross-check operations.
	ProductVersion	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.

* Indicates the primary database key.

13.21 KR Primary Relation

The KR Primary relation describes capabilities of Key Recovery modules. This relation can be updated in conjunction with CSSM Insert and Remove events to add or remove records as appropriate.

	Field Name	Field Data Type	Comment
*	ModuleID	STRING	GUID (in string format) uniquely identifying service provider modules
*	SSID	UINT32	4 byte Subservice ID A subservice can be used to support different KR mechanisms. Examples include Encapsulation, Escrow.
	Manifest	BLOB	Reserved for future use.
	ModuleName	STRING	Manifest Section Name.
	CompatCSSMVersion	STRING	Lowest compatible CSSM version
	Version	STRING	Service provider version string (in dotted high/low format - e.g. 2.0).
	Vendor	STRING	Service provider vendor name in ASCII text.
	Description	STRING	Human-readable description of this KR subservice.
	ConfigFileLocation	STRING	Search path in platform-specific format for KR configuration files

* Indicates the primary database key.

Chapter 14

MDS Name Space and Directory Structures

The Module Directory Service leverages name space and types defined by the CDSA Data Storage Library Services API. Even though MDS is a standalone service, application developer can use a similar programming paradigm to access MDS-managed databases and DL module-managed databases.

MDS defines several new data types in addition to those defined by CDSA's Data Storage Library Services. The new types define names for the MDS relations and the records stored in those relations. The MDS-specific names and data structures are defined below.

14.1 MDS Name Space

A relation identifier, also referred to as CSSM_DB_RECORDTYPE, identifies an MDS relation. MDS relation identifiers are allocated from the CSSM_DB_RECORDTYPE name space. The MDS system reserves and uses the following name space definitions.

```
/*  MDS predefined values for a 16K name space */
#define CSSM_DB_RELATIONID_MDS_START (0x40000000)
#define CSSM_DB_RELATIONID_MDS_END (0x40004000)
```

14.2 Object Directory

This constant defines the programmatic name for type of records stored in the MDS Object Directory database.

```
#define MDS_OBJECT_RECORDTYPE (CSSM_DB_RELATIONID_MDS_START)
```

14.3 CDSA Directory

These constants define the programmatic names for the record types stored in the MDS CDSA Directory database.

```
#define MDS_CDSA_SCHEMA_START         (MDS_OBJECT_RECORDTYPE)
#define MDS_CDSADIR_CSSM_RECORDTYPE   (MDS_CDSA_SCHEMA_START + 1)
#define MDS_CDSADIR_KRMM_RECORDTYPE   (MDS_CDSA_SCHEMA_START + 2)
#define MDS_CDSADIR_EMM_RECORDTYPE    (MDS_CDSA_SCHEMA_START + 3)
#define MDS_CDSADIR_COMMON_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 4)
#define MDS_CDSADIR_CSP_PRIMARY_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 5)
#define MDS_CDSADIR_CSP_CAPABILITY_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 6)
#define MDS_CDSADIR_CSP_ENCAPSULATED_PRODUCT_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 7)
#define MDS_CDSADIR_CSP_SC_INFO_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 8)
#define MDS_CDSADIR_DL_PRIMARY_RECORDTYPE
                                      (MDS_CDSA_SCHEMA_START + 9)
#define MDS_CDSADIR_DL_ENCAPSULATED_PRODUCT_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 10)
#define MDS_CDSADIR_CL_PRIMARY_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 11)
```

```
#define MDS_CDSADIR_CL_ENCAPSULATED_PRODUCT_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 12)
#define MDS_CDSADIR_TP_PRIMARY_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 13)
#define MDS_CDSADIR_TP_OIDS_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 14)
#define MDS_CDSADIR_TP_ENCAPSULATED_PRODUCT_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 15)
#define MDS_CDSADIR_EMM_PRIMARY_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 16)
#define MDS_CDSADIR_AC_PRIMARY_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 17)
#define MDS_CDSADIR_KR_PRIMARY_RECORDTYPE
                                     (MDS_CDSA_SCHEMA_START + 18)
#define MDS_CDSADIR_MDS_SCHEMA_RELATIONS
                                     (MDS_CDSA_SCHEMA_START + 19)
#define MDS_CDSADIR_MDS_SCHEMA_ATTRIBUTES
                                     (MDS_CDSA_SCHEMA_START + 20)
#define MDS_CDSADIR_MDS_SCHEMA_INDEXES
                                     (MDS_CDSA_SCHEMA_START + 21)
```

14.3.1 CDSA Relation Attributes

These constants define the programmatic names for the attributes of the CDSA relations. The constant assigned to an attribute must be unique with the relation containing that attribute.

```
/*  MDS predefined values for a 16K name space */
#define CSSM_DB_ATTRIBUTE_MDS_START (0x40000000)
#define CSSM_DB_ATTRIBUTE_MDS_END   (0x40004000)

#define MDS_CDSAATTR_MODULE_ID              (CSSM_DB_ATTRIBUTE_MDS_START + 1)
#define MDS_CDSAATTR_MANIFEST               (CSSM_DB_ATTRIBUTE_MDS_START + 2)
#define MDS_CDSAATTR_MODULE_NAME            (CSSM_DB_ATTRIBUTE_MDS_START + 3)
#define MDS_CDSAATTR_PATH                   (CSSM_DB_ATTRIBUTE_MDS_START + 4)
#define MDS_CDSAATTR_CDSAVERSION            (CSSM_DB_ATTRIBUTE_MDS_START + 5)
#define MDS_CDSAATTR_VENDOR                 (CSSM_DB_ATTRIBUTE_MDS_START + 6)
#define MDS_CDSAATTR_DESC                   (CSSM_DB_ATTRIBUTE_MDS_START + 7)
#define MDS_CDSAATTR_POLICY_STMT            (CSSM_DB_ATTRIBUTE_MDS_START + 8)
#define MDS_CDSAATTR_EMM_SPEC_VERSION       (CSSM_DB_ATTRIBUTE_MDS_START + 9)
#define MDS_CDSAATTR_EMM_VERSION            (CSSM_DB_ATTRIBUTE_MDS_START + 10)
#define MDS_CDSAATTR_EMM_VENDOR             (CSSM_DB_ATTRIBUTE_MDS_START + 11)
#define MDS_CDSAATTR_EMM_TYPE               (CSSM_DB_ATTRIBUTE_MDS_START + 12)
#define MDS_CDSAATTR_SSID                   (CSSM_DB_ATTRIBUTE_MDS_START + 13)
#define MDS_CDSAATTR_SERVICE_TYPE           (CSSM_DB_ATTRIBUTE_MDS_START + 14)
#define MDS_CDSAATTR_NATIVE_SERVICES        (CSSM_DB_ATTRIBUTE_MDS_START + 15)
#define MDS_CDSAATTR_DYNAMIC_FLAG           (CSSM_DB_ATTRIBUTE_MDS_START + 16)
#define MDS_CDSAATTR_MULTITHREAD_FLAG       (CSSM_DB_ATTRIBUTE_MDS_START + 17)
#define MDS_CDSAATTR_SERVICE_MASK           (CSSM_DB_ATTRIBUTE_MDS_START + 18)
#define MDS_CDSAATTR_CSP_TYPE               (CSSM_DB_ATTRIBUTE_MDS_START + 19)
#define MDS_CDSAATTR_CSP_FLAGS              (CSSM_DB_ATTRIBUTE_MDS_START + 20)
#define MDS_CDSAATTR_CSP_CUSTOMFLAGS        (CSSM_DB_ATTRIBUTE_MDS_START + 21)
#define MDS_CDSAATTR_USEE_TAGS              (CSSM_DB_ATTRIBUTE_MDS_START + 22)
#define MDS_CDSAATTR_CONTEXT_TYPE           (CSSM_DB_ATTRIBUTE_MDS_START + 23)
#define MDS_CDSAATTR_ALG_TYPE               (CSSM_DB_ATTRIBUTE_MDS_START + 24)
#define MDS_CDSAATTR_GROUP_ID               (CSSM_DB_ATTRIBUTE_MDS_START + 25)
#define MDS_CDSAATTR_ATTRIBUTE_TYPE         (CSSM_DB_ATTRIBUTE_MDS_START + 26)
#define MDS_CDSAATTR_ATTRIBUTE_VALUE        (CSSM_DB_ATTRIBUTE_MDS_START + 27)
#define MDS_CDSAATTR_PRODUCT_DESC           (CSSM_DB_ATTRIBUTE_MDS_START + 28)
#define MDS_CDSAATTR_PRODUCT_VENDOR         (CSSM_DB_ATTRIBUTE_MDS_START + 29)
#define MDS_CDSAATTR_PRODUCT_VERSION        (CSSM_DB_ATTRIBUTE_MDS_START + 30)
#define MDS_CDSAATTR_PRODUCT_FLAGS          (CSSM_DB_ATTRIBUTE_MDS_START + 31)
#define MDS_CDSAATTR_PRODUCT_CUSTOMFLAGS    (CSSM_DB_ATTRIBUTE_MDS_START + 32)
#define MDS_CDSAATTR_STANDARD_DESC          (CSSM_DB_ATTRIBUTE_MDS_START + 33)
#define MDS_CDSAATTR_STANDARD_VERSION       (CSSM_DB_ATTRIBUTE_MDS_START + 34)
#define MDS_CDSAATTR_READER_DESC            (CSSM_DB_ATTRIBUTE_MDS_START + 35)
```

```
#define MDS_CDSAATTR_READER_VENDOR          (CSSM_DB_ATTRIBUTE_MDS_START + 36)
#define MDS_CDSAATTR_READER_VERSION         (CSSM_DB_ATTRIBUTE_MDS_START + 37)
#define MDS_CDSAATTR_READER_FWVERSION       (CSSM_DB_ATTRIBUTE_MDS_START + 38)
#define MDS_CDSAATTR_READER_FLAGS           (CSSM_DB_ATTRIBUTE_MDS_START + 39)
#define MDS_CDSAATTR_READER_CUSTOMFLAGS     (CSSM_DB_ATTRIBUTE_MDS_START + 40)
#define MDS_CDSAATTR_READER_SERIALNUMBER    (CSSM_DB_ATTRIBUTE_MDS_START + 41)
#define MDS_CDSAATTR_SC_DESC                (CSSM_DB_ATTRIBUTE_MDS_START + 42)
#define MDS_CDSAATTR_SC_VENDOR              (CSSM_DB_ATTRIBUTE_MDS_START + 43)
#define MDS_CDSAATTR_SC_VERSION             (CSSM_DB_ATTRIBUTE_MDS_START + 44)
#define MDS_CDSAATTR_SC_FWVERSION           (CSSM_DB_ATTRIBUTE_MDS_START + 45)
#define MDS_CDSAATTR_SC_FLAGS               (CSSM_DB_ATTRIBUTE_MDS_START + 46)
#define MDS_CDSAATTR_SC_CUSTOMFLAGS         (CSSM_DB_ATTRIBUTE_MDS_START + 47)
#define MDS_CDSAATTR_SC_SERIALNUMBER        (CSSM_DB_ATTRIBUTE_MDS_START + 48)
#define MDS_CDSAATTR_DL_TYPE                (CSSM_DB_ATTRIBUTE_MDS_START + 49)
#define MDS_CDSAATTR_QUERY_LIMITS           (CSSM_DB_ATTRIBUTE_MDS_START + 50)
#define MDS_CDSAATTR_CONJUNCTIVE_OPS        (CSSM_DB_ATTRIBUTE_MDS_START + 51)
#define MDS_CDSAATTR_RELATIONAL_OPS         (CSSM_DB_ATTRIBUTE_MDS_START + 52)
#define MDS_CDSAATTR_PROTOCOL               (CSSM_DB_ATTRIBUTE_MDS_START + 53)
#define MDS_CDSAATTR_CERT_TYPEFORMAT        (CSSM_DB_ATTRIBUTE_MDS_START + 54)
#define MDS_CDSAATTR_CRL_TYPEFORMAT         (CSSM_DB_ATTRIBUTE_MDS_START + 55)
#define MDS_CDSAATTR_CERT_FIELDNAMES        (CSSM_DB_ATTRIBUTE_MDS_START + 56)
#define MDS_CDSAATTR_BUNDLE_TYPEFORMAT      (CSSM_DB_ATTRIBUTE_MDS_START + 57)
#define MDS_CDSAATTR_CERT_CLASSNAME         (CSSM_DB_ATTRIBUTE_MDS_START + 58)
#define MDS_CDSAATTR_ROOTCERT               (CSSM_DB_ATTRIBUTE_MDS_START + 59)
#define MDS_CDSAATTR_ROOTCERT_TYPEFORMAT    (CSSM_DB_ATTRIBUTE_MDS_START + 60)
#define MDS_CDSAATTR_VALUE                  (CSSM_DB_ATTRIBUTE_MDS_START + 61)
#define MDS_CDSAATTR_REQCREDENTIALS         (CSSM_DB_ATTRIBUTE_MDS_START + 62)
#define MDS_CDSAATTR_SAMPLETYPES            (CSSM_DB_ATTRIBUTE_MDS_START + 63)
#define MDS_CDSAATTR_ACLSUBJECTTYPES        (CSSM_DB_ATTRIBUTE_MDS_START + 64)
#define MDS_CDSAATTR_AUTHTAGS               (CSSM_DB_ATTRIBUTE_MDS_START + 65)
#define MDS_CDSAATTR_USEETAG                (CSSM_DB_ATTRIBUTE_MDS_START + 66)
#define MDS_CDSAATTR_RETRIEVALMODE          (CSSM_DB_ATTRIBUTE_MDS_START + 67)
#define MDS_CDSAATTR_OID                    (CSSM_DB_ATTRIBUTE_MDS_START + 68)
#define MDS_CDSAATTR_XLATIONTYPEFORMAT      (CSSM_DB_ATTRIBUTE_MDS_START + 69)
#define MDS_CDSAATTR_DEFAULT_TEMPLATE_TYPE  (CSSM_DB_ATTRIBUTE_MDS_START + 70)
#define MDS_CDSAATTR_TEMPLATE_FIELD_NAMES   (CSSM_DB_ATTRIBUTE_MDS_START + 71)
#define MDS_CDSAATTR_AUTHORITY_REQUEST_TYPE (CSSM_DB_ATTRIBUTE_MDS_START + 72)
#define MDS_CDSAATTR_CONFIG_FLAG            (CSSM_DB_ATTRIBUTE_MDS_START + 73)
#define MDS_CDSAATTR_CSSM_GUID              (CSSM_DB_ATTRIBUTE_MDS_START + 74)
#define MDS_CDSAATTR_POLICY_TYPE            (CSSM_DB_ATTRIBUTE_MDS_START + 75)
#define MDS_CDSAATTR_POLICY_NAME            (CSSM_DB_ATTRIBUTE_MDS_START + 76)
#define MDS_CDSAATTR_POLICY_PATH            (CSSM_DB_ATTRIBUTE_MDS_START + 77)
#define MDS_CDSAATTR_POLICY_INFO            (CSSM_DB_ATTRIBUTE_MDS_START + 78)
#define MDS_CDSAATTR_POLICY_MANIFEST        (CSSM_DB_ATTRIBUTE_MDS_START + 79)
```

14.4 MDS Meta-Data Names

These constants define the programmatic names for the meta-data attributes that describe the MDS relations.

```
/** Meta-data names for the MDS Object directory relation **/
#define MDS_OBJECT_NUM_RELATIONS  (1)
#define MDS_OBJECT_NUM_ATTRIBUTES (5)

/** Defined constant for # of relations in the CDSA directory **/
#define MDS_CDSADIR_NUM_RELATIONS (19)

/** Meta-data names for the MDS CSSM relation **/
#define MDS_CDSADIR_CSSM_NUM_ATTRIBUTES (5)

/** Meta-data names for the MDS KRMM relation **/
#define MDS_CDSADIR_KRMM_NUM_ATTRIBUTES (6)
```

```
/** Meta-data names for the MDS EMM relation **/
#define MDS_CDSADIR_EMM_NUM_ATTRIBUTES (11)

/** Meta-data names for the MDS Common relation **/
#define MDS_CDSADIR_COMMON_NUM_ATTRIBUTES (9)

/** Meta-data names for the MDS CSP Primary relation **/
#define MDS_CDSADIR_CSP_PRIMARY_NUM_ATTRIBUTES (13)

/** Meta-data names for the MDS CSP Capabilities relation **/
#define MDS_CDSADIR_CSP_CAPABILITY_NUM_ATTRIBUTES (9)

/** Meta-data names for the MDS CSP Encapsulated Product relation **/
#define MDS_CDSADIR_CSP_ENCAPSULATED_PRODUCT_NUM_ATTRIBUTES (16)

/** Meta-data names for the MDS CSP SmartcardInfo relation **/
#define MDS_CDSADIR_CSP_SC_INFO_NUM_ATTRIBUTES (9)

/** Meta-data names for the MDS DL Primary relation **/
#define MDS_CDSADIR_DL_PRIMARY_NUM_ATTRIBUTES (13)

/** Meta-data names for the MDS DL Encapsulated Product relation **/
#define MDS_CDSADIR_DL_ENCAPSULATED_PRODUCT_NUM_ATTRIBUTES (10)

/** Meta-data names for the MDS CL Primary relation **/
#define MDS_CDSADIR_CL_PRIMARY_NUM_ATTRIBUTES (13)

/** Meta-data names for the MDS CL Encapsulated Product relation **/
#define MDS_CDSADIR_CL_ENCAPSULATED_PRODUCT_NUM_ATTRIBUTES (8)

/** Meta-data names for the MDS TP Primary relation **/
#define MDS_CDSADIR_TP_PRIMARY_NUM_ATTRIBUTES (10)

/** Meta-data names for the MDS TP Policy-OIDS relation **/
#define MDS_CDSADIR_TP_OIDS_NUM_ATTRIBUTES  (4)

/** Meta-data names for the MDS TP Encapsulated Product relation **/
#define MDS_CDSADIR_TP_ENCAPSULATED_PRODUCT_NUM_ATTRIBUTES (13)

/** Meta-data names for MDS EMM Service Provider Primary relation **/
#define MDS_CDSADIR_EMM_PRIMARY_NUM_ATTRIBUTES (11)

/** Meta-data names for MDS AC Primary relation **/
#define MDS_CDSADIR_AC_PRIMARY_NUM_ATTRIBUTES (6)

/** Meta-data names for the MDS KR relation **/
#define MDS_CDSADIR_KR_PRIMARY_RELATION_NUM_ATTRIBUTES (8)

/** Meta-data names for MDS Schema relation **/
#define MDS_CDSADIR_SCHEMA_RELATONS_NUM_ATTRIBUTES (2)
#define MDS_CDSADIR_SCHEMA_ATTRIBUTES_NUM_ATTRIBUTES (6)
#define MDS_CDSADIR_SCHEMA_INDEXES_NUM_ATTRIBUTES (5)
```

14.5 Data Structure

MDS defines a small number of data structures that are visible to the user through the MDS APIs. MDS type definitions are dependent on CDSA type definitions. These type are re-qualified as MDS data types to separate the MDS type space from CSSM and DL name spaces.

14.5.1 MDS_HANDLE

This defines an opaque handle used to identify the MDS context in which a user can receive MDS services.

```
typedef CSSM_DL_HANDLE     MDS_HANDLE;
```

14.5.2 MDS_DB_HANDLE

This defines an opaque handle used to identify an MDS-managed database.

```
typedef CSSM_DL_DB_HANDLE    MDS_DB_HANDLE;
```

14.5.3 MDS_FUNC

This structure defines a table of function pointer returned by MDS to a user when a service context has been established between MDS and a user. The user accesses MDS services through these function pointers.

```
typedef struct mds_funcs {
    CSSM_RETURN (CSSMAPI *DbOpen)
        (MDS_HANDLE MdsHandle,
        const char *DbName,
        const CSSM_NET_ADDRESS *DbLocation,
        CSSM_DB_ACCESS_TYPE AccessRequest,
        const CSSM_ACCESS_CREDENTIALS *AccessCred,
        const void *OpenParameters,
        CSSM_DB_HANDLE *hMds);
    CSSM_RETURN (CSSMAPI *DbClose)
        (MDS_DB_HANDLE MdsDbHandle);
    CSSM_RETURN (CSSMAPI *GetDbNames)
        (MDS_HANDLE MdsHandle,
        CSSM_NAME_LIST_PTR *NameList);
    CSSM_RETURN (CSSMAPI *GetDbNameFromHandle)
        (MDS_DB_HANDLE MdsDbHandle,
        char **DbName);
    CSSM_RETURN (CSSMAPI *FreeNameList)
        (MDS_HANDLE MdsHandle,
        CSSM_NAME_LIST_PTR NameList);
    CSSM_RETURN (CSSMAPI *DataInsert)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_DB_RECORDTYPE RecordType,
        const CSSM_DB_RECORD_ATTRIBUTE_DATA *Attributes,
        const CSSM_DATA *Data,
        CSSM_DB_UNIQUE_RECORD_PTR *UniqueId);
    CSSM_RETURN (CSSMAPI *DataDelete)
        (MDS_DB_HANDLE MdsDbHandle,
        const CSSM_DB_UNIQUE_RECORD *UniqueRecordIdentifier);
    CSSM_RETURN (CSSMAPI *DataModify)
```

```
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_DB_RECORDTYPE RecordType,
        CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier,
        const CSSM_DB_RECORD_ATTRIBUTE_DATA *AttributesToBeModified,
        const CSSM_DATA *DataToBeModified,
        CSSM_DB_MODIFY_MODE ModifyMode);
    CSSM_RETURN (CSSMAPI *DataGetFirst)
        (MDS_DB_HANDLE MdsDbHandle,
        const CSSM_QUERY *Query,
        CSSM_HANDLE_PTR ResultsHandle,
        CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
        CSSM_DATA_PTR Data,
        CSSM_DB_UNIQUE_RECORD_PTR *UniqueId);
    CSSM_RETURN (CSSMAPI *DataGetNext)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_HANDLE ResultsHandle,
        CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
        CSSM_DATA_PTR Data,
        CSSM_DB_UNIQUE_RECORD_PTR *UniqueId);
    CSSM_RETURN (CSSMAPI *DataAbortQuery)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_HANDLE ResultsHandle);
    CSSM_RETURN (CSSMAPI *DataGetFromUniqueRecordId)
        (MDS_DB_HANDLE MdsDbHandle,
        const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord,
        CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
        CSSM_DATA_PTR Data);
    CSSM_RETURN (CSSMAPI *FreeUniqueRecord)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord);
    CSSM_RETURN (CSSMAPI *CreateRelation)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_DB_RECORDTYPE RelationID,
        const char *RelationName,
        uint32 NumberOfAttributes,
        const CSSM_DB_SCHEMA_ATTRIBUTE_INFO *pAttributeInfo,
        uint32 NumberOfIndexes,
        const CSSM_DB_SCHEMA_INDEX_INFO *pIndexInfo);
    CSSM_RETURN (CSSMAPI *DestroyRelation)
        (MDS_DB_HANDLE MdsDbHandle,
        CSSM_DB_RECORDTYPE RelationID);
} MDS_FUNCS, *MDS_FUNCS_PTR;
```

Chapter 15

Module Directory Services APIs

The Module Directory Service API leverages a large subset of the CDSA Data Storage Library API to support query, update and schema management operations. Because MDS is a separate system from CDSA, MDS defines a small set of context management functions. The user must invoke these functions to create and destroy service contexts with MDS.

15.1 MDS Context APIs

The man-page definitions for Key Recovery Module Management Operations are presented in this section.

NAME

MDS_Initialize

SYNOPSIS

```
CSSM_RETURN CSSMAPI MDS_Initialize
    (const CSSM_GUID *pCallerGuid,
    const CSSM_DATA *pCallerManifest,
    const CSSM_MEMORY_FUNCS *pMemoryFunctions,
    MDS_FUNCS_PTR pDlFunctions,
    MDS_HANDLE *hMds)
```

DESCRIPTION

This function initiates a service context with MDS and returns an opaque handle corresponding to that context. The caller provides memory functions that MDS can use to manage memory in the caller's space on behalf of the caller. The caller also provides input/output table pDlFunctions to get access to MDS databases.

If the caller is a CDSA service provider that will require write-access to an MDS database, (such as a module that supports dynamic insertion and removal events), then the caller can provide the caller's GUID as input parameter *pCallerGuid*. When provided as input, the GUID is associated with the MDS handle and is used during *DbOpen* processing. If write-access is requested during *DbOpen*, MDS uses the associated GUID to locate the service provider's signed manifest credentials in the *DS Common* relation. The service provider module and its credentials are verified to ensure that write-access is permitted on this database by this module.

The installers will have to provide the *pCallerManifest* instead of *pCallerGuid*, as GUID cannot be used to locate an application unless it is installed. Only one of the two parameters *pCallerGuid* and *pCallerManifest* should be non NULL in an *MDS_Initialize*() call, otherwise an error will be returned.

PARAMETERS

pCallerGuid (input/optional)
The GUID of the module calling MDS.

pCallerManifest (input/optional)
The Manifest of the module calling MDS.

pMemoryFunctions (input)
The memory-management routines MDS uses to allocate query results on behalf of the caller.

pDlFunctions (output)
The function table containing MDS programming interfaces for database access.

hMds (output)
A new handle that can be used to interact with the MDS. The value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

This API returns CSSM Data Storage Library error codes from the function *CSSM_DL_DbCreate*(). These error codes are listed below. See also the Error Codes and Error Values section in the Data Storage Library Services API.

CSSMERR_DL_INVALID_POINTER

CSSMERR_DL_INTERNAL_ERROR

CSSMERR_DL_MEMORY_ERROR

CSSMERR_DL_FUNCTION_FAILED

NAME

MDS_Terminate

SYNOPSIS

```
CSSM_RETURN CSSMAPI MDS_Terminate
    (MDS_HANDLE MdsHandle)
```

DESCRIPTION

This function terminates the MDS service context identified by the opaque *MdsHandle*. The MDS handle is invalidated and MDS frees all internal resources associated with the context.

PARAMETERS

MdsHandle (input)
The MDS handle corresponding to the context being terminated.

RETURN

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

This API returns CSSM Data Storage Library error codes from the function *CSSM_DL_DbCreate*( ). These error codes are listed below. See also the Error Codes and Error Values section in the Data Storage Library Services API.

CSSMERR_DL_INVALID_DL_HANDLE

15.2 MDS Installation APIs

The man-page definitions for MDS Installation functions are presented in this section.

NAME

MDS_Install

SYNOPSIS

```
CSSM_RETURN CSSMAPI MDS_Install
    (MDS_HANDLE MdsHandle)
```

DESCRIPTION

This function creates the Object Directory database containing the *Object* relation, and the CDSA Directory database containing the set of CDSA-specific relations defined in this specification. The *MdsHandle* identifies an MDS context created by invoking *MDS_Initialize*( ). The context contains information about the access rights of the caller. Write-access is required to perform this operation.

PARAMETERS

MdsHandle (input)
The MDS handle identifying an MDS context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

This API returns CSSM Data Storage Library error codes from the function *CSSM_DL_DbCreate*( ). These error codes are listed below. See also the **Error Codes and Error Values** section in the **Data Storage Library Services API**

CSSMERR_DL_INVALID_DL_HANDLE
CSSMERR_DL_DATASTORE_ALREADY_EXISTS
CSSMERR_DL_INVALID_ACCESS_REQUEST
CSSMERR_DL_INVALID_DB_LOCATION
CSSMERR_DL_INVALID_DB_NAME
CSSMERR_DL_INVALID_OPEN_PARAMETERS
CSSMERR_DL_INVALID_RECORD_INDEX
CSSMERR_DL_INVALID_RECORDTYPE
CSSMERR_DL_INVALID_FIELD_NAME
CSSMERR_DL_UNSUPPORTED_FIELD_FORMAT
CSSMERR_DL_UNSUPPORTED_INDEX_INFO
CSSMERR_DL_UNSUPPORTED_LOCALITY
CSSMERR_DL_UNSUPPORTED_NUM_ATTRIBUTES
CSSMERR_DL_UNSUPPORTED_NUM_INDEXES
CSSMERR_DL_UNSUPPORTED_NUM_RECORDTYPES
CSSMERR_DL_UNSUPPORTED_RECORDTYPE
CSSMERR_DL_FIELD_SPECIFIED_MULTIPLE
CSSMERR_DL_INCOMPATIBLE_FIELD_FORMAT
CSSMERR_DL_INVALID_PARSING_MODULE

MDS_Uninstall

NAME

MDS_Uninstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI MDS_Uninstall
    (MDS_HANDLE MdsHandle)
```

DESCRIPTION

This function deletes the Object Directory database containing the *Object* relation, and the CDSA Directory database containing the set of CDSA-specific relations defined in this specification. The *MdsHandle* identifies the MDS context created by invoking *MDS_Initialize*(). The context contains information about the access rights of the caller. Write-access is required to perform this operation.

PARAMETERS

MdsHandle (input)
The MDS handle identifying a valid MDS context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

This API returns CSSM Data Storage Library error codes from the function *CSSM_DL_DbDelete*(). These error codes are listed below. See also the **Error Codes and Error Values** section in the **Data Storage Library Services API**

CSSMERR_DL_INVALID_DL_HANDLE
CSSMERR_DL_DATASTORE_IS_OPEN
CSSMERR_DL_INVALID_DB_LOCATION
CSSMERR_DL_INVALID_DB_NAME
CSSMERR_DL_DATASTORE_DOESNOT_EXIST

15.3 MDS Database Service APIs

MDS uses a large subset of the CDSA Data Storage Library APIs to support query, update, and schema management operations. The MDS supported functions are defined by the function table MDS_FUNCS. The service performed by each of these functions is the same as the service defined for the corresponding Data Storage Library operations.

For the definitions of these operations, refer to the relevant chapters on Data Storage Library Services API, in this Technical Standard.

15.4 Write-Access to MDS Databases

15.4.1 Updating MDS Schema

The MDS schema may require updating after initial installation. Conditions requiring updates are:

- MDS specification change
- EMM service provider installation
- CDSA versioning and migration
- Proprietary schema extensions.

Schema can be updated using *CreateRelation* and *DestroyRelation*. Schema update operations require special privilege and exclusive access to database files. The mechanisms for enforcing privileged access include:

- Privileged application
- File permissions
- Administrative passphrase.

Exclusive access to database schema requires opening the MDS database with access rights defined by CSSM_DB_ACCESS_PRIVILEGED.

15.4.2 Updating MDS Databases

MDS records may be inserted, modified or deleted using MDS interfaces. The MDS service must be opened with write permissions CSSM_DB_ACCESS_WRITE for update operations to succeed. Read permission is needed CSSM_DB_ACCESS_READ for query operations to succeed.

Additional privileges may be needed to open a database with CSSM_DB_ACCESS_WRITE. Privileges may be acquired, and privileged access enforced, by file permissions and manifest based cross-check.

Installation programs and CDSA service providers that support dynamic insert/remove events are the primary software modules that need to update records in MDS databases. MDS requires that these callers provide signed manifest credentials authorizing write access to MDS databases. The signed manifest must contain an MDS access mask authorizing write access for the caller.

An installer provides its signed manifest credential as an explicit input parameter to the *MDS_Initialize*( ) function. When a signed manifest is provided, MDS proceeds as follows:

- Verifies the integrity and authenticity of the signed manifest
- Verifies the integrity of the caller's executable code
- If both verifications succeed, extracts the access control information provided in the signed manifest, and associates it with an MDS handle returned by the function.

CDSA service providers that support insert/remove events need write access to MDS databases. The signed manifest of a CDSA service provider or any CDSA application is stored in the MDS database during installation of the service provider or application. The service provider's record is identified by the provider's GUID. Similarly application's record is identified by the application's GUID. When the service provider or application calls the *MDS_Initialize*( ) function, the GUID is passed as an input and signed manifest parameter of *MDS_Initialize*( ) should be NULL. When a GUID is provided, MDS proceeds as follows:

- Uses the GUID to retrieve the service provider's signed manifest from the MDS database
- Extracts the access control information provided in the signed manifest, and associates it with an MDS handle returned by the function.

The integrity and authenticity of the service provider has already been checked when the service provider was loaded into the CDSA environment.

When MDS databases are opened, an access control mask can be applied to the access type requested database open operation.

15.4.3 Manifest Attributes for MDS Access Control Privileges

Signed Manifests, as defined elsewhere in this specification, contain a set of name-value pairs, called *attributes*. Access privileges are granted to a module by adding an appropriate attribute to the module's signed manifest. The tag name for this attribute is CDSA_ACCESS_TAG

The value associated with the tag name sets the access permission granted to the module. Valid values are those access flag values defined by the CDSA Data Storage Library service definition of CSSM_DB_ACCESS_TYPE.

CDSA_DB_ACCESS_TYPE is a base64 encodeed, unsigned 32-bit integer in big-endian ordering.

Chapter 16

MDS Administration

MDS administration is partially platform-dependent. The general activities of MDS administration are described in this chapter. Specific activities that can be defined in a platform-independent are defined for use on all platforms.

16.1 MDS Installation

The MDS service is installed separately from any particular instance of CDSA. However, CDSA installation is dependent on MDS. The CDSA installation scripts should check for MDS availability before proceeding.

There should be a single MDS service per system. Conventions for locating MDS application information must be defined on a per-platform basis. The MDS application attributes that should be available to applications include:

- MDS Version
- MDS Installation location
- MDS Manifest
- Database name(s)
- Database location(s)
- Relation identifiers.

CDSA-related installation programs use the MDS registry information to discover if the appropriate MDS is available on the system. It may be necessary to upgrade MDS binaries or update MDS schema before CSSM, EMM, or service provider modules can be installed. For this reason, a CDSA installation package must include MDS installation programs.

The MDS installation program creates the databases and relations defined in this specification.

Elective module manager (EMM) installation programs may contain MDS installation programs that update the MDS schema to accommodate EMM service providers.

16.2 General Access Control over MDS Databases

Update access to MDS data may require satisfying an access control policy defined by the MDS provider. The mechanisms for enforcing privileged access include:

- Privileged application
- File permissions
- Administrative ACL

16.2.1 Privileged Application

Administrative tools that contain access permissions information in their manifest constitute privileged applications. If MDS interfaces are not statically bound to the administrative application then the MDS library must authenticate the manifest of the privileged application. Static binding removes the need for bilateral authentication of an administration application with an MDS library and is the desired approach. This technique should be combined with other access mechanisms to extend access privilege semantics to file system access control mechanisms.

16.2.2 File Permissions

File permissions can be established at database creation time. Default file permissions and file owners are set explicitly by the MDS provider. The file permissions are enforced at database open. The database access flags supplied with the *DbOpen* call are mapped to file system permission bits

Mapping the DL database access flags to file permissions is platform-specific. If the host operating system supports User, Group, Other privileges controlling Read, Write, and Execute permission bits, the following table suggests how the DL access flags, of type CSSM_DB_ACCESS_TYPE, could be mapped on a UNIX* platform.

DL Access Flags	Permission Bits	Process Privilege
CSSM_DB_ACCESS_READ	r--	other
CSSM_DB_ACCESS_WRITE	rw-	group
CSSM_DB_ACCESS_PRIVILEGED	rw-	user
CSSM_DB_ACCESS_ASYNCHRONOUS	N/A	group

Read-only access is granted to all processes. This enables the *DbQuery* interfaces. Read/Write access it granted to installation programs and services handling dynamic service provider insert and remove events. Write access is enabled for *DbInsert*, *DbUpdate* and *DbDelete*. An administration application that updates the MDS schema or reorganizes the database must own the database files to obtain privileged access. This, along with read/write privileges, allows exclusive access to the MDS database. When opened with CSSM_DB_ACCESS_PRIVILEGED, no other processes may open the database. In order to make changes to the MDS schema, the database needs to be opened with CSSM_DB_ACCESS_PRIVILEGED; in this case, the user/application has also the rights to query and insert/delete/update records in any user relation.

A privileged user can set use of CSSM_DB_ACCESS_ASYNCHRONOUS. It is MDS administration policy that determines the degree of perceived risk due to cached write operations. By default CSSM_DB_ACCESS_ASYNCHRONOUS is enabled.

16.2.3 Administrator ACLs

An additional degree of protection may be achieved through use of ACLs. The CSSM_RESOURCE_CONTROL_CONTEXT structure may be used in conjunction with CSSM_DB_ACCESS_WRITE and/or CSSM_DB_ACCESS_PRIVILEGED to control which entities are permitted to update MDS databases.

Technical Standard

Part 9:

Key Recovery (KR) Services

The Open Group

Chapter 17

Overview

17.1 Introduction

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.

17.2 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.

17.2.1 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.

17.2.2 Key Recovery Phases

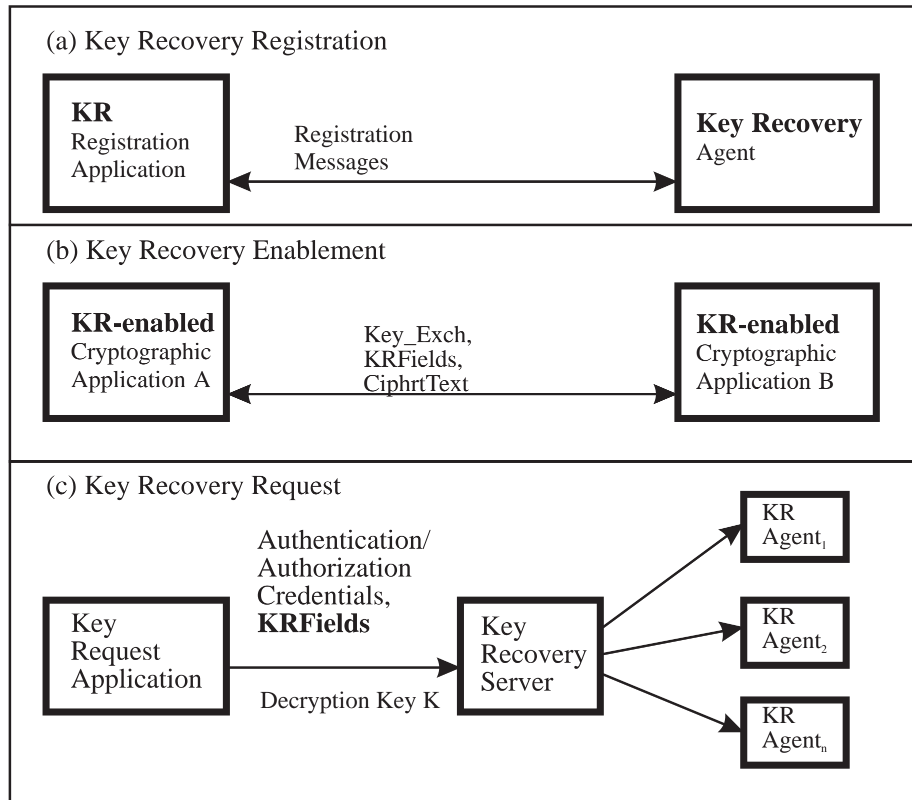


Figure 17-1 Key Recovery Phases

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 23-1 illustrates the three phases of key recovery. In Figure 23-1(a), a key recovery client registers with a recovery agent prior to engaging in cryptographic communication. In Figure 23-1(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 23-1(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.

The function calls that perform the *Registration*, *Enablement*, and *Request*, functions identified in Figure 23-1 on page 643 are summarized in Section 25.1 on page 655, and defined later in Chapter 25.

17.2.3 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.

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.

17.2.4 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.

17.3 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.

Key Recovery Enablement

18.1 Key Recovery in the CDSA

Figure 24-1 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).

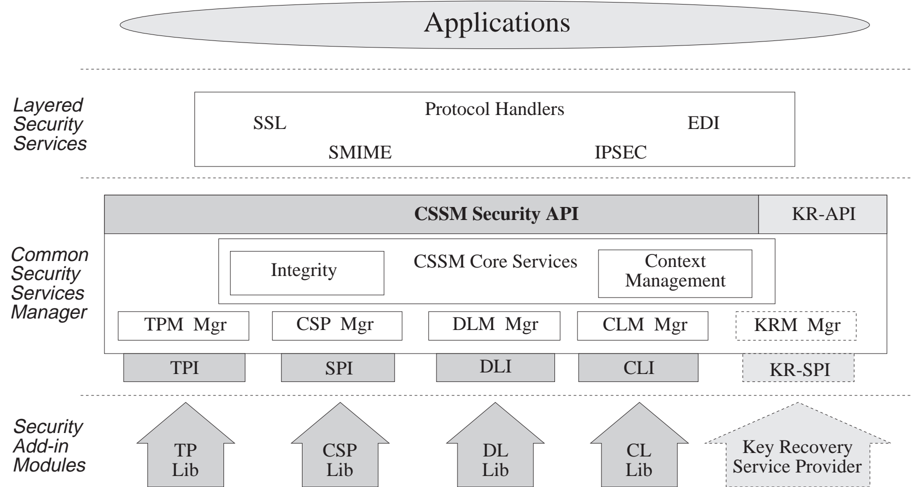


Figure 18-1 Elective Key Recovery Services in the CSSM

18.2 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

18.3 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.

18.4 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.

18.4.1 Operational Scenarios for Key Recovery

There are three basic operational scenarios for key recovery:

- Enterprise key recovery
- Law enforcement key recovery
- Individual key recovery

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

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).

18.4.2 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.

18.4.3 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.

18.4.4 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.

18.4.5 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.

18.4.6 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 19

Key Recovery Interfaces

19.1 Summary of Interface Calls

19.1.1 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* in **Part 2** of this Technical Standard. The applicable generic management functions include:

- *CSSM_ModuleLoad*( )
- *CSSM_ModuleUnload*( )
- *CSSM_ModuleAttach*( )
- *CSSM_ModuleDetach*( )

19.1.2 Key Recovery Module Management Operations

CSSM_KR_SetEnterpriseRecoveryPolicy( )
This call establishes the identity of the file that contains the enterprise key recovery policy function.

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 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.

19.1.3 Key Recovery Context Operations

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_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_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_KR_GetPolicyInfo( )
Returns the key recovery policy information pertaining to a given cryptographic context.

19.1.4 Key Recovery Registration Operations

CSSM_KR_RegistrationRequest( )
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 CSSM_KR_RegistrationRetrieve function.

CSSM_KR_RegistrationRetrieve( )
KRSP_RegistrationRetrieve( )
Completes a recovery registration operation. The result of the registration operation is returned in the form of a key recovery profile.

19.1.5 Key Recovery Enablement Operations

CSSM_KR_GenerateRecoveryFields( )
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 can be input to the encryption APIs in the cryptographic framework.

CSSM_KR_ProcessRecoveryFields( )
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.

19.1.6 Key Recovery Request Operations

CSSM_KR_RecoveryRequest( )
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 CSSM_KR_GetRecoveredObject function.

CSSM_KR_RecoveryRetrieve( )
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 CSSM_KRGetRecoveredObject function.

CSSM_KR_GetRecoveredObject( )
KRSP_GetRecoveredObject( )
Retrieves a single recovered key and its associated meta information.

CSSM_KR_RecoveryRequestAbort( )
KRSP_RecoveryRequestAbort( )
Terminates a recovery request operation and releases any state information associated with it.

19.1.7 Privileged Context Function

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.

19.1.8 Extensibility Function

CSSM_KR_PassThrough( )
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.

19.2 Example Application Using Key Recovery APIs

To understand the role of key recovery in encrypted data communication, consider the following scenario, illustrated in Figure 25-1. 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 *CSSM_CSP_CreateSymmetricContext*() to obtain a cryptographic context handle (HB1) representing the encryption key used by A.
- B invokes *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.

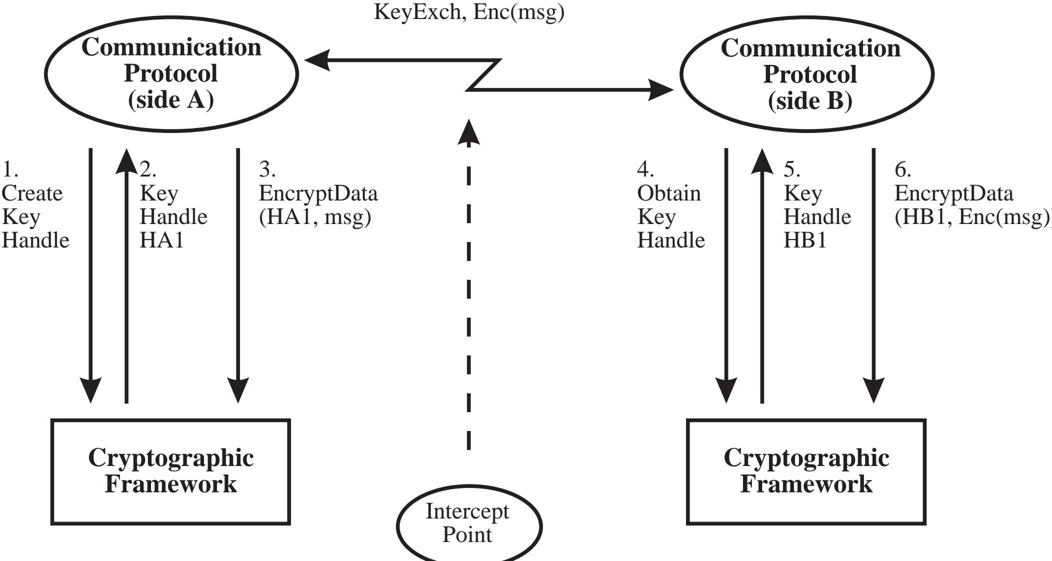


Figure 19-1 Encrypted Communications without Key Recovery

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 25-1 to take advantage of KR-API functions, as illustrated in Figure 25-2. 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 *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 *CSSM_EncryptData*( ). The encrypt API will only accept a separate cryptographic context handle generated by the CSSM.
- A invokes *CSSM_KR_CreateRecoveryEnablementContext*( ) and obtains a HA2, representing the local, and optionally the remote, key recovery profiles.
- Using handles HA1 and HA2, A invokes *CSSM_KR_GenerateRecoveryFields*( ) to update the cryptographic context referenced by handle HA1 for subsequent encryption operations based on the same context information. The *CSSM_KR_GenerateRecoveryFields*( ) call also generates a set of key recovery fields that are returned to A.
- A invokes *CSSM_EncryptData*( ) and provides as parameters the cryptographic context handle (HA1), and the message to be encrypted.
- A obtains the encrypted message and sends it to B, along with the key recovery fields. 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 *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.

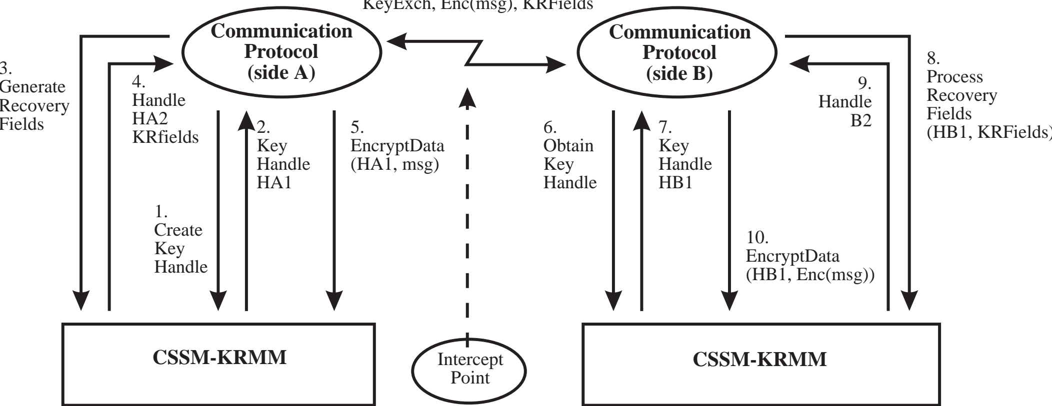


Figure 19-2 Encrypted Communications with Key Recovery Enablement

- In preparation, B invokes *CSSM_KR_CreateRecoveryEnablementContext*( ) to obtain the context handle HB2. This key recovery enablement context contains information used to process the key recovery fields in order to verify their integrity.
- B invokes *CSSM_KR_ProcessRecoveryFields*( ) and provides the two handles HB1 and HB2 as parameters If the recovery fields process correctly, the cryptographic context referenced by HB1 is updated and can be used in subsequent decrypt API calls to decrypt the message.
- B invokes *CSSM_DecryptData*( ) and provides the handle (HB1) 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.

19.3 Data Structures

19.3.1 CSSM_KRSP_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 */
```

19.3.2 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;
```

Definitions

Type
The type of the key recovery name space.

Length
The length of the name (in bytes).

Name
The name represented in a string.

19.3.3 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_CERTGROUP_PTR UserCertificate; /* public key certificate
                                           of the user */
    CSSM_CERTGROUP_PTR KRSCertChain;    /* cert chain for the
                                                 KRSP coordinator */
    uint8 LE_KRANum; /* number of KRA cert chains in the
                                                   following list */
    CSSM_CERTGROUP_PTR LE_KRACertChainList; /* list of Law
                             enforcement KRA certificate chains*/

    uint8 ENT_KRANum; /* number of KRA cert chains in the
                                                   following list */
    CSSM_CERTGROUP_PTR ENT_KRACertChainList; /* list of
                              Enterprise KRA certificate chains*/
    uint8 INDIV_KRANum; /* number of KRA cert chains in the
                                                   following list */
    CSSM_CERTGROUP_PTR INDIV_KRACertChainList; /* list of
```

```
                                      Individual KRA certificate chains*/

          CSSM_DATA_PTR INDIV_AuthenticationInfo; /* authentication
                             information for individual key recovery */

          uint32 KRSPFlags;  /* flag values to be interpreted by KRSP */

          CSSM_DATA_PTR KRSPExtensions; /* reserved for extensions
                                                  specific to KRSPs */

      } CSSM_KR_PROFILE, *CSSM_KR_PROFILE_PTR;
```

Definitions

UserName
The user's name.

UserCertificate
The user's certificate chain, used for identity and authentication when performing policy evaluation.

KRSCertChain
The certificate chain of Key Recovery Coordinator.

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.

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.

INDIV_AuthenticationInfo
Authentication information to be used for INDIV key recovery.

KRSPFlags
A bit mask specifying the user's selected service options specific to the selected key recovery service module.

KRSPExtensions
Reserved for future use.

19.3.4 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:

```
/* Attribute data type tags */
#define CSSM_ATTRIBUTE_DATA_KR_PROFILE 0x03000000

/* local entity profile */
CSSM_ATTRIBUTE_KRPROFILE_LOCAL
    = (CSSM_ATTRIBUTE_DATA_KR_PROFILE | 32),

/* remote entity profile */
CSSM_ATTRIBUTE_KRPROFILE_REMOTE
    = (CSSM_ATTRIBUTE_DATA_KR_PROFILE | 33),
```

19.3.5 CSSM_KR_POLICY_FLAGS

```
typedef uint32 CSSM_KR_POLICY_FLAGS;

#define CSSM_KR_INDIV           (0x00000001)
#define CSSM_KR_ENT             (0x00000002)
#define CSSM_KR_LE_MAN          (0x00000004)
#define CSSM_KR_LE_USE          (0x00000008)
#define CSSM_KR_LE              (CSSM_KR_LE_MAN|CSSM_KR_LE_USE)
#define CSSM_KR_OPTIMIZE        (0x00000010)
#define CSSM_KR_DROP_WORKFACTOR (0x00000020)
```

19.3.6 CSSM_KR_POLICY_LIST_ITEM

```
typedef struct cssm_kr_policy_list_item {
    struct kr_policy_list_item  *next;
    CSSM_ALGORITHMS AlgorithmId;
    CSSM_ENCRYPT_MODE Mode;
    uint32 MaxKeyLength;
    uint32 MaxRounds;
    uint8 WorkFactor;
    CSSM_KR_POLICY_FLAGS PolicyFlags;
    CSSM_CONTEXT_TYPE AlgClass;
} CSSM_KR_POLICY_LIST_ITEM, *CSSM_KR_POLICY_LIST_ITEM_PTR;
```

Definitions

AlgorithmID
The algorithm ID.

Mode
Encrypt mode. CSSM_ALGCLASS_SYMMETRIC and CSSM_ALGCLASS_ASYMMETRIC are valid options in this case.

Class
The class of the indicated algorithm.

MaxKeyLength
The maximum key length allowed for this encryption algorithm ID.

MaxRounds
Maximum number of encryption rounds.

WorkFactor
The maximum allowed workfactor value that may be used for law enforcement key recovery.

PolicyFlags
The mandatory policy flags.

AlgClass
CSSM_ALGCLASS_SYMMETRIC and CSSM_ALGCLASS_ASYMMETRIC are valid options in this case.

19.3.7 CSSM_KR_POLICY_INFO

```
typedef struct cssm_kr_policy_info {
    CSSM_BOOL krbNotAllowed;
    uint32 numberOfEntries;
    CSSM_KR_POLICY_LIST_ITEM *policyEntry;
} CSSM_KR_POLICY_INFO, *CSSM_POLICY_KR_INFO_PTR;
```

Definitions

krbNotAllowed
If CSSM_TRUE, generation of key recovery blocks is not allowed.

numberOfEntries
Number of entries in the *policyEntry* array.

policyEntry
A list of CSSM_KR_POLICY_LIST_ITEM, containing the encryption strength and other information for each encryption algorithm.

19.4 Key Recovery MDS Relation

Key Recovery service modules use the Module Directory Services (MDS) facility to store information about the module and it's services. This information is stored in MDS during installation of a KR service module. Applications use this information to identify and attach a service module that provides the required services. CSSM also uses the MDS KR records to locate object code and integrity credentials when loading a selected service provider module.

KR service modules store information in two MDS relations:

- Primary EMM Service Provider Relation
- KR Primary Relation.

CSSM uses the Primary EMM Service Provider Relation during module management operations. This relation is defined in Module Directory Services. Applications use the KR Primary Relation (see Section 19.21.0 on page 620) to locate and select specific KR modules.

19.4.1 Generic 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 **Part 2** of this Technical Standard.

The applicable generic management functions include:

- *CSSM_ModuleLoad*()
- *CSSM_ModuleUnload*()
- *CSSM_ModuleAttach*()
- *CSSM_ModuleDetach*()

19.5 Key Recovery Module Management Operations

The man-page definition for Key Recovery Module Management operations is presented in this section.

NAME

CSSM_KR_SetEnterpriseRecoveryPolicy

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_SetEnterpriseRecoveryPolicy
    (const CSSM_DATA *RecoveryPolicyFileName,
    const CSSM_CRYPTO_DATA *OldPassPhrase,
    const CSSM_CRYPTO_DATA *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 indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_FILENAME
Invalid policy file name.

CSSM_MEMORY_ERROR
Memory error.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_KR_INVALID_PASSWORD
Invalid password.

19.6 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_RETURN CSSMAPI CSSM_KR_CreateRecoveryRegistrationContext
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE *NewContext)
```

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 KR SPI that is to be used.

NewContext (output) A handle to the key recovery registration context. This value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_MEMORY_ERROR
Memory error.

SEE ALSO

CSSM_KR_RegistrationRequest()

NAME

CSSM_KR_CreateRecoveryEnablementContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_CreateRecoveryEnablementContext
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_KR_PROFILE *LocalProfile,
    const CSSM_KR_PROFILE *RemoteProfile,
    CSSM_CC_HANDLE *NewContext)
```

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 KR SPI that is to be used.

LocalProfile (input)
The key recovery profile for the local client.

RemoteProfile (input/optional)
The key recovery profile for the remote client.

NewContext (output)
A handle to the key recovery enablement context. This value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_GenerateRecoveryFields( )
CSSM_KR_ProcessRecoveryFields( )

NAME

CSSM_KR_CreateRecoveryRequestContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_CreateRecoveryRequestContext
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_KR_PROFILE *LocalProfile,
    CSSM_CC_HANDLE *NewContext)
```

DESCRIPTION

This call creates a key recovery request context based on a KR SPI 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 KR SPI 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.

NewContext (output)
A handle to the key recovery context. This value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_KR_INVALID_PROFILE
Invalid profile.

CSSM_MEMORY_ERROR
Memory error.

SEE ALSO

CSSM_KR_RecoveryRequest()
CSSM_KR_RecoveryRetrieve()

NAME

CSSM_KR_GetPolicyInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_GetPolicyInfo
    (CSSM_CC_HANDLE CCHandle,
    CSSM_KR_POLICY_FLAGS *EncryptionProhibited,
    uint32 *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.

EncryptionProhibited (output)
The usability field for law enforcement and enterprise key recovery. Possible values are:
- KR_LE
  Signifies that law enforcement key recovery enablement needs to be done.
- KR_ENT
  Signifies that enterprise key recovery enablement is required.

WorkFactor (output)
The maximum permissible workfactor value that may be used for law enforcement key recovery.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_MEMORY_ERROR
Memory error.

19.7 Key Recovery Registration Operations

The man-page definitions for Key Recovery Registration operations are presented in this section.

NAME

CSSM_KR_RegistrationRequest, for the CSSM API
KRSP_RegistrationRequest, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRequest
    (CSSM_CC_HANDLE RecoveryRegistrationContext,
    const CSSM_DATA *KRInData,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    CSSM_KR_POLICY_FLAGS KRFlags,
    sint32 *EstimatedTime,
    CSSM_HANDLE_PTR ReferenceHandle)

SPI:
CSSM_RETURN CSSMKRI KRSP_RegistrationRequest
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KRRegistrationContextHandle,
    const CSSM_CONTEXT *KRRegistrationContext,
    CSSM_DATA *KRInData,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    CSSM_KR_POLICY_FLAGS KRFlags,
    sint32 *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 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 used to invoke the associated *CSSM_KR_RegistrationRetrieve*( ) (for the API) or *KRSP_RegistrationRetrieve*( ) (for the SPI) function, after the *EstimatedTime* in seconds.

API PARAMETERS

RecoveryRegistrationContext (input)
The handle to the key recovery registration context.

KRInData (input)
Input data for key recovery registration.

AccessCredentials (input/optional)
User access credentials, including certificates, samples (password, biometrics) and callback function and context pointers used to authenticate the caller.

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. When the local service provider module or the key recovery server cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceHandle (output)
A handle that references the outstanding registration request. This handle must be used to retrieve the registration result using the *CSSM_KR_RegistrationRetrieve*( ) function.

SPI 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)
As for the API.

AccessCredentials (input/optional)
As for the API.

KRFlags (input)
As for the API.

EstimatedTime (output)
The estimated time after which the CSSM_KR_RegistrationRetrieve call should be invoked to obtain registration results. When the local service provider module or the key recovery server cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceHandle (output)
As for the API, except use the *KRSP_RegistrationRetrieve*( ) function to retrieve the registration result.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid registration handle.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Memory error.

SEE ALSO

CSSM_KR_CreateRecoveryRegistrationContext( )
CSSM_KR_RecoveryRetrieve( )
KRSP_RecoveryRetrieve( )

NAME

CSSM_KR_RegistrationRetrieve, for the CSSM API
KRSP_RegistrationRetrieve, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    sint32 *EstimatedTime,
    CSSM_KR_PROFILE_PTR *KRProfile)

SPI:
CSSM_RETURN CSSMKRI KRSP_RegistrationRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE ReferenceHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCredential,
    sint32 *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.

It is possible that the key recovery registration process has not yet completed, in which case the returned *EstimatedTime* is the updated estimate for completion of the registration procedure.

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 (in the case of the SPI, using the same *ReferenceHandle*).

API 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.

AccessCredentials (input/optional)
User access credentials, including certificates, samples (password, biometrics) and callback function and context pointers used to authenticate the caller.

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. When the local service provider module or the key recovery server cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

KRProfile (output)
The key recovery profile that is filled in by the registration operation.

SPI PARAMETERS

All as for the API, plus as follows:

ReferenceHandle (input)
Indicates which outstanding recovery request is to be completed.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid reference handle.

CSSM_MEMORY_ERROR
Memory error.

SEE ALSO

CSSM_KR_CreateRecoveryRegistrationContext( )
CSSM_KR_RecoveryRequest( )
KRSP_RecoveryRequest( )

19.8 Key Recovery Enablement Operations

The man-page definitions for Key Recovery Enablement operations are presented in this section.

NAME

CSSM_KR_GenerateRecoveryFields, for the CSSM API
KRSP_GenerateRecoveryFields, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_GenerateRecoveryFields
    (CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    const CSSM_DATA *KRSPOptions,
    CSSM_KR_POLICY_FLAGS KRFlags,
    CSSM_DATA_PTR KRFields)

SPI:
CSSM_RETURN CSSMKRI KRSP_GenerateRecoveryFields
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KREnablementContextHandle,
    const CSSM_CONTEXT *KREnablementContext,
    CSSM_CC_HANDLE CryptoContextHandle,
    const CSSM_CONTEXT *CryptoContext,
    CSSM_DATA *KRSPOptions,
    CSSM_KR_POLICY_FLAGS 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.

API 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 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 an uninterpreted data blob.

SPI 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)
As for the API.

KRFields (output)
As for the API.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_CreateRecoveryEnablementContext( )
CSSM_KR_ProcessRecoveryFields( )
KRSP_ProcessRecoveryFields( )

NAME

CSSM_KR_ProcessRecoveryFields, for the CSSM API
KRSP_ProcessRecoveryFields, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_ProcessRecoveryFields
    (CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    const CSSM_DATA *KRSPOptions,
    CSSM_KR_POLICY_FLAGS KRFlags,
    const CSSM_DATA *KRFields)

SPI:
CSSM_RETURN CSSMKRI KRSP_ProcessRecoveryFields
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KREnablementContextHandle,
    const CSSM_CONTEXT *KREnablementContext,
    CSSM_CC_HANDLE CryptoContextHandle,
    const CSSM_CONTEXT *CryptoContext,
    CSSM_DATA_PTR KRSPOptions,
    CSSM_KR_POLICY_FLAGS KRFlags,
    CSSM_DATA_PTR KRFields)
```

API DESCRIPTION

This call processes a set of key recovery fields given the key recovery context, and the cryptographic context for the *decryption* operation. If the processing is successful, the cryptographic context is updated and can be used for subsequent *decrypt* API calls.

SPI 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.

API 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, in the form of a data blob.

SPI 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)
As for the API.

KRFields (input)
As for the API

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_CreateRecoveryEnablementContext( )
CSSM_KR_GenerateRecoveryFields( )
KRSP_GenerateRecoveryFields( )

19.9 Key Recovery Request Operations

The man-page definitions for Key Recovery Request operations are presented in this section.

NAME

CSSM_KR_RecoveryRequest, for the CSSM API
KRSP_RecoveryRequest, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequest
    (CSSM_CC_HANDLE RecoveryRequestContext,
    const CSSM_DATA *KRInData,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    sint32 *EstimatedTime,
    CSSM_HANDLE_PTR ReferenceHandle)

SPI:
CSSM_RETURN CSSMKRI KRSP_RecoveryRequest
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KRRequestContextHandle,
    const CSSM_CONTEXT *KRRequestContext,
    const CSSM_DATA *KRInData,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    sint32 *EstimatedTime,
    CSSM_HANDLE_PTR ReferenceHandle)
```

DESCRIPTION

This function initiates a key recovery request operation. The *RequestContext* 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 associated CSSM_KR_ or KRSP_ *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 calling the *retrieve* function.

API 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.

AccessCredentials (input/optional)
User access credentials, including certificates, samples (password, biometrics) and callback function and context pointers used to authenticate the caller.

EstimatedTime (output)
The estimated time after which the caller should invoke the associated *CSSM_KR_RecoveryRetrieve*( ) function to obtain a cache of recovered keys. When the local service provider module or the key recovery server cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceHandle (output)
Handle representing this outstanding recovery request. This handle should be used as input to the *CSSM_KR_RecoveryRetrieve*( ) function.

SPI PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform upcalls 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)
As for the API

AccessCredentials (input/optional)
As for the API.

EstimatedTime (output)
As for the API except that the relevant follow-up call is *KRSP_RecoveryRetrieve*( ).

ReferenceHandle (output)
As for the API except this handle may be used to invoke the *KRSP_RecoveryRetrieve*( ) function.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_CreateRecoveryRequestContext( )
CSSM_KR_RecoveryRetrieve( )
KRSP_RecoveryRetrieve( )

NAME

CSSM_KR_RecoveryRetrieve, for the CSSM API
KRSP_RecoveryRetrieve, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials.
    CSSM_HANDLE ReferenceHandle,
    sint32 *EstimatedTime,
    CSSM_HANDLE_PTR CacheHandle,
    uint32 *NumberOfRecoveredKeys)

SPI:
CSSM_RETURN CSSMKRI KRSP_RecoveryRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_ACCESS_CREDENTIALS *AccessCredentials,
    sint32 *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 associated CSSM_KR_ or 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*.

API PARAMETERS

KRSPHandle (input)
The handle of the KR module for this operation.

AccessCredentials (input/optional)
User access credentials, including certificates, samples (password, biometrics) and callback function and context pointers used to authenticate the caller.

ReferenceHandle (input)
Indicates which outstanding recovery request is to be completed.

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. When the local service provider module or the key recovery server cannot estimate the time required to perform the requested service, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

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 associated *CSSM_KR_GetRecoveredObject*( ) (API) or *KRSP_GetRecoveredObject*( ) (SPI) function to complete the recovery of an individual key.

NumberOfRecoveredKeys (output)
The number of recovered key objects in the cache.

SPI PARAMETERS

As for the API, except that there is no *ReferenceHandle* parameter.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_CreateRecoveryRequestContext( )
CSSM_KR_RecoveryRequest( )
CSSM_KR_GetRecoveredObject( )
CSSM_KR_RecoveryRequestAbort( )

KRSP_CreateRecoveryRequestContext( )
KRSP_RecoveryRequest( )
KRSP_GetRecoveredObject( )
KRSP_RecoveryRequestAbort( )

NAME

CSSM_KR_GetRecoveredObject, for the CSMM API
KRSP_GetRecoveredObject, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_GetRecoveredObject
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle,
    uint32 IndexInResults,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    uint32 Flags,
    CSSM_KEY_PTR RecoveredKey,
    CSSM_DATA_PTR OtherInfo)

SPI:
CSSM_RETURN CSSMKRI KRSP_GetRecoveredObject
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle,
    uint32 IndexInResults,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_RESOURCE_CONTROL_CONTEXT *CredAndAclEntry,
    uint32 Flags,
    CSSM_KEY_PTR RecoveredKey,
    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*( ) (for the CSSM API) or *KRSP_RecoveryRetrieve*( ) (for the KR SPI) 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 of the KR module to perform this operation.

CacheHandle (input)
The handle returned from a successful *CSSM_KR_RecoveryRequest*( ) (for the CSSM API) or *KRSP_RecoveryRequest*( ) (for the KR SPI) operation.

IndexInResults (input)
The index into the results that are referenced by the *CacheHandle* parameter. The *IndexInResults* ranges from 0 to (NumberOfRecoveredKeys-1), if *NumberOfRecoveredKeys* is 1

or larger. *NumberOfRecoveredKeys* is returned by a successful call to the associated CSSM_KR_ or KRSP_ *RecoveryRetrieve*( ).

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. It may be set to NULL if the key is to be returned in raw form to the caller.

CredAndAclEntry (input/optional)
: A structure containing one or more credentials authorized for creating a key and the prototype ACL entry that will control future use of the newly created key. The credentials and ACL entry prototype can be presented as immediate values, or callback functions can be provided for use by the CSP to acquire the credentials and/or the ACL entry interactively. If the CSP provides public access for creating a key, then the credentials can be NULL. If the CSP defines a default initial ACL entry for the new key, then the ACL entry prototype can be an empty list.

Flags (input)
: Flag values relevant for recovery of a key. Possible value is CERT_RETRIEVE — if the recovered key is a private key, return the corresponding public key certificate in the *OtherInfo* parameter.

RecoveredKey (output)
: This parameter is used when recovering a symmetric key. The recovered key is stored in the key structure provided by the caller.

OtherInfo (output/optional)
: This parameter is used if there is additional information associated with the recovered key (such as the public key certificate when recovering a private key) that is to be returned. The object is opaque, and the caller must have knowledge of the expected structure of this result.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

SEE ALSO

CSSM_KR_CreateRecoveryRequestContext( )
CSSM_KR_RecoveryRequest( )
CSSM_KR_RecoveryRetrieve( )
CSSM_KR_RecoveryRequestAbort( )

KRSP_CreateRecoveryRequestContext( )
KRSP_RecoveryRequest( )
KRSP_RecoveryRetrieve( )
KRSP_RecoveryRequestAbort( )

NAME

CSSM_KR_RecoveryRequestAbort, for the CSSM API
KRSP_RecoveryRequestAbort, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequestAbort
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle)

SPI:
CSSM_RETURN CSSMKRI KRSP_RecoveryRequestAbort
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function is invoked after a successful call to the associated *CSSM_KR_RecoveryRetrieve*( ) (for the CSSM API) or *KRSP_RecoveryRetrieve*( ) (for the KR SPI) function, and after all desired keys were recovered using the associated CSMM_KR_ or KRSP_ *GetRecoveredObject*( ) function. 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*( ) (for the CSSM API) or *KRSP_RecoveryRetrieve*( ) (for the KR SPI) operation.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_HANDLE
Invalid cache handle.

SEE ALSO

CSSM_KR_CreateRecoveryRequestContext( )
CSSM_KR_RecoveryRequest( )
CSSM_KR_RecoveryRetrieve( )
CSSM_KR_GetRecoveredObject( )

KRSP_CreateRecoveryRequestContext( )
KRSP_RecoveryRequest( )
KRSP_RecoveryRetrieve( )
KRSP_GetRecoveredObject( )

NAME

CSSM_KR_QueryPolicyInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_QueryPolicyInfo
    (CSSM_ALGORITHMS AlgorithmId,
    CSSM_ENCRYPT_MODE Mode,
    CSSM_CONTEXT_TYPE Class,
    CSSM_KR_POLICY_INFO_PTR *PolicyInfoData)
```

DESCRIPTION

This function queries the law enforcement CSSM policy in effect and returns relevant information for application use. No privilege is required to invoke this function.

The policy information reports the maximum key length that can be generated, per cipher algorithm type and mode, without a need to generate key recovery blocks. It also specifies whether it is the jurisdiction of manufacturing or the jurisdiction of use to enforce the given policy. For special situations where the jurisdiction of use prohibits generation of key recovery fields, that information will also be provided.

Applications can request policy information relative to a specific algorithm, by providing the CSSM algorithm identifier in first parameter to the call. If a CSSM_ALGID_NONE is provided in this field, the *PolicyInfoData* will contain information pertaining to the entire set of algorithms controlled for the law enforcement jurisdiction. The *Mode* parameter can be specified exactly, or set to CSSM_ALGMODE_NONE. In the latter case, all matching algorithm ids are returned, regardless of the actual mode. The class parameter should be set to correctly to symmetric or asymmetric, otherwise the results will not be accurate.

If the API can not find a matching entry in the configured policies, the *numberOfEntries* field in *PolicyInfoData* is set to 0, and the return code to CSSM_OK.

Applications have the responsibility to free the memory associated with the policy information data when no longer needed, by calling *CSSM_KR_FreePolicyInfo*().

PARAMETERS

AlgorithmID (input)
CSSM defined algorithm identifier for which policy information is requested. This parameter must be CSSM_ALGID_NONE if global policy information is desired.

Mode (input)
The desired algorithm mode. This parameter can be set to CSSM_ALGMODE_NONE to get all applicable modes.

Class (input)
The class of the desired algorithm. The allowed values are CSSM_ALGCLASS_ASYMMETRIC and CSSM_ALGCLASS_SYMMETRIC.

PolicyInfoData (output)
Pointer to a CSSM policy information data structure to receive the query results.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid KR handle

CSSM_KR_INVALID_MODE
Invalid or unsupported algorithm mode

CSSM_KR_INVALID_CLASS
Invalid or unsupported class of algorithm

CSSM_KR_INVALID_POINTER
Invalid pointer for **policyinfo** structure

NAME

CSSM_KR_FreePolicyInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_FreePolicyInfo
    (CSSM_KR_POLICY_INFO_PTR PolicyInfoData)
```

DESCRIPTION

This function frees the memory allocated in the *PolicyInfoData* by a successful call to the *CSSM_KR_QueryPolicyInfo*( ) API.

PARAMETERS

PolicyInfoData (input)
The pointer to the memory that stores the policy data and was allocated by a successful call to the *CSSM_KR_QueryPolicyInfo*( ) API.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_KR_INVALID_POINTER
Invalid input data

SEE

CSSM_KR_QueryPolicyInfo( )

19.10 Privileged Context Operation

The man-page definition for the KR SPI Privileged Context operation is presented in this section.

NAME

KRSP_PassPrivFunc

SYNOPSIS

```
CSSM_RETURN CSSMKRI KRSP_PassPrivFunc
    (CSSM_PRIV_FUNC_PTR CSSM_SetContextPriv)
```

DESCRIPTION

This function is used to provide the KR SPI 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 KR SPI to make a cryptographic context privileged with respect to key recovery policy enforcement decisions, so that the KR SPI itself is allowed to bypass the key recovery policy controls. The KR SPI 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 KR SPI to set or drop the privilege state flag for a given cryptographic context.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

CSSM_MEMORY_ERROR
Not enough memory.

19.11 Extensibility Function

The man-page definition for Extensibility functionality is presented in this section.

NAME

CSSM_KR_PassThrough, for the CSSM API br KRSP_PassThrough, for the KR SPI

SYNOPSIS

```
API:
CSSM_RETURN CSSMAPI CSSM_KR_PassThrough
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)

SPI:
CSSM_RETURN CSSMKRI KRSP_PassThrough
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KeyRecoveryContextHandle,
    const CSSM_CONTEXT *KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContextHandle,
    const CSSM_CONTEXT *CryptoContext,
    uint32 PassThroughId,
    const void *InputParams,
    void **OutputParams)
```

DESCRIPTION

The *KRSP_PassThrough*() function is provided to allow KR SPI 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 KR SPI can be assigned at the discretion of the KR SPI module developer. However, its parameter list and return value must match what is shown below.

For the CSSM API, 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.

The error codes given in this definition constitute the generic error codes which may be used by all KR SPIs to describe common error conditions. KR SPI developers may also define their own module-specific error codes, as described in **Part 9** — *CSSM Add-in Module Structure and Administration Specification* .

API 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 specifying the exported/custom function to perform.

InputParams (input)
A pointer to the 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 data pointers.

OutputParams (output)
A pointer to the data structure containing the output from the *PassThrough* function. The output data must be interpreted by the calling application based on externally available information.

SPI PARAMETERS

KRHandle (input)
As for the API.

KeyRecoveryContextHandle (input/optional)
The handle that describes the add-in key recovery service provider module used to perform upcalls to CSSM for the memory functions managed by CSSM.

KeyRecoveryContext (input/optional)
As for the API.

CryptoContextHandle (Input/optional)
The handle that describes the cryptographic context used to link to the CSP-managed information.

CryptoContext (input/optional)
As for the API.

PassThroughId (input)
As for the API.

InputParams (input)
As for the API.

OutputParams (output)
As for the API. The KR SPI will allocate the memory for this structure, and the application should free the memory for the structure.

RETURN VALUES

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

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.

Technical Standard

Part 10:

Embedded Integrity Services Library (EISL)

The Open Group

Chapter 20

Introduction

20.1 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?"

20.2 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.

20.3 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 the signature of the containing module. The public key used for verification is embedded in the containing module 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.

20.4 A Phased Approach

The establishment of integrity between two dynamically loaded, executable objects proceeds in three phases:

- Self-check
- Bilateral authentication
- 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 dynamically loaded, executable objects.

20.4.1 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.

20.4.2 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.

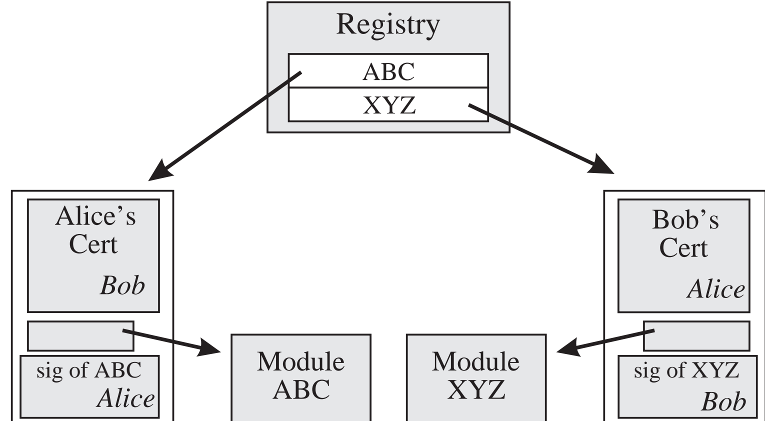


Figure 20-1 Bilateral Authentication Using Software Credentials

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.

20.4.3 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.

20.5 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.

20.5.1 Location of Modules and Credentials

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 **Signed Manifest Specification** for the definition 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.

20.5.2 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.

20.5.3 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.

20.5.4 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 the format, creation, and use of these credentials can be found in:

- **Signed Manifest** specification
- **Add-in Module Structure and Administration** specification

For convenience, a brief introduction 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.

Creating a signed manifest is one of the last steps in manufacturing a CDSA component. The manufacturing process creates at least three separate files for the manifest sections, signer information, and signature block. The files are zipped to create a single credential file in PKZIP format. Each file has an identifying suffix:

- The zipped credential filename suffix is .esw
- The manifest filename suffix is .mf
- The signer information filename suffix is .sf
- The signature block filename suffix identifies the signature type and is one of the following:

  .rsa (PKCS7 signature, MD5 + RSA)

  .dsa (PKCS7 signature, DSA)

  .pgp (Pretty Good Privacy Signature)

When providing credentials as input to an EISL function, two methods are supported. The credentials can be referenced by providing:

- A fully-qualified file system pathname locating the .esw file
- A pointer to the memory-resident copy of the .esw file.

Optionally a manifest section in the credential can include a direct reference to the object code file whose integrity hash is stored in that manifest section. EISL provides functions supporting the use of these direct file references. If a manifest section does not directly reference an object code file, to verify the integrity of an object code module, a caller must use equivalent EISL functions that accept a fully-qualified file system pathname locating the object code module.

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.

20.6 Use of Other Standards or Specifications

This EISL specification uses other industry specifications or standards for certificates, keys, signatures, and cryptographic algorithms. These 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 21

Data Structures

21.1 Object Pointers

Many of the ISL 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 ISL object types is provided at the end of this section.

21.1.1 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;
```

21.1.2 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 *EISL_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;
```

21.1.3 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;
```

21.1.4 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;
```

21.1.5 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;
```

21.1.6 Verified Module Object

A verified module object is returned as a result of verifying the credentials for a module. This object is created by any of the following functions:

EISL_SelfCheck()
EISL_VerifyLoadedModule()
EISL_VerifyLoadedModuleAndCredentials()
EISL_VerifyLoadedModuleAndCredentialsWithCertificate()
EISL_VerifyLoadedModuleAndCredentialData()
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate()
EISL_VerifyAndLoadModule()
EISL_VerifyAndLoadModuleAndCredentials()
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate()
EISL_VerifyAndLoadModuleAndCredentialData()
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate()

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;
```

21.1.7 EISL Object Relationships and Life Cycle

This is shown in the following table.

OBJECT	CONTAINING OBJECT	CREATING FUNCTION(S)	RECYCLING FUNCTION
Verified Signature Root	none	*EISL_CreateVerifiedSignatureRoot* *EISL_CreateVerifiedSignatureRootWithCertificate* *EISL_CreateVerifiedSignatureRootWithCredentialData* *EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate*	*EISL_RecycleVerifiedSignatureRoot*
Verified Module*	Verified Signature Root	*EISL_SelfCheck* *EISL_VerifyLoadedModule* *EISL_VerifyAndLoadModuleAndCredentials* *EISL_VerifyAndLoadModuleAndCredentialsWithCertificate* *EISL_VerifyAndLoadModuleAndCredentialData* *EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate* *EISL_VerifyAndLoadModule* *EISL_VerifyLoadedModuleAndCredentials* *EISL_VerifyLoadedModuleAndCredentialsWithCertificate* *EISL_VerifyLoadedModuleAndCredentialData* *EISL_VerifyLoadedModuleAndCredentialDataWithCertificate* *EISL_DuplicateVerifiedModulePtr*	*ISL_RecycleModuleAndCredentials**
Manifest Section	Verified Signature Root	*(implicit)*	*(implicit)*
Verified Module	Manifest Section	*(implicit)*	*(implicit)*
Verified Certificate	none	*EISL_CreateCertificateChain* *EISL_CreateCertificateChainWithCertificate* *EISL_CreateCertificateChainWithCredentialData* *EISL_CreateCertificateChainWithCredentialDataAnd Certificate*	*EISL_RecycleCertificateChain*
Verified Certificate Chain***	Verified Signature Root	*(implicit)*	*(implicit)*
Verified Certificate	Verified Certificate Chain	*(implicit)*	*(implicit)*
Signature Root Attribuite Iterator	Verified Signature Root	*EISL_CreateManifestAttributeEnumerator*	*EISL_RecycleAttributeEnumerator*
Manifest Section Iterator	Verified Signature Root	*EISL_CreateManifestSectionEnumerator*	*EISL_RecycleManifestSectionEnumerator ** *
Signature Attribute Iterator	Verified Signature Root	*EISL_Create Signature AttributeEnumerator*	*EISL_RecycleSignatureAttributeEnumerator ** *
Signer Info Attribute Iterator	Verified Signature Root	*EISL_CreateSignerInfoAttributeEnumerator*	*EISL_RecycleAttributeEnumerator*
Certificate Attribute Iterator	Verified Certificate	*EISL_CreateCertificateAttributeEnumerator*	*EISL_RecycleCertificateAttributeEnumerator ** *
Manifest Section Attribute Iterator	Verified Signature Root	*EISL_CreateManifestSection*	*EISL_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.

21.2 Types and Data Structure

21.2.1 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.

21.2.2 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_const_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.

21.2.3 ISL_STATUS

```
typedef enum isl_status{
    ISL_OK = 0,
    ISL_FAIL = -1
} ISL_STATUS;
```

21.2.4 ISL_FUNCTION_PTR

The ISL_FUNCTION_POINTER defines a pointer to an ISL function. This type is returned by several functions that locate ISL services.

```
typedef void (*ISL_FUNCTION_PTR)(void);
```

Chapter 22

EISL Functions

22.1 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

EISL_SelfCheck

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_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 either embedded in the containing module 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 *EISL_CheckAddressWithinModule*( ) function.

PARAMETERS

None.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

EISL_CheckAddressWithinModule( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyAndLoadModuleAndCredentialData

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_VerifyAndLoadModuleAndCredentialData
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const 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 *credentialImage*. The *ModuleSearchPath* parameter specifies a fully-qualified file system path name to locate the target object code module. If the signature has more than one signer, the *Signer* parameter selects the signer to be verified.

- Verify module integrity
  If the *CredentialImage* verifies, the integrity of the object code module referenced by the manifest section with the specified *Name* is verified. If verification is successful, a verified module object pointer is returned. Otherwise, NULL is returned.

If the object module referenced by *ModuleSearchPath* 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 verification is successful, the module is loaded.

If the module is already loaded, it is verified in memory.

The *CredentialImage* contains a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

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 *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyAndLoadModule*( ) provides the same functionality.

PARAMETERS

CredentialsImage (input)
: A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
: A string containing the fully-qualified path name to locate the object code associated with the signed manifest credentials.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest of the object code referenced by *ModueSearchPath*.

Signer (input/optional)
: The signer information (as a key for directly signed objects) or issuer name (as a certificate

for objects signed by the key associated with a certificates) of the entity whose signature is to be verified. 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/optional)
This is the public key of the signer or trusted 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

EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyAndLoadModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR
  EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const ISL_CONST_DATA Certificate)
```

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 *Certificate* contains the public key that is used to verify the signature on the specified *CredentialsImage*. The *ModuleSearchPath* parameter specifies a fully-qualified file system path name to locate the target object code module. If the signature has more than one signer, the *Signer* parameter selects the signer to be verified.
- Verify module integrity
  If the *CredentialImage* verifies, the integrity of the object code module referenced by the manifest section with the specified *Name* is verified. If verification is successful, a verified module object pointer is returned. Otherwise, NULL is returned.

If the object module referenced by *ModuleSearchPath* 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 verification is successful, the module is loaded.

If the module is already loaded, it is verified in memory.

The *CredentialImage* contains a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

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 *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyAndLoadModule*( ) provides the same functionality.

PARAMETERS

CredentialsImage (input)
: A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
: A string containing the fully-qualified path name to locate the object code associated with the signed manifest credentials.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest of the object code referenced by *ModueSearchPath*.

Signer (input/optional)
The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is to be verified. 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.

Certificate (input/optional)
This is a certificate containing the public key of the signer or trusted root certificate authority. If the *Certificate* is NULL, a default public key value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialWithCertificate( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyAndLoadModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyAndLoadModuleAndCredentials

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_VerifyAndLoadModuleAndCredentials
    (ISL_CONST_DATA Credentials,
    ISL_CONST_DATA Name,
    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 *Name* 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 *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyAndLoadModule*( ) provides the same functionality.

Cleanup is done by *EISL_RecycleVerifiedModuleCredentials.*( )

PARAMETERS

Credentials (input)
The full file name to the signature file.

Name (input)
The ame of the manifest section that refers to the object code to be verified.

Signer (input/optional)
The signer information (for directly signed signatures) or issuer name (if signed by certificates). If *Signer.Data* 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/optional)
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 *PublicKey.Data* 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

EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyAndLoadModule( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_RecycleVerifiedModuleCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )

NAME

EISL_VerifyAndLoadModuleAndCredentialsWithCertificate

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR
  EISL_VerifyAndLoadModuleAndCredentialsWithCertificate
    (const ISL_CONST_DATA Credentials,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const ISL_CONST_DATA Certificate)
```

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 *Certificate* contains the public key that is used to verify the signature on the specified *Credentials* The *Credentials* parameter specifies a fully-qualified file system path name to locate the signed manifest associated with the target object code module. If the signature block in the signed manifest contains more than one signature, the *Signer* parameter selects the signer to be verified.

- Verify module integrity
  If the credentials are valid, the *Name* identifies the manifest section containing a fully-qualified file system path name to locate the object code module. If integrity verification of the object module is 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.

The *Credentials* contain a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

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 *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyAndLoadModule*( ) provides the same functionality.

PARAMETERS

Credentials (input)
: A string containing the fully-qualified path name to locate the signed manifest credentials associated with the object code.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest and a fully-qualified file

Signer (input/optional)
: The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is

to be verified. If *Signer.Data* 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.

Certificate (input/optional)
This is a certificate containing the public key of the signer or trusted root certificate authority. If *Certificate.Data* is NULL, a default public key value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyAndLoadModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyLoadedModuleAndCredentialData

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_VerifyLoadedModuleAndCredentialData
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const 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 *CredentialsImage*. The *CredentialsImage* is a memory-resident signed manifest credential associated with the target object code. If the signature block in the signed manifest contains 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 *Name* is verified. The *ModuleSearchPath* is a fully-qualifies file system name locating the object code in the file system. The object code file name is used to locate the loaded object code and verification is performed on the loaded object code. If successful, a verified module object pointer is returned. Otherwise, NULL is returned.

The contains a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

This function combines many smaller functions into one call for a common case. If greater flexibility is needed, a series of calls that includes *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyLoadedModule*( ) provides the same functionality.

PARAMETERS

CredentialsImage (input)
: A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
: A string containing the fully-qualified path name to locate the object code associated with the signed manifest credentials.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest of the object code referenced by *ModuleSearchPath*.

Signer (input/optional)
: The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is to be verified. If *Signer.Data* 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/optional)
This is the public key of the signer or trusted 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

EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyLoadedModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyLoadedModuleAndCredentialDataWithCertificate

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const ISL_CONST_DATA Certificate)
```

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 *Certificate* contains the public key that is used to verify the signature on the specified *CredentialsImage*. The *CredentialsImage* is a memory-resident signed manifest credential associated with the target object code. If the signature block in the signed manifest contains 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 *Name* is verified. The *ModuleSearchPath* is a fully-qualifies file system name locating the object code in the file system. The object code file name is used to locate the loaded object code and verification is performed on the loaded object code. If successful, a verified module object pointer is returned. Otherwise, NULL is returned.

The *CredentialsImage* contains a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

This function combines many smaller functions into one call for a common case. If greater flexibility is needed, a series of calls that includes *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyLoadedModule*( ) provides the same functionality.

PARAMETERS

CredentialsImage (input)
: A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
: A string containing the fully-qualified path name to locate the object code associated with the signed manifest credentials.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest of the object code referenced by *ModueSearchPath*.

Signer (input/optional)
: The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is

to be verified. If *Signer.Data* 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.

Certificate (input/optional)
This is a certificate containing the public key of the signer or trusted root certificate authority. If *Certificate.Data* is NULL, a default public key value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyLoadedModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_VerifyLoadedModuleAndCredentials

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_VerifyLoadedModuleAndCredentials
    (ISL_CONST_DATA Credentials,
    ISL_CONST_DATA Name,
    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 Name 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 *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyLoadedModule*( ) provides the same functionality. Cleanup is done by *EISL_RecycleVerifiedModuleCredentials*( ).

PARAMETERS

Credentials (input)
The full file name to the signature file.

Name (input)
The name of the manifest section that refers to the object code to be verified.

Signer (input/optional)
The signer information (for directly signed signatures) or issuer name (if signed by certificates). If *Signer.Data* is NULL, a default value is assumed.

PublicKey (input/optional)
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 *PublicKey.Data* 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

EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyLoadedModule( )
EISL_CreateVerifiedSignatureRoot( )

EISL_RecycleVerifiedModuleCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

NAME

EISL_VerifyLoadedModuleAndCredentialsWithCertificate

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR
EISL_VerifyLoadedModuleAndCredentialsWithCertificate
    (const ISL_CONST_DATA Credentials,
    const ISL_CONST_DATA Name,
    const ISL_CONST_DATA Signer,
    const ISL_CONST_DATA Certificate)
```

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 *Certificate* contains the public key that is used to verify the signature on the specified *Credentials*. The *Credentials* parameter specifies a fully-qualified file system path name to locate the signed manifest associated with the target object code module. If the signature block in the signed manifest contains more than one signature, the *Signer* parameter selects the signer to be verified.
- Verify module integrity
  If the credentials are valid, the *Name* identifies the manifest section containing a fully-qualified file system path name to locate the object code module. The object code file name is used to locate the loaded object code and verification is performed on the loaded object code. If integrity verification of the loaded object module is successful, a verified module pointer is returned. Otherwise, NULL is returned.

The *Credentials* contain a PKCS#7 signature block as well as free-standing X.509 certificates. These certificates can be used to form a certificate chain used in the verification process.

When the verification result is no longer needed, the returned verified object module reference can be freed using *EISL_RecycleVerifiedModuleCredentials*( ).

This function combines many smaller functions into one call for a common case. If greater flexibility is needed, a series of calls that includes *EISL_CreateCertificateChain*( ), *EISL_CopyCertificateChain*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), *EISL_FindManifestSection*( ), and *EISL_VerifyLoadedModule*( ) provides the same functionality.

PARAMETERS

Credentials (input)
: A string containing the fully-qualified path name to locate the signed manifest credentials associated with the object code.

Name (input)
: The name of the manifest section containing attributes including a cryptographic digest and a fully-qualified file system path name for the object code.

Signer (input/optional)
: The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is to be verified. If *Signer.Data* 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.

Certificate (input/optional)
This is a certificate containing the public key of the signer or trusted root certificate authority. If *Certificate.Data* is NULL, a default public key value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_CreateCertificateChain( )
EISL_FindManifestSection( )
EISL_CopyCertificateChain( )
EISL_VerifyLoadedModule( )
EISL_CreateVerifiedSignatureRoot( )
EISL_RecycleVerifiedModuleCredentials( )

NAME

EISL_GetCertficateChain

SYNOPSIS

```
ISL_VERIFIED_CERTIFICATE_CHAIN_PTR EISL_GetCertificateChain
    (ISL_VERIFIED_MODULE_PTR Module)
```

DESCRIPTION

This function returns a reference to the certificate chain that was constructed and verified by *EISL_SelfCheck*( ), *EISL_VerifyLoadedModuleAndCredentials*( ) or *EISL_VerifyAndLoadModuleAndCredentials*( ).

PARAMETERS

Module (input)

A verified module object returned by the *EISL_SelfCheck*( ), *EISL_VerifyLoadedModuleAndCredentials*( ), or *EISL_VerifyAndLoadModuleAndCredentials*( ) function.

Verified module objects created by *EISL_VerifyAndLoadModule*( ) and *EISL_VerifyLoadedModule*( ) return a NULL certificate chain.

RETURN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_SelfCheck( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

NAME

EISL_ContinueVerification

SYNOPSIS

```
uint32 EISL_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 *EISL_VerifyAndLoadModuleAndCredentials*(), *EISL_SelfCheck*(), *EISL_VerifyLoadedModuleAndCredentials*(), *EISL_VerifyAndLoadModule*(), or *EISL_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 any of the following functions:

EISL_SelfCheck()
EISL_VerifyLoadedModule()
EISL_VerifyLoadedModuleAndCredentials()
EISL_VerifyLoadedModuleAndCredentialsWithCertificate()
EISL_VerifyLoadedModuleAndCredentialData()
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate()
EISL_VerifyAndLoadModule()
EISL_VerifyAndLoadModuleAndCredentials()
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate()
EISL_VerifyAndLoadModuleAndCredentialData()
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate()
EISL_DuplicateVerifiedModulePtr()

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

EISL_RecycleVerifiedModuleCredentials()
EISL_SelfCheck()
EISL_VerifyLoadedModule()
EISL_VerifyAndLoadModule()
EISL_VerifyLoadedModuleAndCredentials()
EISL_VerifyAndLoadModuleAndCredentials()
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate()
EISL_VerifyAndLoadModuleAndCredentialData()

EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

EISL_DuplicateVerifiedModulePtr

NAME

EISL_DuplicateVerifiedModulePtr

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_DuplicateVerifiedModulePtr
    (ISL_VERIFIED_MODULE_PTR Module)
```

DESCRIPTION

This function clones the state information associated with an existing verified module pointer. If necessary a full copy is created, otherwise a reference count is incremented to indicate additional users of the object. The function returns a new verified module pointer referencing the cloned state information.

PARAMETERS

Module (input)

A verified module object to be duplicated. This can be returned by any of the following functions:

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )

RETURN VALUE

A pointer to the verified module state is returned if successful, otherwise NULL.

SEE ALSO

EISL_RecycleVerifiedModuleCredentials( )
EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

NAME

EISL_RecycleVerifiedModuleCredentials

SYNOPSIS

```
ISL_STATUS EISL_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 any of the following functions:

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

22.2 Signature Root Methods

The man-page definitions for Signature Root Methods are presented in this section.

NAME

EISL_CreateVerifiedSignatureRootWithCredentialData

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR
EISL_CreateVerifiedSignatureRootWithCredentialData
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    const ISL_CONST_DATA Signer,
    const ISL_CONST_DATA PublicKey)
```

DESCRIPTION

This function uses the *PublicKey* to verify a digital signature contained in the *CredentialsImage*. It does not construct certificate chains, but must use the key directly. If the credentials support multiple signers, the *Signer* parameter specifies which signature to verify.

This function does not verify the objects referenced in the manifest sections of the *CredentialsImage*. However, the manifest sections are verified, and the attributes in the sections can be trusted.

The *ModuleSearchPath* is a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*. The *ModuleSearchPath* is stored as state information associated with the verified signature root returned by this function. The information is available to subsequent operations on the verified signature root.

The manifest sections can be enumerated using the object created by *EISL_CreateManifestSectionEnumerator*().

PARAMETERS

CredentialsImage (input)
A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
A string containing a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*.

Signer (input/optional)
The signer information (as a key for directly signed objects) or issuer name (as a certificate for objects signed by the key associated with a certificates) of the entity whose signature is to be verified. If *Signer.Data* 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/optional)
This is the public key of the signer or trusted root certificate authority. The representation for the key must be compatible with the format of public keys in the selected certificate format. If *PublicKey.Data* is NULL, a default value is assumed.

RETURN VALUE

Pointer to a verified signature root object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate( )
EISL_CreateVerifiedSignatureRoot( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_CreateManifestSectionEnumerator( )
EISL_CreateSignatureAttributeEnumerator( )

NAME

EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR
EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate
    (const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath,
    ISL_VERIFIED_CERTIFICATE_PTR Cert)
```

DESCRIPTION

This function uses the public key contained in the *Cert* to verify a digital signature contained in the *CredentialsImage*. The *Cert* must be a verified certificate. This function does not construct certificate chains, but must use the signer identification and public key contained in the certificate.

This function does not verify the objects referenced in the manifest sections of the *CredentialsImage*. However, the manifest sections are verified, and the attributes in the sections can be trusted.

The *ModuleSearchPath* is a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*. The *ModuleSearchPath* is stored as state information associated with the verified signature root returned by this function. The information is available to subsequent operations on the verified signature root.

The manifest sections can be enumerated using the object created by *EISL_CreateManifestSectionEnumerator*( ).

PARAMETERS

CredentialsImage (input)
: A pointer to the memory-resident signed manifest credentials to be verified by this function.

ModuleSearchPath (input)
: A string containing a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*.

Cert (input)
: This is a verified certificate containing the public key of the signer or trusted root certificate authority.

RETURN VALUE

Pointer to a verified signature root object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_CreateVerifiedSignatureRootWithCredentialData( )
EISL_CreateVerifiedSignatureRoot( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_CreateManifestSectionEnumerator( )
EISL_CreateSignatureAttributeEnumerator( )

NAME

EISL_CreateVerifiedSignatureRoot

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR EISL_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 *EISL_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/optional)
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 *PublicKey.Data* is NULL, a default value is assumed.

RETURN VALUE

Pointer to a verified signature root object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_CreateManifestSectionEnumerator( )
EISL_CreateSignatureAttributeEnuerator( )
EISL_CreateVerifiedSignatureRootWithCredentialData( )
EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate( )
EISL_CreateManifestAttributeEnumerator( )
EISL_CreateSignerInfoAttributeEnumerator( )

NAME

EISL_CreateVerifiedSignatureRootWithCertificate

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR
EISL_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 *EISL_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

Fn EISL_CreateManifestSectionEnumerator
EISL_CreateSignatureAttributeEnumerator( )
EISL_CreateVerifiedSignatureRootWithCredentialData( )
EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate( )
EISL_CreateManifestAttributeEnumerator( )
EISL_CreateSignerInfoAttributeEnumerator( )

NAME

EISL_FindManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR EISL_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 *EISL_CreateVerifiedSignatureRoot*( ) or *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or implicitly by *EISL_SelfCheck*( ), *EISL_VerifyAndLoadModuleAndCredentials*( ), or *EISL_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

EISL_CreateVerifiedSignatureRoot( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_SelfCheck( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

NAME

EISL_CreateManifestSectionEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR EISL_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 *Root* parameter. The resulting iterator object is activated by invoking the *EISL_GetNextManifestSection*( ) function. The object should be recycled using the *EISL_RecycleManifestSectionEnumerator*( ) call when it is no longer needed.

PARAMETERS

Root (input)

A verified signature root explicitly created by *EISL_CreateVerifiedSignatureRoot*( ) or *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or implicitly by *EISL_SelfCheck*( ), *EISL_VerifyAndLoadModuleAndCredentials*( ), or *EISL_VerifyLoadedModuleAndCredentials*( ).

RETURN VALUE

Pointer to a manifest section iterator object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextManifestSection( )
EISL_RecycleManifestSectionEnumerator( )

NAME

EISL_GetNextManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR EISL_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 *EISL_CreateManifestSectionEnumerator*( ).

RETURN VALUE

The next Manifest Section Object is returned, or NULL if no more sections exist.

SEE ALSO

EISL_CreateManifestSectionEnumerator( )

NAME

EISL_RecycleManifestSectionEnumerator

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateManifestSectionEnumerator*( ).

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateManifestSectionEnumerator( )

NAME

EISL_FindManifestAttribute

SYNOPSIS

```
ISL_STATUS  EISL_FindManifestAttribute
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Context,
    ISL_CONST_DATA Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function locates a specified signature root attribute. The attribute is a name-value pair. Name identifies the signature root attribute to be located. Value is the output parameter containing the length and pointer to the value of the signature root attribute.

This function returns ISL_FAIL if there is no signature root attribute with the specified *Name*.

PARAMETERS

Context (input)

A verified signature root reference returned by one of the functions *EISL_CreateVerifiedSignatureRoot*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or *EISL_GetManifestSignatureRoot*( ).

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

EISL_CreateManifestAttributeEnumerator( )
EISL_GetManifestSignatureRoot( )

NAME

EISL_CreateManifestAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR  EISL_CreateManifestAttributeEnumerator
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Context)
```

DESRIPTION

This function creates a dynamic object whose purpose is to list references to the attributes of the signature root of the signed manifest credential. The iterator object is activated using the function *EISL_GetNextAttribute*( ). The iterator object should be used to retrieve the name-value attribute pairs when the caller does not have prior knowledge of the attribute names. The function *EISL_FindManifestAttribute*( ) can be used to directly locate attribute values based on attribute name.

When the iterator object is no longer needed, is must be recycled using the function *EISL_RecycleAttributeEnumerator*( ).

PARAMETERS

Context (input)
A verified signature root reference returned by one of the functions *EISL_CreateVerifiedSignatureRoot*( ), *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or *EISL_GetManifestSignatureRoot*( ).

RETURN VALUE

Pointer to a signature root attribute-iterator object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextAttribute( )
EISL_RecycleAttributeEnumerator( )
EISL_FindManifestAttribute( )

NAME

EISL_FindSignerInfoAttribute

SYNOPSIS

```
ISL_STATUS  EISL_FindSignerInfoAttribute
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Context,
    ISL_CONST_DATA Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function locates an attribute in the signer information block associated with the verified signature root referenced by *Context*. The attribute is a name-value pair. *Name* identifies the signer information attribute to be located. *Value* is the output parameter containing the length and pointer to the value of the signer information attribute.

This function returns ISL_FAIL if there is no signer information attribute with the specified *Name*.

PARAMETERS

Context (input)
: A verified signature root reference returned by one of the functions *EISL_CreateVerifiedSignatureRoot*(), *EISL_CreateVerifiedSignatureRootWithCertificate*(), or *EISL_GetManifestSignatureRoot*().

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

EISL_CreateSignerInfoAttributeEnumerator()

NAME

EISL_CreateSignerInfoAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR EISL_CreateSignerInfoAttributeEnumerator
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Context)
```

DESRIPTION

This function creates a dynamic object whose purpose is to list references to the attributes of the signer information block associated with the verified signature root. The iterator object is activated using the function *EISL_GetNextAttribute*(). The iterator object should be used to retrieve the name-value attribute pairs when the caller does not have prior knowledge of the attribute names. The function *EISL_FindSignerInfoAttribute*() can be used to directly locate attribute values based on attribute name.

When the iterator object is no longer needed, is must be recycled using the function *EISL_RecycleAttributeEnumerator*().

PARAMETERS

Context (input)

A verified signature root reference returned by one of the functions *EISL_CreateVerifiedSignatureRoot*(), *EISL_CreateVerifiedSignatureRootWithCertificate*(), or *EISL_GetManifestSignatureRoot*().

RETURN VALUE

Pointer to a signer info attribute-iterator object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextAttribute()
EISL_RecycleAttributeEnumerator()
EISL_FindSignerInfoAttribute()

NAME

EISL_GetNextAttribute

SYNOPSIS

```
ISL_STATUS EISL_GetNextAttribute
    (ISL_ITERATOR_PTR Iterator,
    ISL_CONST_DATA_PTR Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the attribute name and value 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 name-value pair. The *Name* and *Value* returned by this function cannot be modified by the program. If no more attribute values are present, the function returns ISL_FAIL.

PARAMETERS

Iterator (input)
: An iterator object created by *EISL_CreateManifestAttributeEnumerator*( ) or *EISL_CreateSignerInfoAttributeEnumerator*( ).

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

EISL_CreateManifestAttributeEnumerator( )
EISL_CreateSignerInfoAttributeEnumerator( )

NAME

EISL_RecycleAttributeEnumerator

SYNOPSIS

```
ISL_STATUS  EISL_RecycleAttributeEnumerator
    (ISL_ITERATOR_PTR Iterator)
```

DESCRIPTION

This function destroys and recycles the memory for the attribute iterator. It must be the last call referencing the iterator.

PARAMETERS

Iterator (input)A

A attribute iterator created by *EISL_CreateManifestAttributeEnumerator*( ) or *EISL_CreateSignerInfoAttributeEnumerator*( ).

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateManifestAttributeEnumerator( )
EISL_CreateSignerInfoAttributeEnumerator( )
EISL_GetNextAttribute( )

NAME

EISL_FindSignatureAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateVerifiedSignatureRoot*( ) or *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or implicitly by *EISL_SelfCheck*( ), *EISL_VerifyAndLoadModuleAndCredentials*( ), or *EISL_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

EISL_CreateVerifiedSignatureRoot( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_SelfCheck( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_GetModuleManifestSection( )
EISL_GetManifestSignatureRoot( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

NAME

EISL_CreateSignatureAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR EISL_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 *EISL_GetNextSignatureAttribute*( ) function. The object should be recycled using the *EISL_RecycleSignatureEnumerator*( ) function when it is no longer needed.

PARAMETERS

Root (input)

A verified signature root explicitly created by *EISL_CreateVerifiedSignatureRoot*( ) or *EISL_CreateVerifiedSignatureRootWithCertificate*( ), or implicitly by *EISL_SelfCheck*( ), *EISL_VerifyAndLoadModuleAndCredentials*( ), or *EISL_VerifyLoadedModuleAndCredentials*( ).

RETURN VALUE

Pointer to a signature attribute iterator object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextSignatureAttribute( )
EISL_RecycleSignatureAttributeEnumerator( )
EISL_SelfCheck( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_GetModuleManifestSection( )
EISL_GetManifestSignatureRoot( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

EISL_GetNextSignatureAttribute

NAME

EISL_GetNextSignatureAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_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

EISL_CreateSignatureAttributeEnumerator( )

NAME

EISL_RecycleSignatureAttributeEnumerator

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateSignatureAttributeEnumerator*( ).

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateSignatureAttributeEnumerator( )

NAME

EISL_RecycleVerifiedSignatureRoot

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateVerifiedSignatureRoot*( ) or *EISL_CreateVerifiedSignatureRootWithCertificate*( ).

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateVerifiedSignatureRoot( )
EISL_CreateVerifiedSignatureRootWithCertificate( )
EISL_CreateVerifiedSignatureRootWithCredentialData( )
EISL_CreateVerifiedSignatureRootWithCredentialDataAndCertificate( )

22.3 Certificate Chain Methods

The man-page definitions for functions to manipulate certificate chains in a PKCS#7 signature block are presented in this section.

NAME

EISL_CreateCertificateChainWithCredentialData

SYNOPSIS

```
const ISL_VERIFIED_CERTIFICATE_CHAIN_PTR
EISL_CreateCertificateChainWithCredentialData
    (const ISL_CONST_DATA RootIssuer,
    const ISL_CONST_DATA PublicKey,
    const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath)
```

DESCRIPTION

This function constructs and verifies a certificate chain that begins with the root certificate authority identified by the distinguished name *RootIssuer* and the *PublicKey*, and ends with the certificate of a signer of the signed manifest credentials contained in *CredentialsImage*. The certificates required to construct the chain must be contained in the PKCS#7 signature block of the signed manifest credential.

During the construction process, each certificate is verified, beginning with the certificate of the *RootIssuer*.

The *ModuleSearchPath* is a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*. The *ModuleSearchPath* is stored as state information associated with the verified certificate chain returned by this function. The information is available to subsequent operations on the verified certificate chain.

PARAMETERS

RootIssuer (input) The distinguished name of the root certificate authority

PublicKey (input)
The public key of the root certificate authority.

CredentialsImage (input)
A pointer to the memory-resident signed manifest credentials containing certificates used to construct the certificate chain.

ModuleSearchPath (input)
A string containing a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*.

RETURN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

EISL_CreateCertificateChainWithCredentialDataAndCertificate()
EISL_CreateCertificateChain()
EISL_CreateCertificateChainWithCertificate()
EISL_RecycleCertificateChain()

NAME

EISL_CreateCertificateChainWithCredentialDataAndCertificate

SYNOPSIS

```
ISL_VERIFIED_CERTIFICATE_CHAIN_PTR
EISL_CreateCertificateChainWithCredentialDataAndCertificate
    (const ISL_CONST_DATA Certificate,
    const ISL_CONST_DATA CredentialsImage,
    const ISL_CONST_DATA ModuleSearchPath)
```

DESCRIPTION

This function constructs and verifies a certificate chain that begins with the root certificate authority Certificate and ends with the certificate of a signer of the signed manifest credentials contained in *CredentialsImage*. The certificates required to construct the chain must be contained in the PKCS#7 signature block of the signed manifest credential.

During the construction process, each certificate is verified, beginning with the certificate of the root authority.

The *ModuleSearchPath* is a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*. The *ModuleSearchPath* is stored as state information associated with the verified certificate chain returned by this function. The information is available to subsequent operations on the verified certificate chain.

PARAMETERS

Certificate (input)
The certificate of the root of the certificate chain.

CredentialsImage (input)
A pointer to the memory-resident signed manifest credentials containing certificates used to construct the certificate chain.

ModuleSearchPath (input)
A string containing a colon-separated list of fully-qualified file system path names for locating the object code modules referenced by the manifest sections of the *CredentialsImage*.

RETUEN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

EISL_CreateCertificateChainWithCredentialData( )
EISL_CreateCertificateChain( )
EISL_CreateCertificateChainWithCertificate( )
EISL_RecycleCertificateChain( )

NAME

EISL_CreateCertificateChain

SYNOPSIS

```
ISL_VERIFIED_CERTIFICATE_CHAIN_PTR EISL_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

EISL_RecycleCertificateChain()
EISL_CreateCertificateChainWithCertificate()
EISL_CreateCertificateChainWithCredentialData()
*EISL_*()CreateCertificateChainWithCredentialDataAndCertificate

NAME

EISL_CreateCertificateChainWithCertificate

SYNOPSIS

```
ISL_VERIFIED_CERTIFICATE_CHAIN_PTR
EISL_CreateCertificateChainWithCertificate
    (const ISL_CONST_DATA Certificate,
    const ISL_CONST_DATA Credential)
```

DESCRIPTION

This function constructs and verifies a certificate chain that begins with the root certificate authority *Certificate* and ends with the certificate of a signer of the *Credential*. The *Credential* is a fully-qualified file system path name identifying the location of a signed manifest credential. The certificates required to construct the chain must be contained in the PKCS#7 signature block of the signed manifest credential.

During the construction process, each certificate is verified, beginning with the certificate of the root authority.

PARAMETERS

Certificate (input)
: The certificate of the root of the certificate chain.

Credential (input)
: A pointer to the memory-resident signed manifest credentials containing certificates used to construct the certificate chain.

RETURN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

EISL_CreateCertificateChainWithCredentialData()
EISL_CreateCertificateChainWithCredentialDataAndCertificate()
EISL_CreateCertificateChain()
EISL_RecycleCertificateChain()

EISL_CopyCertificateChain

NAME

EISL_CopyCertificateChain

SYNOPSIS

```
uint32 EISL_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 *EISL_CreateCertificateChain*( ) or *EISL_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

EISL_CreateCertificateChain( )
EISL_GetCertificateChain( )
EISL_CreateCertificateChainWithCertificate( )
EISL_CreateCertificateChainWithCredentialData( )
EISL_CreateCertificateChainWithCredentialDataAndCertificate( )

NAME

EISL_RecycleVerifiedCertificateChain

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateCertificateChain*( ).

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateCertificateChain( )
EISL_CreateCertificateChainWithCertificate( )
EISL_CreateCertificateChainWithCredentialData( )
*EISL_*( )CreateCertificateChainWithCredentialDataAndCertificate

22.4 Certificate Attribute Methods

The man-page definitions for Certificate Methods are presented in this section.

NAME

EISL_FindCertificateAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_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

EISL_CopyCertificateChain( )

NAME

EISL_CreateCertificateAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR EISL_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 *EISL_GetNextCertificateAttribute*( ) function. The object must be recycled using the *EISL_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

EISL_GetNextCertificateAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_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

EISL_CreateCertificateAttributeEnumerator( )

NAME

EISL_RecycleCertificateAttributeEnumerator

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateCertificateAttributeEnumerator*().

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateCertificateAttributeEnumerator()

22.5 Manifest Section Object Methods

The man-page definitions for Manifest Section Object Methods are presented in this section.

NAME

EISL_GetManifestSignatureRoot

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR EISL_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 *EISL_GetNextManifestSection*( ), *EISL_GetModuleManifestSection*( ), or *EISL_FindManifestSection*( ).

RETURN VALUE

Pointer to a signature root object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextManifestSection( )
EISL_FindManifestSection( )
EISL_GetModuleManifestSection( )

NAME

EISL_VerifyAndLoadModule

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_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 *EISL_GetNextManifestSection*( ) or *EISL_FindManifestSection*( ) functions.

RETURN VALUE

Pointer to a verified module object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextManifestSection( )
EISL_FindManifestSection( )

NAME

EISL_VerifyLoadedModule

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR EISL_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 *EISL_GetNextManifestSection*( ), *EISL_GetModuleManifestSection*( ), or *EISL_FindManifestSection*( ) functions.

RETURN VALUE

Pointer to a verified module object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_GetNextManifestSection( )
EISL_FindManifestSection( )

NAME

EISL_FindManifestSectionAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_FindManifestSection*( ), *EISL_GetModuleManifestSection*( ), or *EISL_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

EISL_FindManifestSection( )
EISL_GetNextManifestSection( )

NAME

EISL_CreateManifestSectionAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR EISL_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 *EISL_GetNextManifestSectionAttribute*( ) function. The object must be recycled using the *EISL_RecycleManifestSectionEnumerator*( ) function when it is no longer needed.

PARAMETERS

Section (input)
A manifest section object returned by the *EISL_FindManifestSection*( ), *EISL_GetModuleManifestSection*( ), or *EISL_GetNextManifestSection*( ) functions.

RETURN VALUE

Pointer to a signed object attribute iterator object if successful, or NULL if unsuccessful.

SEE ALSO

EISL_FindManifestSection( )
EISL_GetNextManifestSection( )

NAME

EISL_GetNextManifestSectionAttribute

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_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

EISL_CreateManifestSectionAttributeEnumerator( )

NAME

EISL_RecycleManifestSectionAttributeEnumerator

SYNOPSIS

```
ISL_STATUS EISL_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 *EISL_CreateManifestSectionAttributeEnumerator.*()

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_CreateManifestSectionAttributeEnumerator()

NAME

EISL_GetModuleManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR EISL_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 module object returned by any of the following functions:

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

EISL_SelfCheck( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

22.6 Secure Linkage Services

The man-page definitions for Secure Linkage Services are presented in this section.

NAME

EISL_LocateProcedureAddress

SYNOPSIS

```
ISL_FUNCTION_PTR EISL_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. This function will return the address of the function specified by *Name*, only if that function is exported by the module it appears in.

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 *EISL_CheckAddressWithinModule*( ) can to check the address for containment in the verified module.

PARAMETERS

Module (input)

A verified module object returned by any of the following functions:
EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

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

EISL_CheckAddressWithinModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_SelfCheck( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )

EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

NAME

EISL_GetReturnAddress

SYNOPSIS

```
#define EISL_GetReturnAddress(Address)   \
{                                        \
    /* Platform specific code in here */
}
```

DESCRIPTION

This macro gets the current return address and 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 *EISL_GetReturnAddress*(), *EISL_GetReturnAddress*() returns value R. Function B can validate that address R is within a verified module which should contain function A using *EISL_CheckAddressWithinModule*().

This function macro is platform and compiler dependent.

PARAMETERS

Address (output)
Pointer in which return address value is returned.

RETURN VALUE

Results in copying the return address value in Address pointer.

SEE ALSO

EISL_CheckAddressWithinModule()

NAME

EISL_CheckAddressWithinModule

SYNOPSIS

```
ISL_STATUS EISL_CheckAddressWithinModule
    (ISL_VERIFIED_MODULE_PTR Verification,
    ISL_FUNCTION_PTR Address)
```

DESCRIPTION

The *Address* is checked against the list of valid address ranges for executable code within the module identified by the verified module pointer.

PARAMETERS

Verification (input)
A verified module object returned by any of the following functions:
EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
7Fn EISL_DuplicateVerifiedModulePtr

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

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

NAME

EISL_CheckDataAddressWithinModule

SYNOPSIS

```
ISL_STATUS EISL_CheckDataAddressWithinModule
    (ISL_VERIFIED_MODULE_PTR Verification,
    const void *Address)
```

DESCRIPTION

The *Address* is checked against the list of valid address ranges for the data space within the module identified by the verified module pointer.

PARAMETERS

Verification (input)

A verified module object returned by any of the following functions:
EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
7Fn EISL_DuplicateVerifiedModulePtr

Address (input)

A data address to be checked.

RETURN VALUE

ISL_OK is returned if the data address is a valid address within the bounds of the module, otherwise ISL_FAIL is returned.

SEE ALSO

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_DuplicateVerifiedModulePtr( )

NAME

EISL_GetLibHandle

SYNOPSIS

```
void * EISL_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 any of the following functions:
EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )
EISL_VerifyAndLoadModule( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )

RETURN VALUE

The handle to the loaded object code is returned, or NULL if failure.

SEE ALSO

EISL_SelfCheck( )
EISL_VerifyLoadedModule( )
EISL_VerifyAndLoadModule( )
EISL_VerifyLoadedModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentials( )
EISL_VerifyAndLoadModuleAndCredentialsWithCertificate( )
EISL_VerifyAndLoadModuleAndCredentialData( )
EISL_VerifyAndLoadModuleAndCredentialDataWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialsWithCertificate( )
EISL_VerifyLoadedModuleAndCredentialData( )
EISL_VerifyLoadedModuleAndCredentialDataWithCertificate( )

Technical Standard

Part 11:

Signed Manifest

The Open Group

Chapter 23

Introduction

23.1 Signed Manifests

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.

23.2 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 29-1 on page 792 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.

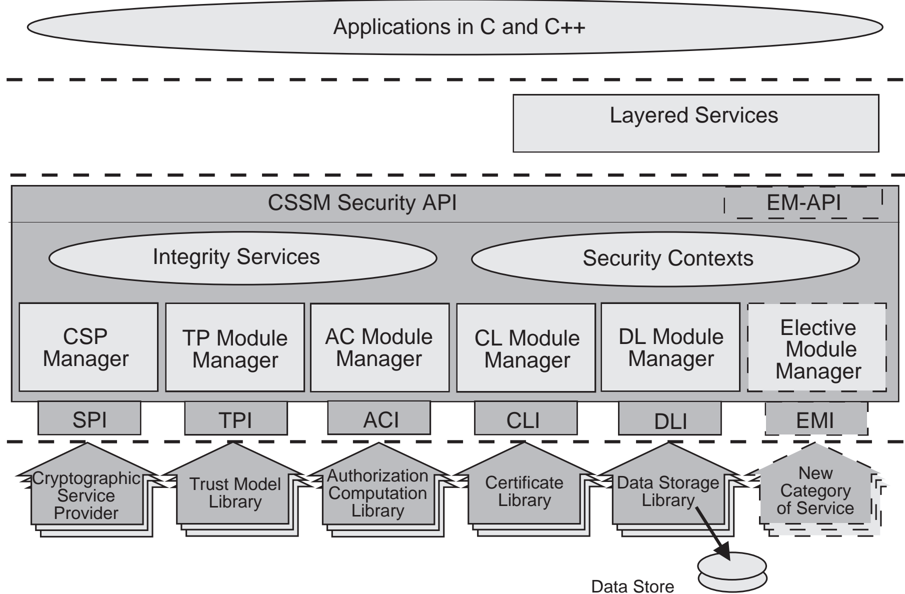


Figure 23-1 The Common Data Security Architecture for All Platforms

Chapter 24

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.

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 25

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.

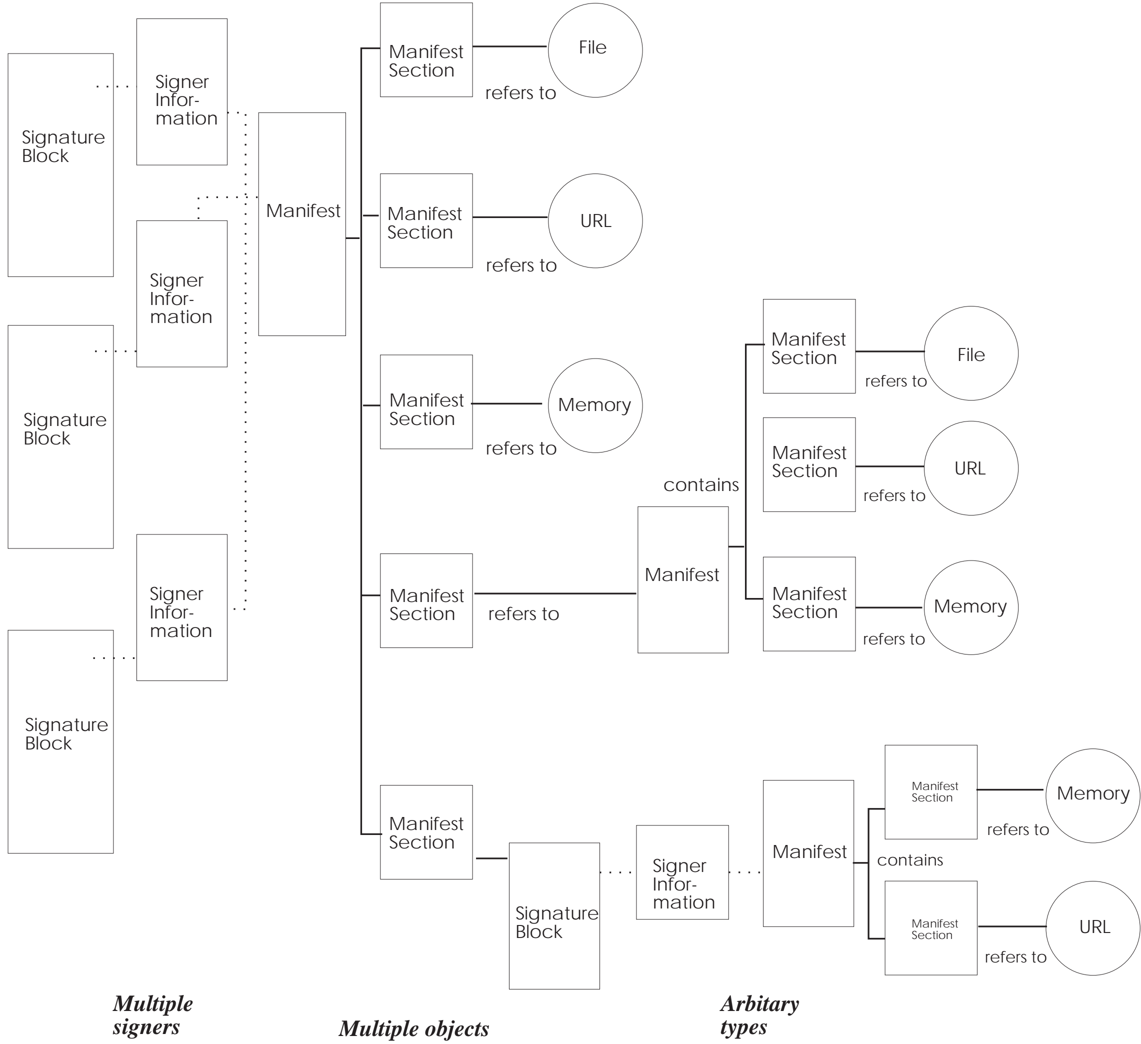


Figure 25-1 Signed Manifest Architectural View

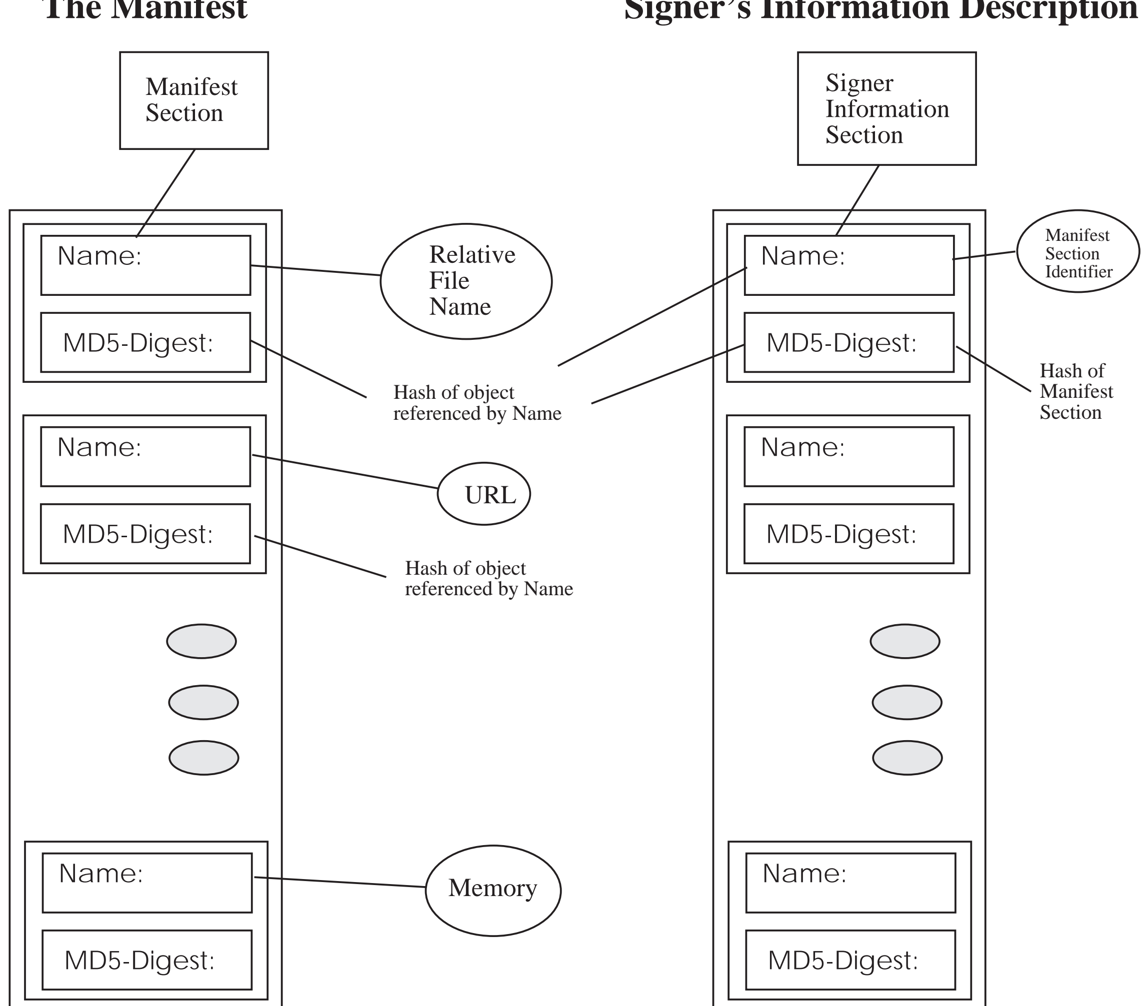
The signed manifest is built from the following components:

- 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 31-2.



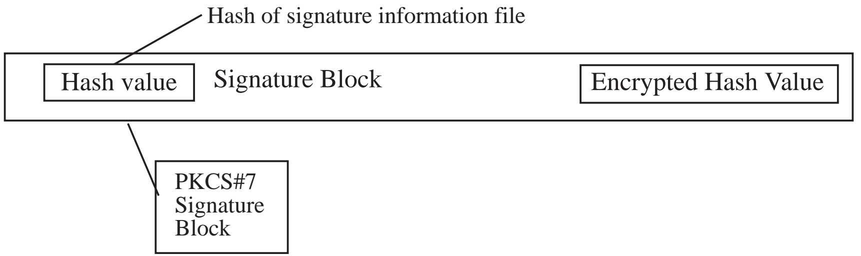


Figure 25-2 Relationships of Manifest, Signer's Info and Signature Block

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 26

Format Specification

This Chapter presents the format specification for the components that make up a signed manifest.

26.1 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

26.1.1 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
```

26.1.2 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.

26.1.3 Format Specification

This section 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_entry:<section><newline>*
```

The format specification for signer information is:

```
signer_info:"Signature-Version: 2.0" \
    <newline>+<header_attr>*<newline>+ \
    <signer_info_ent>+signer_info_ent:<section><newline>*
```

The format specification for a section (both manifest and signer information) is:

```
header_attr:<attribname>: <value><newline>
wspace_sep_lst:<headerchar>+<wspace_next>*
wspace_next:\s+<headerchar>+
section:<nameheader><sect_line>*<newline>
sect_line:{<sect_name>|<digest_alg>|<digest_val>|<sect_attr>}
nameheader:"Name:" <value><newline>
sect_name:"SectionName:" <value><newline>
digest_alg:"Digest-Algorithms:" <value><newline>
digest_val:<digest_name_val>":" <value><newline>
digest_name_val:<attribname>"-Digest"
sect_attr:<attribname>: <value><newline>
value:<otherchar>*<cont>*
attribname:<alphanum><headerchar>*
cont:<newline> <otherchar>+
newline:{\r\n|\n}
headerchar:{<alphanum>|_|-}
alphanum:[A-Za-z0-9]
otherchar:.
number:[0-9]+
```

A section begins with the **Name** token and ends when a new section begins or an end of file is encountered.

26.1.4 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.

26.1.5 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 to this Part 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
```

26.1.6 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.

26.1.7 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
```

26.2 Signer 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.

26.2.1 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.

26.2.2 Signer 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 signature section and 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)
```

26.2.3 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)
```

26.3 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 27

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.

27.1 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:

- For each signature section in the signer's information:
  - — Locate the corresponding manifest section matching on the value of the *Name* attribute
  - — 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.

27.2 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 28

File-Based Representation of Signed Manifests

28.1 Description

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

A persistent signed manifest must reside in at least three files. The three files are zipped to create a single credential file in PKZIP format. Each file has an identifying suffix:

- The zipped credential filename suffix is .esw
- The manifest filename suffix is .mf
- The signer information filename suffix is .sf
- The signature block filename suffix identifies the signature type and is one of the following:

.rsa	(PKCS7 signature, MD5 + RSA)
.dsa	(PKCS7 signature, DSA)
.pgp	(Pretty Good Privacy Signature)

The archive format of an .esw file must conform to the archive format specified by PKWARE.

28.2 Representation Constraints

Filenames for the manifest file, signer information file and signature block file are restricted to the printable characters A-Z a-z 0-9 and dash and underscore. Base filenames consist of at most eight characters.

All file suffixes must be recognized in upper and lower case.

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 do not put a new line at the end of the last line, and one so that the grammar does not have to handle the last entry as a special case, 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, making the maximum length of a header name 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 65,535-byte (not character) header values, and 65,535 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 29

Signed Manifests—Examples

The following is a list of examples that serve to illustrate how this specification meets the requirements for signed manifests.

29.1 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.

29.2 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**.

29.2.1 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.

29.3 Embedded or Nested Referent Objects

29.3.1 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
```

29.3.2 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.

29.3.3 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.

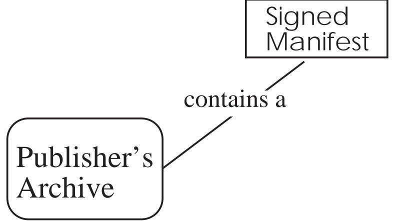


Figure 29-1 Relationship of Publisher's Archive and Signed Manifest

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.

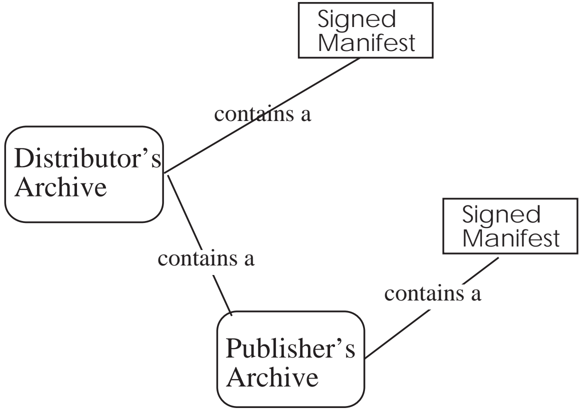


Figure 29-2 Relationship of Distributor's Archive to Publisher's Archive

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
```

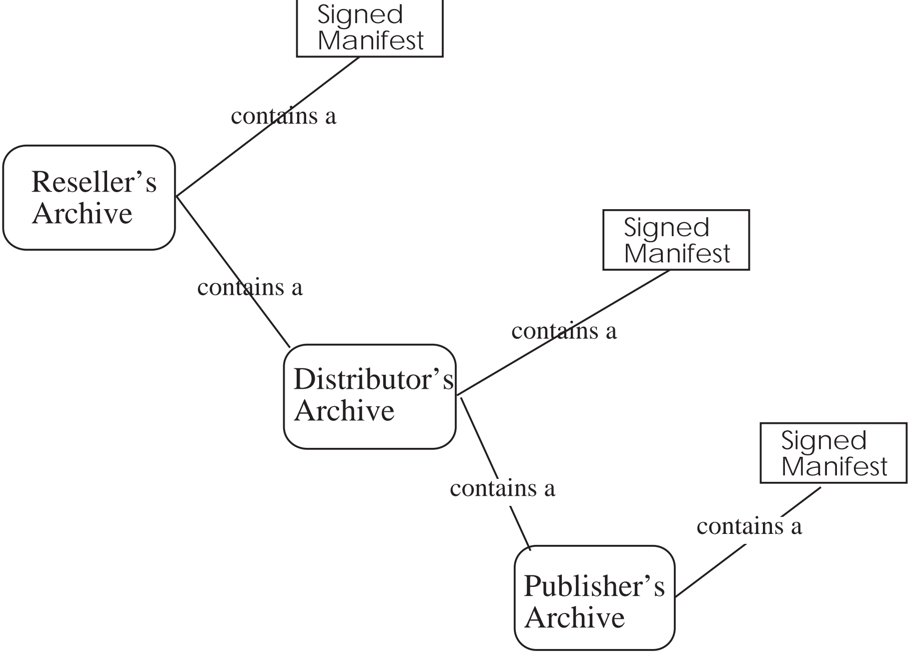


Figure 29-3 Relationship of Reseller to Distributor to Publisher

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.

29.3.4 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.

29.3.5 Foreign Language Support—Multiple Hash Values

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.

29.3.6 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.

29.3.7 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.

Technical Standard

Part 12:

OIDs for Certificate Library Modules

The Open Group

Chapter 30

Introduction

Applications using a single type of certificate can choose among Certificate Library Service Modules only if those service modules process those certificates in a standard manner. General interoperability is difficult to achieve. Standard Object identifiers can enable data-level interoperability, allowing an application to extract values from certificates and CRLs in a uniform manner, regardless of the certificate library module being used to access the certificate.

Certificate values are managed as name-value pairs through the CSSM APIs. Interoperability requires specification of the name space and specification of the representation for certificate values. The name space is defined as a set of OIDs, one per meaningful aggregation of certificate field values. If the certificate field values can be presented in several distinct representations, then each OID also indicates the selected representation of the certificate field values.

Several standards organizations have defined object identifiers for other security objects. In conjunction with the X.501 Directory Standard, the ITU has defined OIDs for directory data types. The standard PKCS-7, version 1.5 includes OID definitions for secured data objects contained in PKCS-7 messages. The X9 Financial standards organization has also defined OIDs for certificate extensions related to secured financial operations and services.

For the promotion of interoperable X.509 certificate services though the Common Data Security Architecture (CDSA), this Technical Standard defines a set of OIDs to identify fields in X.509 certificates and CRLs.

Chapter 31

OIDs for X.509 Certificate Library Modules

31.1 Overview

This chapter specifies object identifiers and corresponding data structures for fields of X.509 Certificates and Certificate Revocation Lists (CRLs). An OID can specify one field or multiple fields contained in a certificate or CRL. The OID also indicates the data representation of the field values. One to three distinct representations are defined for each meaningful aggregation of certificate field values:

- BER/DER encoded values
  The IETF standard specification of X.509 certificates and the BER/DER encoding define one representation for certificate values.
- Native platform encoding of a complex (bushy) data structure
  For performance and when using certificate values locally, applications can prefer decoded certificate values stored in bushy data structures that are native to the platform. C language structures are defined for each named aggregation of certificate field values.
- LDAP String format
  The IETF standard format for LDAP strings is a valid representation for selected fields of a certificate.

Some certificate fields can be returned to an application in any of the three formats. Applications specify the desired format by using distinct OID names. The OID names for a single field in different representations share a common prefix. The selected representation is identified by a unique OID suffix. This allows applications to store tables of the common base and to select the desired representation at runtime by appending the suffix corresponding to the desired representation.

31.2 Interoperable Format Specifications for X.509

31.2.1 Certificate Library Service Provider X.509 Field OIDs

This section defines the OID names to be used to access fields in X.509 certificates and CRLs. The format of the data accessed with each OID is described.

Following sections then describe the OIDs upon which Certificate and CRL OIDs are based:

- Section 37.3 on page 824 describes the OIDs used to access information in an X.509 certificate and the associated data structures
- Chapter 38 on page 839 describes the OIDs used to access information in an X.509 CRL and the associated data structures.

31.2.2 Base of the Object Identifier Name Space

This specification defines five object identifiers, which form the base arcs for Intel Corporation's CDSA name space.

INTEL OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741) }

The object identifier INTEL identifies the base arc of the Intel Corporation name space under the registration authority of the joint ISO and the International Telegraph and Telephone Consultative Committee.

INTEL_CDSASECURITY OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741) CDSA-security (2) }

The object identifier INTEL_CDSASECURITY identifies the base arc for CDSA object identifiers with the Intel Corporation name space. The CDSA name space is subdivided into two subarcs:

- formats
- algs

INTEL_SEC_FORMATS OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741) CDSA-security (2) formats (1) }

The object identifier INTEL_SEC_FORMATS identifies the base arc of object identifiers representing the format or representation of a CDSA security object within the Intel Corporation CDSA name space.

INTEL_SEC_ALGS OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741) CDSA-security (2) algs (2) 5 }

The object identifier INTEL_SEC_ALGS identifies the base arc of object identifiers representing the format or representation of CDSA security algorithms within the Intel Corporation CDSA name space.

The object identifier INTEL_SEC_FORMATS identifies the base arc of object identifiers representing the format or representation of a CDSA security object within the Intel Corporation CDSA name space. A subarc for security object bundles is defined within the CDSA formats object identifier name space.

INTEL_SEC_OBJECT_BUNDLE OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741)
CDSA-security (2) formats (1) bundle(4)}

The object identifier INTEL_SEC_OBJECT_BUNDLE identifies the base arc for object identifiers representing bundles of CDSA security object within the Intel Corporation CDSA name space.

INTEL_CERT_AND_PRIVATE_KEY_2_0 OBJECT IDENTIFIER ::=
{ joint-ise-ccitt (2) country (16) usa (840) org (1) intel (113741)
CDSA-security (2) formats (1) bundle (4) 1}

The object identifier INTEL_CERT_AND_PRIVATE_KEY_2_0 identifies a certificate and private key object contained within a bundle.

31.2.3 Programmatic Definition of Base Object Identifiers

Programmatically these Intel base object identifiers are defined by the following constants.

```
#define INTEL                             96, 134, 72, 1, 134, 248, 77
#define INTEL_LENGTH                      7

#define INTEL_CDSASECURITY                INTEL, 2
#define INTEL_CDSASECURITY_LENGTH         (INTEL_LENGTH + 1)

#define INTEL_SEC_FORMATS                 INTEL_CDSASECURITY, 1
#define INTEL_SEC_FORMATS_LENGTH          (INTEL_CDSASECURITY_LENGTH + 1)

#define INTEL_SEC_ALGS                    INTEL_CDSASECURITY, 2, 5
#define INTEL_SEC_ALGS_LENGTH             (INTEL_CDSASECURITY_LENGTH + 2)

#define INTEL_SEC_OBJECT_BUNDLE           INTEL_SEC_FORMATS, 4
#define INTEL_SEC_OBJECT_BUNDLE_LENGTH    (INTEL_SEC_FORMATS_LENGTH + 1)

#define INTEL_CERT_AND_PRIVATE_KEY_2_0    INTEL_SEC_OBJECT_BUNDLE, 1
#define INTEL_CERT_AND_PRIVATE_KEY_2_0_LENGTH
                                     (INTEL_SEC_OBJECT_BUNDLE_LENGTH + 1)
```

31.2.4 Terminology

BER Integer:
: An integer value, base 256, in two's complement form, most significant digit first, with a minimum number of octets.

31.3 Object Identifiers for X.509 V3 Certificates

31.3.1 Base Object Identifiers

This specification defines object identifiers to name fields and sets of fields within an X.509 certificate. Each object identifier also indicates the representation for the selected field or fields. Possible representations include:

- DER encoded value - as defined by defined the CCITT in Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1). 1988.
- C language structure with values in native platform representation - a data structure is defined for each set of fields that can be reasonably represented as a C language data structure
- LDAP String value - an LDAP string representation is defined for selected certificate fields

Object identifiers are defined corresponding to the certificate fields defined by the X.509 V1 standard and the X.509 V3 standard. Two primary subarcs are defined for this purpose:

```
INTEL_X509V3_CERT_R08  OBJECT IDENTIFIER ::= { INTEL_SEC_FORMATS, 1, 1 }
INTEL_X509V3_SIGN_R08  OBJECT IDENTIFIER ::= { INTEL_SEC_FORMATS, 3, 2 }
```

The object identifier INTEL_X509V3_CERT_R08 identifies the base arc for object identifiers representing the format and name of one or more fields contained in an X.509 version 3 certificate. The object identifier INTEL_X509V3_SIGN_R08 identifies the base arc for object identifiers representing the format and name of the subfields of a digital signature contained in an X.509 version 3 certificate

A subarc for X.509 version certificate extensions is defined under INTEL_X509V3_CERT_R08 as follows:

```
INTEL_X509V3_CERT_PRIVATE_EXTENSIONS
    OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 50 }
```

31.3.2 Programmatic Definition of Base Object Identifiers

Programmatically, these object identifiers are defined by the following constants.

```
/* Prefix for defining Certificate field OIDs */
#define INTEL_X509V3_CERT_R08           INTEL_SEC_FORMATS, 1, 1
#define INTEL_X509V3_CERT_R08_LENGTH    INTEL_SEC_FORMATS_LENGTH + 2

/* Prefix for defining Certificate Extension field OIDs */
#define INTEL_X509V3_CERT_PRIVATE_EXTENSIONS  INTEL_X509V3_CERT_R08, 50
#define INTEL_X509V3_CERT_PRIVATE_EXTENSIONS_LENGTH
                                       INTEL_X509V3_CERT_R08_LENGTH + 1

/* Prefix for defining signature field OIDs */
#define INTEL_X509V3_SIGN_R08           INTEL_SEC_FORMATS, 3, 2
#define INTEL_X509V3_SIGN_R08_LENGTH    INTEL_SEC_FORMATS_LENGTH + 2

/* Suffix specifying format or representation of a field value */
/* Note that if a format suffix is not specified, a flat data
representation is implied */
#define INTEL_X509_C_DATATYPE             1
#define INTEL_X509_LDAPSTRING_DATATYPE    2
```

31.3.3 Object Identifiers for Fields

This specification defines object identifiers for naming fields of an X.509 version 3 or X.509 version 1 certificate. The object identifier also indicates the representation or format of the specific field or fields from the certificate. The valid representations include:

- Flat data representation. Generally a DER encoded value - as defined by the CCITT in Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1). 1988 -- with the object type tag discarded. When an OID indicates a flat data representation of a DER encoded value (where the DER encoding includes tag, length & value), the tag of the DER encoding is discarded, *FieldValue.Length* is the length (in bytes) of the value, and *FieldValue.Data* is the value. The length and value are contained in a single CSSM_DATA structure.
- C language structure with values in native platform representation - a data structure is defined for each set of fields that can be reasonably represented as a C language data structure. When an OID indicates a C structure, the *FieldValue.Length* is the size (in bytes) of the pointer to the C structure, and *FieldValue.Data* points to the C structure.
- LDAP String value - an LDAP string representation is defined for selected certificate fields. When an OID indicates an LDAP string representation, the *FieldValue.Length* is the length (in bytes) of the LDAP string and *FieldValue.Data* is the LDAP string. The LDAP string is represented as a *PrintableString* or in a UTF8 encoding as defined in LDAP RFC 2253.

31.3.4 Certificate OID Definition

The certificate object identifiers are defined as follows:

X509V3SignedCertificate
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 0}

X509V3SignedCertificateCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 0, INTEL_X509_C_DATATYPE},

X509V3TbsCertificate
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 1},

X509V3TbsCertificateCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 1, INTEL_X509_C_DATATYPE}

X509V1Version
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 2}

X509V1SerialNumber
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 3}

X509V1IssuerName
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 5},

X509V1IssuerNameCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 5, INTEL_X509_C_DATATYPE}

X509V1IssuerNameLDAP
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 5, INTEL_X509_LDAPSTRING_DATATYPE}

X509V1ValidityNotBefore
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 6}

X509V1ValidityNotAfter
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 7}

X509V1SubjectName
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 8}

X509V1SubjectNameCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 8, INTEL_X509_C_DATATYPE}

X509V1SubjectNameLDAP
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 8, INTEL_X509_LDAPSTRING_DATATYPE}

CSSMKeyStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 20}

X509V1SubjectPublicKeyCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 20, INTEL_X509_C_DATATYPE}

X509V1SubjectPublicKeyAlgorithm
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 9}

X509V1SubjectPublicKeyAlgorithmParameters
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 18}

X509V1SubjectPublicKey
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 10}

X509V1CertificateIssuerUniqueId
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 11}

X509V1CertificateSubjectUniqueId
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 12}

X509V3CertificateExtensionsStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 21}

X509V3CertificateExtensionsCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 21, INTEL_X509_C_DATATYPE}

X509V3CertificateNumberOfExtensions
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 14}

X509V3CertificateExtensionStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 13}

X509V3CertificateExtensionCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 13, INTEL_X509_C_DATATYPE}

X509V3CertificateExtensionId
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 15}

X509V3CertificateExtensionCritical
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 16}

X509V3CertificateExtensionType
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 19}

X509V3CertificateExtensionValue
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 17}

31.3.5 Signature OID Definition

The signature object identifiers for a digital signature are defined as follows:

X509V1SignatureStruct
OBJECT IDENTIFIER :: = {INTEL_X509V3_SIGN_R08, 0}

X509V1SignatureCStruct
OBJECT IDENTIFIER ::= {INTEL_X509V3_SIGN_R08, 0, INTEL_X509_C_DATATYPE}

X509V1SignatureAlgorithm
OBJECT IDENTIFIER ::= {INTEL_X509V3_SIGN_R08, 1}

X509V1SignatureAlgorithmParameters
OBJECT IDENTIFIER ::= {INTEL_X509V3_SIGN_R08, 3}

X509V1Signature
OBJECT IDENTIFIER ::= {INTEL_X509V3_SIGN_R08, 2}

31.3.6 Extension OID Definition

The X.509 standard extension OIDs can be used to access the associated certificate (and CRL) extension data.

In addition, Intel has defined and reserved a base object identifier name space for the definition of new OIDs that name specific, new certificate extensions.

INTEL_X509V3_CERT_R08, 50
is reserved for the Extension Contents OID tree

INTEL_X509V3_CERT_PRIVATE_EXTENSIONS
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_R08, 50}

Under the subarc INTEL_X509V3_CERT_PRIVATE_EXTENSIONS, Intel defines the following object identifiers:

SubjectSignatureBitmap
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_PRIVATE_EXTENSIONS,1}

SubjectPicture
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_PRIVATE_EXTENSIONS,2}

SubjectEmailAddress
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_PRIVATE_EXTENSIONS,3}

UseExemptions
OBJECT IDENTIFIER ::= {INTEL_X509V3_CERT_PRIVATE_EXTENSIONS,4}

31.4 C Language Data Structures

This section defines the C Language Data Structures for X.509 Certificates (and CRLs).

31.4.1 CSSM_BER_TAG

This data type defines CSSM programmatic names for the standard DER tags found in DER-encoded values. These tag values are included in a structure containing a certificate field value when the DER type for that field is ambiguous.

```
typedef uint8 CSSM_BER_TAG;

#define BER_TAG_UNKNOWN              0
#define BER_TAG_BOOLEAN              1
#define BER_TAG_INTEGER              2
#define BER_TAG_BIT_STRING           3
#define BER_TAG_OCTET_STRING         4
#define BER_TAG_NULL                 5
#define BER_TAG_OID                  6
#define BER_TAG_OBJECT_DESCRIPTOR    7
#define BER_TAG_EXTERNAL             8
#define BER_TAG_REAL                 9
#define BER_TAG_ENUMERATED           10


/* 12 to 15 are reserved for future versions of the recommendation */
#define BER_TAG_PKIX_UTF8_STRING     12

#define BER_TAG_SEQUENCE             16
#define BER_TAG_SET                  17

#define BER_TAG_NUMERIC_STRING       18
#define BER_TAG_PRINTABLE_STRING     19
#define BER_TAG_T61_STRING           20
#define BER_TAG_TELETEX_STRING       BER_TAG_T61_STRING
#define BER_TAG_VIDEOTEX_STRING      21
#define BER_TAG_IA5_STRING           22

#define BER_TAG_UTC_TIME             23
#define BER_TAG_GENERALIZED_TIME     24

#define BER_TAG_GRAPHIC_STRING       25
#define BER_TAG_ISO646_STRING        26
#define BER_TAG_GENERAL_STRING       27
#define BER_TAG_VISIBLE_STRING       BER_TAG_ISO646_STRING

/* 28 - are reserved for future versions of the recommendation */
#define BER_TAG_PKIX_UNIVERSAL_STRING 28
#define BER_TAG_PKIX_BMP_STRING      30
```

31.4.2 CSSM_X509_ALGORITHM_IDENTIFIER

This structure holds an object identifier naming a cryptographic algorithm and an optional set of parameters to be used as input to that algorithm.

```
typedef struct cssm_x509_algorithm_identifier {
    CSSM_OID algorithm;
    CSSM_DATA parameters;
} CSSM_X509_ALGORITHM_IDENTIFIER, *CSSM_X509_ALGORITHM_IDENTIFIER_PTR;
```

DESCRIPTION

algorithm
An industry standard OID value naming a cryptographic algorithm.

parameters
An optional algorithm-specific set of parameters to be used as input to the algorithm. If no parameters are specified, *parameters.Length* = 0 and *parameters.Data* = NULL.

31.4.3 CSSM_X509_TYPE_VALUE_PAIR

This structure contain an type-value pair.

```
/* X509 Distinguished name structure */
typedef struct cssm_x509_type_value_pair {
    CSSM_OID type;
    CSSM_BER_TAG valueType;    /* The Tag to be used when */
                                   /*this value is BER encoded */
    CSSM_DATA value;
} CSSM_X509_TYPE_VALUE_PAIR, *CSSM_X509_TYPE_VALUE_PAIR_PTR;
```

DESCRIPTION

type
An industry standard OID identifying the type of the value.

valueType
A tag to be used when the value is encoded.

value
The data value.

31.4.4 CSSM_X509_RDN

This structure contains a Relative Distinguished Name composed of an ordered set of type-value pairs.

```
typedef struct cssm_x509_rdn {
    uint32 numberOfPairs;
    CSSM_X509_TYPE_VALUE_PAIR_PTR AttributeTypeAndValue;
} CSSM_X509_RDN, *CSSM_X509_RDN_PTR;
```

DESCRIPTION

numberOfPairs
The number of type-value pairs in the Relative Distinguished Name.

AttributeTypeAndValue
A pointer to an array of type-value pairs.

31.4.5 CSSM_X509_NAME

This structure contains a set of Relative Distinguished Names.

```
typedef struct cssm_x509_name {
    uint32 numberOfRDNs;
    CSSM_X509_RDN_PTR RelativeDistinguishedName;
} CSSM_X509_NAME, *CSSM_X509_NAME_PTR;
```

DESCRIPTION

numberOfRDNs
The number of Distinguished Names in this set.

RelativeDistinguishedName
A pointer to an array of Relative Distinguished Names.

31.4.6 CSSM_X509_SUBJECT_PUBLIC_KEY_INFO

This structure contains the public key and the description of the verification algorithm appropriate for use with this key.

```
/* Public key info struct */
typedef struct cssm_x509_subject_public_key_info {
    CSSM_X509_ALGORITHM_IDENTIFIER algorithm;
    CSSM_DATA subjectPublicKey;
} CSSM_X509_SUBJECT_PUBLIC_KEY_INFO, *CSSM_X509_SUBJECT_PUBLIC_KEY_INFO_PTR;
```

DESCRIPTION

algorithm
A substructure containing the algorithm id and input parameters for the algorithm.

SubjectPublicKey
The public key material in an industry standard representation appropriate for the keypair type.

31.4.7 CSSM_X509_TIME

Time is represented as a string according to the definitions of *GeneralizedTime* and *UTCTime* defined in RFC 2459.

```
typedef struct cssm_x509_time {
    CSSM_BER_TAG timeType;
    CSSM_DATA time;
} CSSM_X509_TIME, *CSSM_X509_TIME_PTR;
```

DESCRIPTION

timeType
A tag indicating the type of the time value.

time
The time value.

31.4.8 CSSM_X509_VALIDITY

```
/* Validity struct */
typedef struct x509_validity {
    CSSM_X509_TIME notBefore;
    CSSM_X509_TIME notAfter;
} CSSM_X509_VALIDITY, *CSSM_X509_VALIDITY_PTR;
```

DESCRIPTION

notBefore
A CSSM_X509_TIME indicating the beginning of the validity period for a certificate.

notAfter
A CSSM_X509_TIME indicating the end of the validity period for a certificate.

31.4.9 CSSM_X509_OPTION

This data type is used to indicate the presence or absence of an optional field value.

```
#define CSSM_X509_OPTION_PRESENT CSSM_TRUE
#define CSSM_X509_OPTION_NOT_PRESENT CSSM_FALSE
typedef CSSM_BOOL CSSM_X509_OPTION;
```

DESCRIPTION

CSSM_X509_OPTION_PRESENT
indicates the value is present

CSSM_X509_OPTION_NOT_PRESENT
indicates the value is not present

31.4.10 CSSM_X509EXT_BASICCONSTRAINTS

```
typedef struct cssm_x509ext_basicConstraints {
    CSSM_BOOL cA;
    CSSM_X509_OPTION pathLenConstraintPresent;
    uint32 pathLenConstraint;
} CSSM_X509EXT_BASICCONSTRAINTS, *CSSM_X509EXT_BASICCONSTRAINTS_PTR;
```

DESCRIPTION

cA
Indicates whether the certificate identifies a Certification Authority.

pathLenConstraintPresent
Indicates whether the optional *pathLenConstraint* value is present.

pathLenConstraint
An integer specifying the maximum number of certificates allowed in a verifiable certificate chain including this CA certificate.

31.4.11 CSSM_X509EXT_DATA_FORMAT

This list defines the valid formats for a certificate extension.

```
typedef enum extension_data_format {
    CSSM_X509_DATAFORMAT_ENCODED = 0,
    CSSM_X509_DATAFORMAT_PARSED,
    CSSM_X509_DATAFORMAT_PAIR,
} CSSM_X509EXT_DATA_FORMAT;
```

DESCRIPTION

CSSM_X509_DATAFORMAT_ENCODED
Indicates that the extension value is returned as a tag and BER encoded value.

CSSM_X509_DATAFORMAT_PARSED
Indicates that the extension value is in a parsed format associated with the X509 Extension OID For instance, the parsed representation of an extension with X509 Extension OID CSSMOID_X509ExtBasicConstraints is X509EXT_BASICCONTRAINTS.

CSSM_X509_DATAFORMAT_EXTPAIR
Indicates that the extension value is being returned in two representations, encoded and parsed.

31.4.12 CSSM_X509EXT_TAGandVALUE

This structure contains a BER/DER encoded extension value and the type of that value.

```
typedef struct cssm_x509_extensionTagAndValue {
    CSSM_BER_TAG type;
    CSSM_DATA value;
} CSSM_X509EXT_TAGandVALUE, *CSSM_X509EXT_TAGandVALUE_PTR;
```

DESCRIPTION

type
A DER tag indicating the type of the encoded value in the extension.

value
The encoded value stored in the extension.

31.4.13 CSSM_X509EXT_PAIR

This structure aggregates two extension representations: a tag and value, and a parsed X509 extension representation.

```
typedef struct cssm_x509ext_pair {
    CSSM_X509EXT_TAGandVALUE tagAndValue;
    void *parsedValue;
} CSSM_X509EXT_PAIR, *CSSM_X509EXT_PAIR_PTR;
```

DESCRIPTION

tagAndValue
A CSSM_X509EXT_TAGandVALUE structure.

parsedValue
A pointer to a parsed representation of the extension; the format of the data is determined based on the X509 extension OID specified.

31.4.14 CSSM_X509_EXTENSION

This structure contains a complete certificate extension.

```
/* Extension structure */
typedef struct cssm_x509_extension {
    CSSM_OID extnId;
    CSSM_BOOL critical;
    CSSM_X509EXT_DATA_FORMAT format;
    union cssm_x509ext_value {
        CSSM_X509EXT_TAGandVALUE *tagAndValue;
        void *parsedValue;
        CSSM_X509EXT_PAIR *valuePair;
    } value;
    CSSM_DATA BERvalue;
} CSSM_X509_EXTENSION, *CSSM_X509_EXTENSION_PTR;
```

DESCRIPTION

extnId
An OID uniquely naming the extension.

critical
A flag indicating whether the extension is critical. If an extension is critical, then the certificate can not be validly used by any application that does not "understand" the meaning of the extension and its contained value. If an extension is not critical, the certificate can be validly used by any application regardless of its knowledge and use of the extension.

value
A pointer to the extension value represented in the specified format.

BERvalue
A packed, BER/DER encoded representation of the extension value; the encoding includes the extension tag, length and value.

31.4.15 CSSM_X509_EXTENSIONS

This structure contains the set of all certificate extensions contained in a certificate.

```
typedef struct cssm_x509_extensions {
    uint32 numberOfExtensions;
    CSSM_X509_EXTENSION_PTR extensions;
} CSSM_X509_EXTENSIONS, *CSSM_X509_EXTENSIONS_PTR;
```

DESCRIPTION

numberOfExtensions
The number of extensions contained in this structure.

extensions
A pointer to a set of CSSM_X509_EXTENSION structures.

31.4.16 CSSM_X509_TBS_CERTIFICATE

This structure contains a complete X.509 certificate.

```
/* X509V3 certificate structure */
typedef struct cssm_x509_tbs_certificate {
    CSSM_DATA version;
    CSSM_DATA serialNumber;
    CSSM_X509_ALGORITHM_IDENTIFIER signature;
    CSSM_X509_NAME issuer;
    CSSM_X509_VALIDITY validity;
    CSSM_X509_NAME subject;
    CSSM_X509_SUBJECT_PUBLIC_KEY_INFO subjectPublicKeyInfo;
    CSSM_DATA issuerUniqueIdentifier;
    CSSM_DATA subjectUniqueIdentifier;
    CSSM_X509_EXTENSIONS extensions;
} CSSM_X509_TBS_CERTIFICATE, *CSSM_X509_TBS_CERTIFICATE_PTR;
```

DESCRIPTION

version
An optional value indicating whether the certificate is an X.509 V1 certificate an X.509 V2 certificate or an X.509 V3 certificate. The default version is X.509 V1.

serialNumber
The certificate serial number. The serial number with the issuer should form a unique identifier value for a certificate.

signature
A structure containing the the cryptographic algorithm identifier and an optional set of parameters to be used as input to that algorithm to computer the cryptographic structure over the other fields in the certificate.

issuer
A structure containing the Relative Distinguished Name of the entity who issued and signed the certificate.

validity
A structure containing the beginning and end date for valid use of this certificate.

subject
A structure containing the Relative Distinguished Name of the entity that is the subject of this certificate.

subjectPublicKeyInfo
A structure containing the public key of a public-private keypair owned by the certificate subject and the cryptographic algorithm identifier and an optional set of parameters to be used as input to that algorithm when using the public key.

issuerUniqueIdentifier
An optional unique identifier for the issuing entity. If *issuerUniqueIdentifier* is not specified, *issuerUniqueIdentifier.Length* = 0 and *issuerUniqueIdentifier.Data* = NULL.

subjectUniqueIdentifier
An optional unique identifier for the subject entity. If *subjectUniqueIdentifier* is not specified, *subjectUniqueIdentifier.Length* = 0 and *subjectUniqueIdentifier.Data* = NULL.

extensions
An optional set of CSSM_X509_EXTENSION certificate structures. If no extensions are specified, *extensions.numberOfExtensions* = 0.

31.4.17 CSSM_X509_SIGNATURE

This structure contains a cryptographic digital signature.

```
/* Signature structure */
typedef struct cssm_x509_signature {
    CSSM_X509_ALGORITHM_IDENTIFIER algorithmIdentifier;
    CSSM_DATA encrypted;
} CSSM_X509_SIGNATURE, *CSSM_X509_SIGNATURE_PTR;
```

DESCRIPTION

algorithmIdentifier
A structure containing a description of the signing algorithm used to create the digital signature. The signing algorithm indicates the verification algorithm required to verify the signature.

encrypted
The data generated by a signing operation.

31.4.18 CSSM_X509_SIGNED_CERTIFICATE

This structure associates a set of decoded certificate values with the signature covering those values.

```
/* Signed certificate structure */
typedef struct cssm_x509_signed_certificate {
    CSSM_X509_TBS_CERTIFICATE certificate;
    CSSM_X509_SIGNATURE signature;
} CSSM_X509_SIGNED_CERTIFICATE, *CSSM_X509_SIGNED_CERTIFICATE_PTR;
```

DESCRIPTION

certificate
A structure containing a decoded representation of an X.509 certificate.

signature
A structure containing the signature over the certificate.

31.4.19 CSSM_X509EXT_POLICYQUALIFIERINFO

```
typedef struct cssm_x509ext_policyQualifierInfo {
    CSSM_OID policyQualifierId;
    CSSM_DATA value;
} CSSM_X509EXT_POLICYQUALIFIERINFO, *CSSM_X509EXT_POLICYQUALIFIERINFO_PTR;
```

DESCRIPTION

policyQualifierId
An OID that uniquely identifies a policy.

value
The encoded policy qualifier value; encoding includes the tag and length.

31.4.20 CSSM_X509EXT_POLICYQUALIFIERS

```
typedef struct cssm_x509ext_policyQualifiers {
    uint32 numberOfPolicyQualifiers;
    CSSM_X509EXT_POLICYQUALIFIERINFO *policyQualifier;
} CSSM_X509EXT_POLICYQUALIFIERS, *CSSM_X509EXT_POLICYQUALIFIERS_PTR;
```

DESCRIPTION

numberOfPolicyQualifiers
The number of policy qualifiers.

policyQualifier
A pointer to an array of policy qualifier structures

31.4.21 CSSM_X509EXT_POLICYINFO

```
typedef struct cssm_x509ext_policyInfo {
    CSSM_OID policyIdentifier;
    CSSM_X509EXT_POLICYQUALIFIERS policyQualifiers;
} CSSM_X509EXT_POLICYINFO, *CSSM_X509EXT_POLICYINFO_PTR;
```

DESCRIPTION

policyIdentifier
An OID that uniquely identifies a policy.

policyQualifiers
A pointer to a structure that that indicates the policy qualifiers associated with the policy identifier.

31.5 Certificate OIDs and Certificate Data Structures

This section addresses the association between certificate OIDs and certificate data structures.

The certificate object identifiers indicate selected fields from an X.509 certificate. The object identifier is a required input parameter to "create" certificates, "get" certificate values out of the certificate, or "set" values for a certificate template (in anticipation of creating a certificate). Certificate creation functions accept input values as CSSM_FIELD structures. Each CSSM_FIELD structure contains an OID and a value. The value is contained in a CSSM_DATA structure. A CSSM_DATA structure contains a length and a pointer to the actual data value. The length indicates the number of bytes in the data value. The length is represented as a platform-dependent 32-bit unsigned integer. The data value referenced by the pointer is in one of three encoding: BER/DER, LDAP string or native, bushy C language structure.

The CSSM "get" functions accept an OID as input and return a single CSSM_DATA structure. The same use model is applied in this case.

The following table maps the object identifier for a selected set of certificate fields to the structure and format accepted as input by the "create" and "set" operations, and returned as output by the "get" operation.

Certificate OID Name	Structure and Format of the →Data entry of a CSSM_DATA structure
X509V3SignedCertificate	BER/DER-encoded CSSM_X509_SIGNED_CERTIFICATE structure
X509V3SignedCertificateCStruct	CSSM_X509_SIGNED_CERTIFICATE structure
X509V3TbsCertificate	BER/DER-encoded, CSSM_X509_TBS_CERTIFICATE structure
X509V3TbsCertificateCStruct	CSSM_X509_TBS_CERTIFICATE structure
X509V1Version	BER Integer
X509V1SerialNumber	BER Integer
X509V1IssuerName	BER/DER-encoded CSSM_X509_NAME structure
X509V1IssuerNameCStruct	CSSM_X509_NAME structure
X509V1IssuerNameLDAP	LDAP string structure
X509V1ValidityNotBefore	UTC Time string structure
X509V1ValidityNotAfter	UTC Time string structure
X509V1SubjectName	BER/DER-encoded CSSM_X509_NAME structure
X509V1SubjectNameCStruct	CSSM_X509_NAME structure
X509V1SubjectNameLDAP	LDAP string structure
CSSMKeyStruct	CSSM_KEY structure
X509V1SubjectPublicKeyCStruct	CSSM_X509_SUBJECT_PUBLIC_KEY_INFO structure
X509V1SubjectPublicKeyAlgorithm	Algorithm OID
X509V1SubjectPublicKeyAlgorithmParameters	BER/DER-encoded parameters
X509V1SubjectPublicKey	Byte string
X509V1CertificateIssuerUniqueId	Byte string
X509V1CertificateSubjectUniqueId	Byte string
X509V3CertificateExtensionsStruct	BER/DER-encoded CSSM_X509_EXTENSIONS structure
X509V3CertificateExtensionsCStruct	CSSM_X509_EXTENSIONS structure
X509V3CertificateNumberOfExtensions	Platform-dependent integer
X509V3CertificateExtensionStruct	BER/DER-encoded CSSM_X509_EXTENSION structure
X509V3CertificateExtensionCStruct	CSSM_X509_EXTENSION structure
X509V3CertificateExtensionId	Extension OID
X509V3CertificateExtensionCritical	CSSM_BOOL value

X509V3CertificateExtensionType	CL_DER_TAG_TYPE
X509V3CertificateExtensionValue	Byte string
Certificate Extension OIDs	CSSM_X509_EXTENSION structure for the extension with the specified Certificate Extension OID

Signature OID Names	**Structure and Format of the →Data entry of a CSSM_DATA structure**
X509V1SignatureStruct	BER/DER-encoded CSSM_X509_SIGNATURE structure
X509V1SignatureCStruct	CSSM_X509_SIGNATURE structure
X509V1SignatureAlgorithm	Algorithm OID
X509V1SignatureAlgorithmParameters	BER/DER encoded parameters
X509V1Signature	Byte string

Chapter 32

OIDs for X.509 Certificate Revocation Lists

32.1 Base Object Identifiers

The following object identifiers define names for single fields or sets of related fields in an X.509 certificate revocation list (CRL). Each object identifier also indicates the representation for the selected field or fields. Possible representations include:

- DER encoded value
- C language structure with values in native platform representation
- LDAP String value

Object identifiers are defined corresponding to the CRL fields defined by the X.509 V2 standard. One primary subarc is defined for this purpose:

INTEL_X509V2_CRL_R08
  OBJECT IDENTIFIER ::= { INTEL_SEC_FORMATS, 2, 1}

32.2 Programmatic Definition of Base Object Identifiers

Programmatically these object identifiers are defined by the following constants.

```
#define INTEL_X509V2_CRL_R08         INTEL_SEC_FORMATS,   2, 1
#define INTEL_X509V2_CRL_R08_LENGTH  INTEL_SEC_FORMATS_LENGTH+2
```

32.3 Object Identifiers for Fields

32.3.1 CRL OIDs

X509V2CRLSignedCrlStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 0 }

X509V2CRLSignedCrlCStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 0, INTEL_X509_C_DATATYPE }

X509V2CRLTbsCertListStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 1 }

X509V2CRLTbsCertListCStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 1, INTEL_X509_C_DATATYPE }

X509V2CRLVersion
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 2 }

X509V1CRLIssuerStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 3 }

X509V1CRLIssuerNameCStruct
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 3, INTEL_X509_C_DATATYPE }

X509V1CRLIssuerNameLDAP
  OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 3, INTEL_X509_LDAPSTRING_DATATYPE }

X509V1CRLThisUpdate
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 4 }

X509V1CRLNextUpdate
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 5 }

32.3.2 CRL Entry (CRL CertList) OIDs

X509V1CRLRevokedCertificatesStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 7 }

X509V1CRLRevokedCertificatesCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 7, INTEL_X509_C_DATATYPE }

X509V1CRLNumberOfRevokedCertEntries
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 6 }

X509V1CRLRevokedEntryStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 15 }

X509V1CRLRevokedEntryCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 15, INTEL_X509_C_DATATYPE }

X509V1CRLRevokedEntrySerialNumber
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 16 }

X509V1CRLRevokedEntryRevocationDate
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 17 }

32.3.3 CRL Entry (CRL CertList) Extension OIDs

X509V2CRLRevokedEntryAllExtensionsStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 18 }

X509V2CRLRevokedEntryAllExtensionsCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 18, INTEL_X509_C_DATATYPE }

X509V2CRLRevokedEntryNumberOfExtensions
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 20 }

X509V2CRLRevokedEntrySingleExtensionStruct
OBJECT IDENTIFIER ::{ INTEL_X509V2_CRL_R08, 19 }

X509V2CRLRevokedEntrySingleExtensionCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 19, INTEL_X509_C_DATATYPE }

X509V2CRLRevokedEntryExtensionId
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 21 }

X509V2CRLRevokedEntryExtensionCritical
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 22 }

X509V2CRLRevokedEntryExtensionType
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 23 }

X509V2CRLRevokedEntryExtensionValue
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 24 }

32.3.4 CRL Extension OIDs

X509V2CRLAllExtensionsStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 8 }

X509V2CRLAllExtensionsCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 8, INTEL_X509_C_DATATYPE }

X509V2CRLNumberOfExtensions
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 10 }

X509V2CRLSingleExtensionStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 9 }

X509V2CRLSingleExtensionCStruct
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 9, INTEL_X509_C_DATATYPE }

X509V2CRLExtensionId
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 11 }

X509V2CRLExtensionCritical
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 12 }

X509V2CRLExtensionType
OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 13 }

X509V2CRLExtensionValue OBJECT IDENTIFIER ::= { INTEL_X509V2_CRL_R08, 14 }

32.4 C Language Data Structures for X.509 CRLs

32.4.1 CSSM_X509_REVOKED_CERT_ENTRY

This structure contains a single entry in a certificate revocation list.

```
/* x509V2 entry in the CRL revokedCertificates sequence */
typedef struct cssm_x509_revoked_cert_entry {
    CSSM_DATA certificateSerialNumber;
    CSSM_X509_TIME revocationDate;
    CSSM_X509_EXTENSIONS extensions;
} CSSM_X509_REVOKED_CERT_ENTRY, *CSSM_X509_REVOKED_CERT_ENTRY_PTR;
```

DESCRIPTION

certificateSerialNumber

revocationDate
The date on which revocation occurred.

extensions
Optional sequence of CRL extensions. If no extensions are specified, extensions.Length = 0 and extensions.Data = NULL.

32.4.2 CSSM_X509_REVOKED_CERT_LIST

This structure defines an unordered linked list containing certificate revocation nodes. This structure aggregates the records describing revoked certificates.

```
typedef struct cssm_x509_revoked_cert_list {
    uint32 numberOfRevokedCertEntries;
    CSSM_X509_REVOKED_CERT_ENTRY_PTR revokedCertEntry;
} CSSM_X509_REVOKED_CERT_LIST, *CSSM_X509_REVOKED_CERT_LIST_PTR;
```

DESCRIPTION

numberOfRevokedCertEntries
Number of revoked certificates in the linked list.

revokedCertEntry
A pointer to the CRL entry describing a revoked sertificate.

32.4.3 CSSM_X509_TBS_CERTLIST

This structure defines a complete, but unsigned certificate revocation list. This includes the header information describing the CRL, and the list of CRL entries identifying the revoked certificates and describing the circumstances of the revocation operation.

```
/* x509v2 Certificate Revocation List (CRL) (unsigned) structure */
typedef struct cssm_x509_tbs_certlist {
    CSSM_DATA version;
    CSSM_X509_ALGORITHM_IDENTIFIER signature;
    CSSM_X509_NAME issuer;
    CSSM_X509_TIME thisUpdate;
    CSSM_X509_TIME nextUpdate;
    CSSM_X509_REVOKED_CERT_LIST_PTR revokedCertificates;
```

```
    CSSM_X509_EXTENSIONS extensions;
} CSSM_X509_TBS_CERTLIST, *CSSM_X509_TBS_CERTLIST_PTR;
```

DESCRIPTION

version
A BER Integer indicating the CRL version.

signature
A structure specifying the cryptographic signaturing algorithm and optional parameters that will be used to sign the CRL. This value may be NULL.

issuer
A structure containing relative distinguished name components that form the issuer name.

thisUpdate
The issue-date for this CRL.

nextUpdate
The planned date for issuing the next CRL.

revokedCertificates
A linked list of revoked certificate nodes.

extensions
An optional structure containing extension data that further describes the CRL. If no extensions are specified, *extensions.Length* = 0 and = NULL.

32.4.4 CSSM_X509_SIGNED_CRL

This structure aggregates an unsigned CRL and a signature over that CRL.

```
typedef struct cssm_x509_signed_crl {
    CSSM_X509_TBS_CERTLIST tbsCertList;
    CSSM_X509_SIGNATURE signature;
} CSSM_X509_SIGNED_CRL, *CSSM_X509_SIGNED_CRL_PTR;
```

DESCRIPTION

tbsCertList
An unsigned structure containing header information describing the CRL and the list of revoked certificates .

signature
A structure containing the signature algorithm and parameters used to sign the *tbsCerlist* and the digital signature generated by that algorithm.

32.5 Associating CRL OIDs and CRL Data Structures

The CRL object identifiers indicate selected fields from an X.509 CRL. The object identifier is a required input parameter to "create" CRLs, "get" field values out of the CRL, or "set" values for a CRL template (in anticipation of creating a CRL). CRL creation functions accept input values as CSSM_FIELD structures. Each CSSM_FIELD structure contains an OID and a value. The value is contained in a CSSM_DATA structure. A CSSM_DATA structure contains a length and a pointer to the actual data value. The length indicates the number of bytes in the data value. The length is represented as a platform-dependent 32-bit unsigned integer. The data value referenced by the pointer is in one of three encoding: BER/DER, LDAP string or native, bushy C language structure.

The CSSM "get" functions accept an OID as input and return a single CSSM_DATA structure. The same use model is applied in this case.

The following table maps the object identifier for a selected set of CRL fields to the structure and format accepted as input by the "create" and "set" operations, and returned as output by the "get" operation.

CRL OID Names	Structure and Format of the ->Data entry of a CSSM_DATA structure
X509V2CRLSignedCrlStruct	BER/DER-encoded CSSM_X509_SIGNED_CRL structure
X509V2CRLSignedCrlCStruct	CSSM_X509_SIGNED_CRL structure
X509V2CRLTbsCertListStruct	BER/DER-encoded CSSM_X509_TBS_CERTLIST structure
X509V2CRLTbsCertListCStruct	CSSM_X509_TBS_CERTLIST structure
X509V2CRLVersion	BER Integer
X509V1CRLIssuerStruct	BER/DER-encoded CSSM_X509_NAME structure
X509V1CRLIssuerNameCStruct	CSSM_X509_NAME structure
X509V1CRLIssuerNameLDAP	LDAP string
X509V1CRLThisUpdate	UTC Time string
X509V1CRLNextUpdate	UTC Time string
X509V1CRLRevokedCertificatesStruct	BER/DER-encoded CSSM_X509_REVOKED_CERT_LIST structure
X509V1CRLRevokedCertificatesCStruct	CSSM_X509_REVOKED_CERT_LIST structure
X509V1CRLNumberOfRevokedCertEntries	Platform-dependent integer
X509V1CRLRevokedEntryStruct	BER/DER-encoded CSSM_X509_REVOKED_CERT_ENTRY structure
X509V1CRLRevokedEntryCStruct	CSSM_X509_REVOKED_CERT_ENTRY structure
X509V1CRLRevokedEntrySerialNumber	BER Integer
X509V1CRLRevokedEntryRevocationDate	UTC Time string
X509V2CRLRevokedEntryAllExtensionsStruct	BER/DER-encoded CSSM_X509_EXTENSIONS structure
X509V2CRLRevokedEntryAllExtensionsCStruct	CSSM_X509_EXTENSIONS structure
X509V2CRLRevokedEntryNumberOfExtensions	Platform-dependent integer
X509V2CRLRevokedEntrySingleExtensionStruct	BER/DER-encoded CSSM_X509_EXTENSION structure
X509V2CRLRevokedEntrySingleExtensionCStruct	CSSM_X509_EXTENSION structure
X509V2CRLRevokedEntryExtensionId	Extension OID
X509V2CRLRevokedEntryExtensionCritical	CSSM_BOOL
X509V2CRLRevokedEntryExtensionType	CL_DER_TAG_TYPE
X509V2CRLRevokedEntryExtensionValue	Byte string
X509V2CRLAllExtensionsStruct	BER/DER-encoded CSSM_X509_EXTENSIONS structure
X509V2CRLAllExtensionsCStruct	CSSM_X509_EXTENSIONS structure
X509V2CRLNumberOfExtensions	Platform-dependent integer

X509V2CRLSingleExtensionStruct	BER/DER-encoded CSSM_X509_EXTENSION structure
X509V2CRLSingleExtensionCStruct	CSSM_X509_EXTENSION structure
X509V2CRLExtensionId	Extension OID
X509V2CRLExtensionCritical	CSSM_BOOL
X509V2CRLExtensionType	CL_DER_TAG_TYPE
X509V2CRLExtensionValue	Byte string

Technical Standard

Part 13:

CSSM Elective Module Manager (EMM)

The Open Group

Chapter 33

Introduction

CDSA defines an interoperable, extensible architecture in which applications can selectively and dynamically access security services. The architecture is extensible in 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 39-1 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. Five basic types of module managers are defined:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Authorization Computation 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.

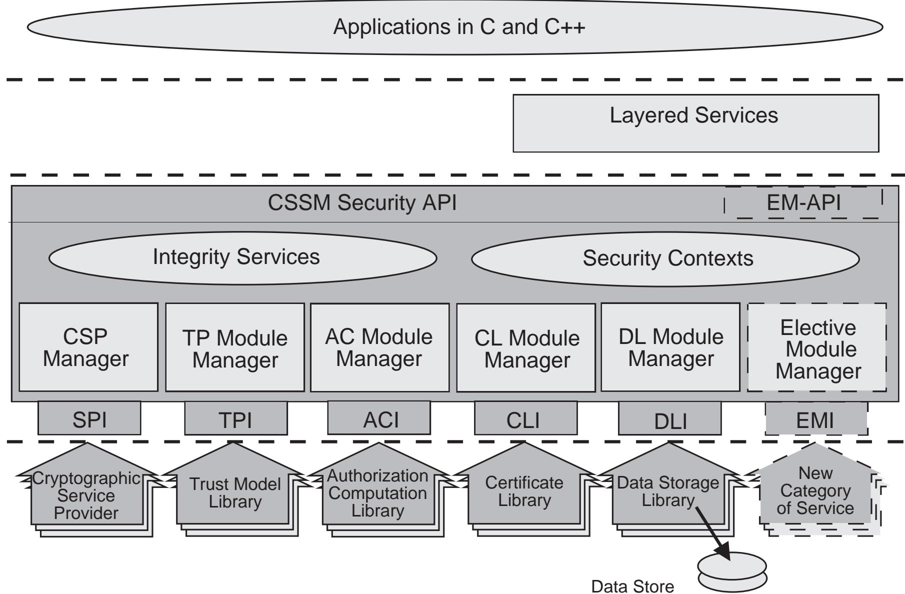


Figure 33-1 Common Data Security Architecture for all Platforms

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 runtime selection of security service modules.

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.

CSSM, add-in modules, elective module managers, and optionally applications verify the identity and integrity of components of CDSA. CSSM checks dynamic components as they are added to the system. These components include elective module managers, add-in service modules, applications, and CSSM itself.

Chapter 34

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. One elective module manager should support only one service type.

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. The module manager for key recovery could be loaded on demand as required but need not be a static part of a system. User authentication based on biometric data can be an elective service. Systems where biometric devices are deployed and required only for authenticating system administrators would not need a biometric module manager as a standard, loaded component on the environment. Elective module management supports on-demand inclusion of a module manager. The module manager defines a new set of 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.

34.1 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_ModuleLoad*() and *CSSM_ModuleAttach*() to select an instance of the add-in service provider. Figure 40-1 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 (dynamically loaded) module manager versus a basic module manager (which is always resident in the system).

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 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 dependent 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.

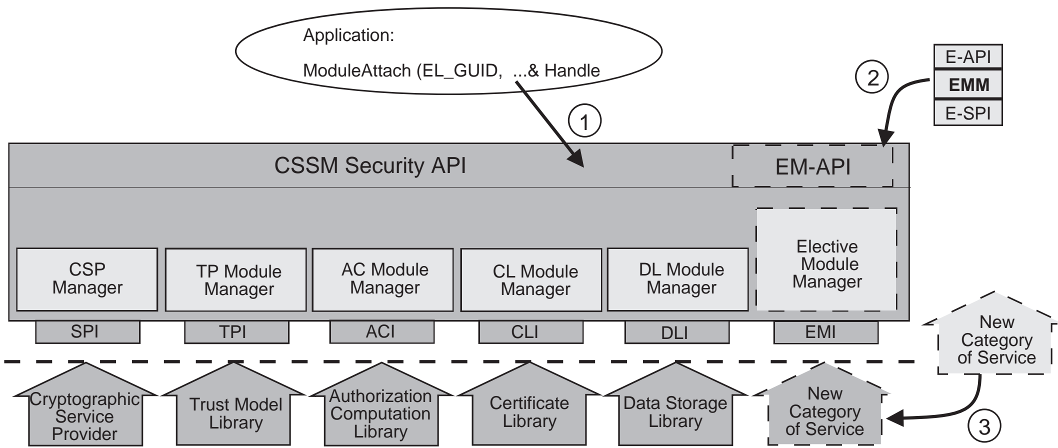


Figure 34-1 Steps to Load a Service Module and its Corresponding EMM

34.2 Registering Module Managers

Elective module managers are installed using platform-specific installation services. During installation the EMM installation program must registered the EMM with CSSM using the Module Directory Services (MDS). MDS defines the EMM relation that stores basic information about the elective module manager. The EMM relation can be queried by anyone, but typically only system administration applications and CSSM will use registry information about module managers. For example, a smart installer for a service module should 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 service module. CSSM uses the EMM relation to obtain the module manager's signed manifest credentials.

34.3 Interaction with CSSM

CSSM supports elective module managers with a number of EMM-specific services. These service functions are defined as upcalls to CSSM. The upcalls support the elective module manager in:

- Obtaining information about a session between an application and a service module under the management of the EMM
- Altering state information pertaining to a session between an application and a service module under the management of the EMM.

CSSM provides this information so CSSM and the elective module manager can effectively support the service module without the module being aware that the elective module manager was dynamically loaded and that CSSM had no prior knowledge of the structures, functions, and services provided by the service module.

34.4 Integrity and Secure Linkage

In general, CSSM provides secure linkage checks between applications and CSSM, and between service modules and CSSM. To perform this service on behalf of an EMM, CSSM must proxy all function calls into and out of the EMM. For this reason, it is inappropriate for the elective module manager to invoke functions other than those functions CSSM has registered with the EMM. This includes application callback functions and service module interfaces via the SPI function table.

CSSM provides upcall proxies to the elective module manager. When performing secure linkage checks, CSSM is aware of the EMM and verifies that the upcall originated from the service provider module or the EMM before passing the upcall on to the application.

To ensure integrity of the execution environment, CDSA recommended that applications and service modules perform secure linkage checks of CSSM. Elective module managers pose a problem for these checks. The EMM's dynamic nature is transparent to the application and to the service module, hence the EMM not included in the CSSM authentication and secure linkage checks performed by the application or the service provider during *CSSM_ModuleAttach*() processing. If applications and service modules must perform secure linkage checks on every invocation, then EMM transparency may not be possible. To achieve complete cover for linkage checks, applications and service modules should use Module Directory Service (MDS) facilities to locate the EMM module and perform cross-check. The verified address range for the EMM can be added to the verified address range of CSSM for the purpose of secure linkage checks.

34.5 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:

- 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 35

Administration of Elective Module Managers

35.1 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 defines 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. This in-memory verification prevents ***stealth*** attacks where the disk-resident object file is unaltered, but the loaded code is tampered. Additionally, a module manager should verify 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.

35.2 Module Manager Credentials

Integrity verification is based on the module manager's signed manifest 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. Signed manifest credentials consist of three sub-blocks:

- A manifest block
- A signer's information block
- A signature block.

The manifest block contains:

- (Optionally) Descriptive attributes of the elective module manager
- A hash of each executable software component of the elective module manager
- (Optionally) A reference to each separately link-able software component comprising the elective module manager.

These three sub-blocks form a single set of credentials. The credentials are stored as an attribute of the module manager. All module manager attributes are stored in relations managed by MDS. The module manager's manifest is stored as a binary blob representation of the manifest file in the MDS EMM relation.

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 41-1. 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 elective module manager 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, not 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 35-1 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 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
- (Optionally) A reference to each object code file that is part of the module manager implementation.

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 of manifests and certificate credentials is specified in the **CDSA Signed Manifests** specification Part of this document.

35.3 Installing an Elective Module Manager

Before an application can use an elective service through CSSM, the elective module manager and at least one elective service module must be installed on the local system and registered with CDSA. The installation program must use platform-specific services to install the module manager on the local system. The installation program must use Module Directory Services (MDS) to register the EMM with CDSA. The installation program must create at least one record in the MDS *EMM* relation. To insert new records, the installation program must have write-access to the MDS CDSA Directory database. MDS controls write-access to all MDS databases. Write-access is granted only to signed, authorized applications. The installation program must be a signed application presenting a signed manifest credential. The program's credential must:

- be signed by an MDS-recognized trusted party
- include the authorization attribute
  [Name: CDSA_ACCESS_TAG, Value: CSSM_DB_ACCESS_WRITE]

The installation program presents its signed manifest to MDS during *DbOpen*. If MDS successfully verifies the installation program's signed manifest credential, then write-access is granted and the program can proceed to insert new records into the MDS CDSA Directory database.

The MDS *EMM* relation defines the following attributes to describe elective module managers:

- The module manager's globally-unique identifier (GUID)
- A logical/descriptive module manager name
- The vendor providing the EMM
- The module manager's manifest
- File system reference to locate the module manager's executable code

MDS also defines the EMM Service Provider relation, which is similar to the MDS *Common* relation. The EMM Service Provider relation stores the general attributes applicable to all elective service modules that are managed by some elective module manager registered with CDSA. Attributes in the EMM Service Provider relation include:

- The module's globally-unique identifier (GUID)
- A logical/descriptive module name
- The module's service type
- The module's manifest
- File system reference to locate the module's executable code.

The EMM service provider attributes are very general. The vendor providing the Elective Module Manager may wish to add new relations to the MDS CDSA Directory database to store more detailed descriptions of the services and capabilities of the service modules providing the new service. These new relations should be created during module manager installation. The schema for these relations is defined by the vendor of the new elective service category.

To create new relations, the installation program must have schema-modification-rights to the MDS CDSA Directory database. MDS controls schema-modification-rights to all MDS databases. Schema-modification-rights are granted only to signed, authorized applications. The installation program must be a signed application presenting a signed manifest credential. The program's credential must:

- be signed by an MDS-recognized trusted party
- include the authorization attribute
  [Name: CDSA_ACCESS_TAG, Value: CSSM_DB_ACCESS_PRIVILEGED]

35.3.1 Global Unique Identifiers (GUIDs)

Each module manager must have a globally-unique identifier (GUID) that CSSM and the module manager itself use to uniquely identify the manager for integrity verification operations.

GUID generators are publicly available for Windows™ 95, Windows NT™, and on many UNIX™ platforms.

35.4 Loading an Elective Module Manager

Before an application can use the functions of a specific add-in module, it must use the *CSSM_ModuleLoad*() and *CSSM_ModuleAttach*() functions 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 Module Directory Services EMM relation 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 invokes the module manager interface *ModuleManagerAuthenticate*(). This function should perform integrity self-check using EISL services. The function should also verify CSSM based on its signed manifest credentials. This completes the second half of the CSSM bilateral authentication protocol.
3. Upon successful completion of the bilateral authentication protocol, the elective module manager returns a small table of function pointers that CSSM uses to invoke the elective module manager.
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.

5. CSSM invokes the module manager function *RegisterDispatchTable*( ) to provide the elective module manager with a function table for accessing the CSSM services.
6. CSSM completes module attach processing on behalf of the EMM by invoking the service module's interface *CSSM_SPI_ModuleAttach*( ). The service module returns an SPI function table to CSSM as a result of successful completion of this function.
7. CSSM invokes the elective module manager interface *RefreshFunctionTable*( ), requesting that the EMM initialize the function name to procedure address mapping for return to the calling application. These function addresses should correspond to procedures within the EMM. These procedures implement the mapping of APIs to SPIs.
8. CSSM returns a module handle and the function name to procedure address mapping to the application that invoked the *CSSM_ModuleAttach*( ) operation.

35.4.1 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. If verification fails, then either the CSSM has been tampered with 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 return the interface function table to the suspect CSSM. The module manager should perform clean-up operations and exit voluntarily. The module manager has refused to provide service in an environment that it could not verify.

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 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.

CSSM and the elective module manager must use 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.

35.4.2 Protocol for Attaching a Service Module

CSSM and an EMM perform the same protocol for each *CSSM_ModuleAttach*() call issued by an application. The first call to *CSSM_ModuleAttach*() causes the EMM to be verified and loaded. Subsequent calls do not repeat the verification and load process, but all other steps are the same for every application call to *CSSM_ModuleAttach*().

For every module attach of a service provider managed by an EMM, the protocol proceeds as follows:

1. CSSM invokes *ModuleManagerAuthenticate*(). Upon successful completion of this function, the elective module manager returns its function table to CSSM. The function table contains module management function pointers to *Initialize*, *Terminate*, *RegisterDispatchTable*, *DeregisterDispatchTable*, and *EventNotifyManager*.
2. Using this function table, CSSM invokes the EMM *Initialize*() function. The EMM performs all service-specific initialization.
3. CSSM invokes the EMM *RegisterDispatchTable*() function. This function provides the EMM with a set of CSSM service functions that the module manager can use to obtain information about the attaching service module and the application that loaded and attached the module. Information accessible through these functions includes:
   - A service function table supplied by the service module
   - A memory-management function table supplied by the application
   - A callback function provided by the application
4. CSSM invokes the *CSSM_SPI_ModuleAttach*() interface of the service module to complete attach-processing on behalf of the EMM. The service module can perform verification of the CSSM and the calling application (particularly to verify the application's request for any use exemptions). The primary result returned from the service provider to CSSM is the SPI function table containing the entry points for the service operations implemented by the service module.
5. CSSM invokes the EMM *RefreshFunctionTable*() interface. In response, EMM fills in the API function addresses for the names in the *FuncNameAddr* table.
6. CSSM returns the name-to-address mapping table for the APIs to the calling application, completing the module attach operation.

The CSSM service functions provided to the elective module manager via the *RegisterDispatchTable*() function are used during the life of the attach session to support information exchange between CSSM and the EMM. CSSM-provided services include:

- Performing secure linkage checks between the application and the service module
- Signaling CSSM of early termination of the elective module manager (typically due to an error condition)
- Providing indirect access to information and services from the application.

The application services available through CSSM include use of the application's memory management functions and access to an application's callback function.

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 must be obtained through

CSSM and used by the elective module manager.

The functions are provided as a set of memory management upcalls. The functions are the application's equivalent of malloc, free, calloc, and realloc. 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.

35.4.3 Protocol for Detaching a Service Module

When an application invokes *CSSM_ModuleDetach*( ), CSSM follows the same protocol for managing every module detach operation:

1. CSSM detaches the service module by invoking *ModuleDetach*( ).
2. If the detached service provider was managed by an EMM, CSSM calls the *DeregisterDispatchTable*( ) interface of the affected EMM.

35.4.4 Protocol for Unloading a Service Module

When an application invokes *CSSM_ModuleUnload*( ), CSSM follows the same protocol for managing every module unload operation:

1. CSSM unloads the service module by invoking *CSSM_ModuleUnload*( ).
2. If the detached service provider was managed by an EMM, CSSM decrements a load-counter. If the load-counter is zero, then CSSM calls the *Terminate* interface of the affected EMM. The EMM should clean-up all system state and prepare to be unloaded.
3. Upon return from the EMM *Terminate* function, CSSM unloads the terminated EMM.

Chapter 36

Elective Module Manager Operations

36.1 Data Structures

36.1.1 CSSM_STATE_FUNCS

This structure is used by CSSM to provide function pointers to an elective module manager. An EMM uses these pointers to access services provided by the CSSM.

```
#define CSSM_HINT_CALLBACK (1)
typedef struct cssm_state_funcs {
    CSSM_RETURN (CSSMAPI *cssm_GetAttachFunctions)
        (CSSM_MODULE_HANDLE hAddIn,
        CSSM_SERVICE_MASK AddinType,
        void **SPFunctions,
        CSSM_GUID_PTR Guid);
    CSSM_RETURN (CSSMAPI *cssm_ReleaseAttachFunctions)
        (CSSM_MODULE_HANDLE hAddIn);
    CSSM_RETURN (CSSMAPI * cssm_GetAppMemoryFunctions)
        (CSSM_MODULE_HANDLE hAddIn,
        CSSM_UPCALLS_PTR UpcallTable);
    CSSM_RETURN (CSSMAPI * cssm_IsFuncCallValid)
        (CSSM_MODULE_HANDLE hAddin,
        CSSM_PROC_ADDR SrcAddress,
        CSSM_PROC_ADDR DestAddress,
        CSSM_PRIVILEGE InPriv,
        CSSM_PRIVILEGE *OutPriv,
        CSSM_BITMASK Hints,
        CSSM_BOOL* IsOK);
    CSSM_RETURN (CSSMAPI * cssm_DeregisterManagerServices)
        (const CSSM_GUID *GUID);
    CSSM_RETURN (CSSMAPI * cssm_DeliverModuleManagerEvent)
        (const CSSM_MANAGER_EVENT_NOTIFICATION *EventDescription);
} CSSM_STATE_FUNCS, *CSSM_STATE_FUNCS_PTR;
```

36.1.2 CSSM_MANAGER_EVENT_TYPES

This list defines a minimal set of event types used by communicating module managers. It is assumed that module managers may define their own protocols for interacting with other managers. Request and Reply messages are generic message categories to support simple message exchange protocols.

```
typedef uint32 CSSM_MANAGER_EVENT_TYPES;

#define CSSM_MANAGER_SERVICE_REQUEST 1
#define CSSM_MANAGER_REPLY 2
```

36.1.3 CSSM_MANAGER_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 DestinationModuleManagerType;
    CSSM_SERVICE_MASK SourceModuleManagerType;
    CSSM_MANAGER_EVENT_TYPES Event;
    uint32 EventId;
    CSSM_DATA 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 results from this event notification.

EventData
Arbitrary data (required or information) for this event.

36.1.4 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_STATE_FUNCS_PTR CssmStateCallTable);
    CSSM_RETURN (CSSMAPI *DeregisterDispatchTable) (void);
    CSSM_RETURN (CSSMAPI *EventNotifyManager)
        (const CSSM_MANAGER_EVENT_NOTIFICATION *EventDescription);
    CSSM_RETURN (CSSMAPI *RefreshFunctionTable)
        (CSSM_FUNC_NAME_ADDR_PTR FuncNameAddrPtr,
        uint32 NumOfFuncNameAddr);
} 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 CSSM function table to an elective module manager. The elective module manager must use these functions to obtain information about the service module and calling application, and to initiate state changes in the application and service module session.

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 CSSM function table for that add-in module should not be used.

EventNotifyManager
Function invoked by CSSM forwarding an event notification from one module manager to another target module manager.

RefreshFunctionTable
Function invoked by CSSM to obtain an initialized API function table corresponding to the service interfaces implemented by a particular EMM service provider module. CSSM forwards this function table to the application that invoked the *ModuleAttach* operation.

36.1.5 CSSM_HINT_xxx Parameter

These are hints that help CSSM to look for the state information about integrity and privilege.

```
#define CSSM_HINT_NONE (0)
#define CSSM_HINT_ADDRESS_APP (1)
#define CSSM_HINT_ADDRESS_SP (2)
```

36.2 Elective Module Manager Functions

The man-page definitions for Elective Module Manager functions, accessible only to CSSM, are presented in this section.

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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See also General Error Codes and Common Error Codes and Values.

CSSMERR_CSSM_MODULE_MANAGER_INITIALIZE_FAIL.

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_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See also the general error codes and common error code and values section.

CSSMERR_CSSM_EMM_AUTHENTICATE_FAILED.

SEE ALSO

Initialize()

NAME

ModuleManagerAuthenticate

SYNOPSIS

```
CSSM_RETURN CSSMAPI ModuleManagerAuthenticate
    (CSSM_KEY_HIERARCHY KeyHierarchy,
    const CSSM_GUID *CssmGuid,
    const CSSM_GUID *AppGuid,
    CSSM_MANAGER_REGISTRATION_INFO_PTR FunctionTable)
```

DESCRIPTION

This function should perform the elective module manager's half of the bilateral authentication procedure with CSSM. The *CssmGuid* is used to locate the CSSM's credentials to be verified. The credentials are a zipped, signed manifest.

The *KeyHierarchy* indicates which public key should be used as the root when checking the integrity of the module manager. The *AppGuid* is used to locate the application's signed manifest credentials. The elective module manager must check the application's credentials to verify the application's authorization. If no privileges are requested, then the application is not required to provide a GUID nor a set of signed manifest credentials.

Upon successful completion, the elective module manager returns its function table to the calling CSSM. The EMM function table contains the set of EMM entry points that CSSM uses to notify the module manager of significant events such as module attach and module detach requests issued by an application, and event notifications issued by other module managers.

This function symbol must be exported by the elective module manager, so CSSM can invoke this function upon completion of the loading process.

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

KeyHierarchy (input)
: The CSSM_KEY_HIERARCHY flag indicating which embedded key(s) CSSM should use when verifying the integrity of the module manager.

CssmGuid (input)
: A CSSM_GUID value identifying the calling CSSM. The elective module manager can use this value to locate the signed manifest credentials for CSSM.

AppGuid (input/optional)
: A CSSM_GUID value identifying the application who invoked the calling CSSM. The elective module manager can use this value to locate the signed manifest credentials for the application.

FunctionTable (output)
: A set of function pointers for EMM-defined functions used by CSSM to communicate state changes related to module attach and module detach operations.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

NAME

RegisterDispatchTable

SYNOPSIS

```
CSSM_RETURN CSSMAPI RegisterDispatchTable
    (CSSM_STATE_FUNCS_PTR CssmStateCallTable)
```

DESCRIPTION

This EMM-defined function is invoked by CSSM once for each *CSSM_ModuleAttach*() operation requesting a service provider of the type managed by the EMM. CSSM uses this function to provide the EMM with a set of CSSM function pointers. The EMM invokes these functions at anytime during the life cycle of the attach-session to obtain information about the current state and to modify the current state of the attach session.

When the attach-session is terminated, CSSM informs the module manager by invoking the EMM function *DeregisterDispatchTable*(). The corresponding set of CSSM state functions become invalid at that time.

PARAMETERS

CssmStateCallTable (input)
: A table of function pointers for the set of CSSM-defined functions the elective module manager can use to query and control the state of an attach-session between an application and a service provider managed by the module manager.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the general error codes and common error code and values section.

SEE ALSO

DeregisterDispatchTable()

NAME

DeregisterDispatchTable

SYNOPSIS

```
CSSM_RETURN (CSSMAPI DeregisterDispatchTable)
    (void)
```

DESCRIPTION

This EMM-defined function is invoked by CSSM once for each *CSSM_ModuleDetach*( ) operation issued against a service provider of the type managed by the EMM. CSSM uses this function to inform the EMM that the set of CSSM function pointers provided to the EMM when the session was attached are no longer valid.

PARAMETERS

None.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the general error codes and common error code and values section.

SEE ALSO

RegisterDispatchTable( )

NAME

EventNotifyManager

SYNOPSIS

```
CSSM_RETURN CSSMAPI EventNotifyManager
   (const CSSM_MANAGER_EVENT_NOTIFICATION *EventDescription)
```

DESCRIPTION

This function receives an event notification from another module manager. The source manager 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 use 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)
The unique service mask identifying the destination module manager.

SourceModuleManagerType (input)
The unique service mask identifying the source module manager.

Event (input)
An identifier indicating the event that has or will take place.

EventId (input/optional)
A unique identifier associated with this event notification. It must be used in any reply notification that results from this event notification.

EventData (input/optional)
Arbitrary data (required or informational) for this event.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See also the general error codes and common error codes and values section.

CSSMERR_CSSM_MODULE_MANAGER_NOT_FOUND.

NAME

RefreshFunctionTable

SYNOPSIS

```
CSSM_RETURN CSSMAPI RefreshFunctionTable
    (CSSM_FUNC_NAME_ADDR_PTR FuncNameAddrPtr,
    uint32 NumOfFuncNameAddr)
```

DESCRIPTION

CSSM invokes this function to obtain the EMM-defined API function. The table is returned to CSSM in *FuncNameAddrPtr* and CSSM returns the table to the application. The application uses this table to invoke the security services defined by the EMM's service category. CSSM must obtain and forward the API table to the application on behalf of the EMM because the application is not aware of the optional nature of the EMM. Applications use CSSM to obtain the API function table for basic module managers and elective module managers, providing a uniform application programming model.

If the Elective Module Manager needs the service provider's SPI function table in order to initialize the API function table, the EMM can obtain the SPI function table by invoking the CSSM-provided service *cssm_GetAttachFunctions*(). The service module may implement only a subset of the defined functions and the EMM may wish to manage these functions in a particular manner through the API mapping. The elective module manager uses the SPI function table to dispatch application calls for service to attached modules.

Multiple applications and multiple instances of a service module can be concurrently active. The single elective module manager is responsible for managing all of these concurrent sessions. After completing initialization of the API function table, the EMM returns the refreshed API table to CSSM.

PARAMETERS

FuncNameAddrPtr (input/output)
A pointer to a table mapping function names to EMM-defined APIs.

NumOfFuncNameAddr (input)
The number of entries in the table referenced by *FuncNameAddrPtr*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See feneral and common error codes and error values section.

36.3 CSSM Service Functions used by an EMM

The man-page definitions for service functions used by an Elective Module Manager are presented in this section.

NAME

cssm_GetAttachFunctions

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_GetAttachFunctions
    (CSSM_MODULE_HANDLE hAddIn,
    CSSM_SERVICE_MASK AddinType,
    void **SPFunctions,
    CSSM_GUID_PTR Guid)
```

DESCRIPTION

This function returns an SPI function table for the service module identified by the module handle. The module must be of the type specified by the service mask. The *SPFunctions* parameter contains the returned function table. The elective module manager must use this function table to forward an application's call to the elective APIs to their corresponding SPIs represented in the function table. The returned *Guid* identifies the service module. It can be used to locate credentials and other information about the service module.

This function sets a lock on the SP functions table. The CSSM service function *cssm_ReleaseAttachFunctions*( ) must be used to release the lock.

PARAMETERS

hAddIn (input)
: The handle identifying the attach-session whose function table is to be returned by this function.

AddinType (input)
: A CSSM_SERVICE_MASK value identifying the type of service module whose function table is to be returned by this function.

SPFunctions (output)
: A pointer to the service module function table, which CSSM acquired from the service module during module-attach processing. The module manager should use this table to forward application invocation of the elective APIs to their corresponding SPIs. The memory pointed to by the function pointers should not be freed by the EMM.

Guid (output)
: A CSSM_GUID value identifying the service module whose function table is to be returned by this function.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See general and common error codes and error values section.

NAME

cssm_ReleaseAttachFunctions

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_ReleaseAttachFunctions
    (CSSM_MODULE_HANDLE hAddIn)
```

DESCRIPTION

This function releases the lock on the SP function table for the service module identified by the module handle. The SPI function table was obtained by the elective module manager through the *cssm_GetAttachFunctions*( ) operation.

PARMETERS

hAddIn (input)
: The handle identifying the attach-session whose function table is to be released by this function.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See also the general error codes and common error code and values section

NAME

cssm_GetAppMemoryFunctions

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_GetAppMemoryFunctions
    (CSSM_MODULE_HANDLE hAddIn,
    CSSM_UPCALLS_PTR UpcallTable)
```

DESCRIPTION

This function gets a function table containing sets of service functions. Among these service functions are four application-provided memory management functions. The elective module manager can use these functions to manage memory on behalf of the application. The returned function table is specific to the attach-session identified by the module handle.

PARAMETERS

hAddIn (input)
The handle identifying the attach-session whose memory management function table is returned by this function.

UpcallTable (output)
The table containing sets of service functions among them a set of four memory management functions provided by the application that initiated the attach-session identified by *hAddIn*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See also the general error codes and common error code and values section

NAME

cssm_IsFuncCallValid

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_IsFuncCallValid
    (CSSM_MODULE_HANDLE hAddin,
    CSSM_PROC_ADDR SrcAddress, /* application */
    CSSM_PROC_ADDR DestAddress,
    CSSM_PRIVILEGE InPriv,
    CSSM_PRIVILEGE *OutPriv,
    CSSM_BITMASK Hints,
    CSSM_BOOL * IsOK)
```

DESCRIPTION

This function checks secure linkage between an application and a service module. Based on address scope of the application and the service module associated with the attach handle, CSSM determines whether the *SrcAddress* is within an associated application and *DestAddress* is within the associated service module. The scope of the application and the service module is determined by their respective signed manifest credentials, which attest to the integrity of each entity.

This function uses the input privilege value *InPriv* to compare against the privilege range associated with the ranges for *SrcAddress* and *DestAddres*. The privilege check is performed when the *InPriv* privilege value is non-NULL. If the EMM wants the global privilege value to be checked, *InPriv* is zero and *OutPriv* is non-NULL. CSSM will return the privilege value in *OutPriv*. If integrity only checks are to be performed, *InPriv* is zero and *OutPriv* is NULL.

Another parameter called *Hints* is used to help CSSM efficiently perform the integrity and privilege verification operations. *Hints* helps CSSM know where to look to find the desired state information. In the regular case, CSSM will look for *SrcAddress* in the *CallerList* and *DestAddress* in the *AttachList*. For callback functions, the *SrcAddress* and *DestAddress* are likely to be in *AttachList*.

PARAMETERS

hAddIn (input)
: The handle identifying the attach-session whose caller and callee scope is being tested by this function.

SrcAddress (input/optional)
: An address to be tested for containment within the application that requested and created the attach-session identified by the module handle.

DestAddress (input/optional)
: An address within a service module. The destination address must be valid for the service provider associated with the attach-session identified by the module handle.

InPriv (input)
: The privilege value to be checked. Privilege checks apply to both *SrcAddress* and *DestAddress*.

OutPriv (output)
: If non-NULL, the global privilege will be checked and returned in *OutPriv*.

Hints (input)
: A flag providing search hints.

IsOK (output)
CSSM_TRUE if success, CSSM_FALSE if fail.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the general error codes and common error code and values section.

NAME

cssm_DeregisterManagerServices

SYNOPSIS

```
void CSSMAPI cssm_DeregisterManagerServices
    (const CSSM_GUID *Guid)
```

DESCRIPTION

This function is used by an elective module manager to de-register its function table with CSSM core services prior to termination. This function is invoked by an elective module manager only when exiting due to an error condition detected by the EMM. This allows CSSM to clean up any state information associated with the exiting EMM.

PARAMETERS

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

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the general error codes and common error code and values section.

Technical Standard

Part 14:

Add-In Module Structure and Administration

The Open Group

Chapter 37

Introduction

37.1 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 18-1 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 five basic categories of security services modules are:

- Cryptographic Service Providers (CSP)
- Trust Policy Modules (TPM)
- Authorization Computation Modules (ACM)
- 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 37-1 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. action. Service providers can require caller authentication before providing services. Application authentication is based on signed manifest credentials associated with the application.

Service modules can leverage other service modules in the implementation of their own services. Service modules acquire attach handles to other modules by:

- Receiving additional module handles from an invoking application
- Selecting and attach additional service module directly.

To prevent stealth attacks, CSSM performs secure linkage checks on function invocation. 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 algorithm configuration parameters for a service provider in the form of a cryptographic context. Applications may create, delete, and modify these contexts as necessary. CDSA components are checkable if the component has a manifest. Checkable components include add-in service modules, CSSM itself, and applications that use CSSM.

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

- Comprehensive, extensible SPIs for each of four categories of security services
- Dynamic management of all security service modules available to applications
- Dynamic management of elective module managers providing new security services
- 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
- Management support for concurrent security operations

37.2 Add-In Module Structure

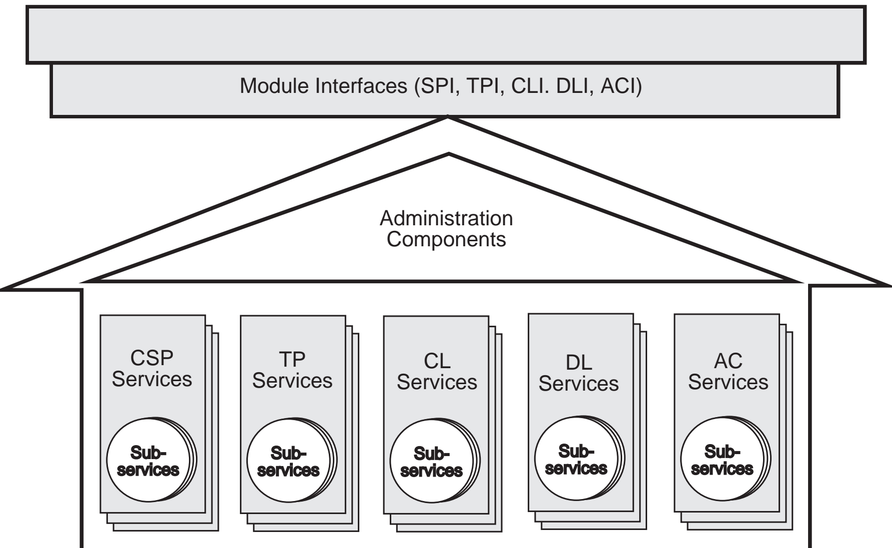


Figure 37-2 CDSA Add-In Module Structure

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 load, attach, and detach. The number, categories, and contents of the service implementations are determined by the module developer.

37.3 Module Installation

Before an application can use a module, the module must be registered by an installation program. The installation program uses platform-specific services to install all executable code associated with the service module. The installation program must also create records in Module Directory Services database. These records store information about the modules. At runtime this information is used by applications to locate and select service modules that provide the services required by the applications. The information is also used by any CDSA component that must check the integrity and authenticity of the service module.

The *MDS Common* relation stores the general attributes applicable to the four basic module types. Module installation must insert a new record in this relation, identifying and locating the new module.

MDS defines the EMM Service Provider relation to store general attributes of elective service providers. When installing a service module for an elective category of service, the installation program must insert a new record in this relation, identifying and locating the new module.

In addition to the *Common* relation, MDS defines numerous relations based on the basic categories of security services. These relations store detailed descriptions of the services provided by the module. New elective categories of service will define additional service-specific relations to store detailed descriptions of those modules that provide that type of service. Applications will search these service-specific relations to select appropriate service modules. The module installation program should consider inserting descriptive records into these additional relations.

37.4 Runtime LifeCycle of the Service Provider Module

Applications dynamically select the service modules that will provide security services to the application. Selection and session establishment is a multiple step process. When the module's services are no longer required, de-selection is also a multi-step process. The runtime life cycle of the service module and the sequence of function calls required among applications, CSSM and the service module are as follows:

1. An application invokes *CSSM_ModuleLoad*( ). CSSM will perform an integrity check on the module and then load the module using *SPI_ModuleLoad*( ).

2. The application invokes *CSSM_ModuleAttach*( ) to complete selection of the module.

3. CSSM carries out the attach process by calling the module interface *CSSM_SPI_ModuleAttach*( ). In response, the service module should perform a cross-check of the calling CSSM's signed manifest credential to guard against rogue callers.

   The *CSSM_SPI_ModuleAttach*( ) function is called each time an application invokes the *CSSM_ModuleAttach*( ) API.

4. When the application no longer requires the module's services, the application invokes *CSSM_ModuleDetach*( ).

5. In response, CSSM invokes *CSSM_SPI_ModuleDetach*( )

6. The application invokes *CSSM_ModuleUnload*( ) to deregister the application callback functions and to release all sessions associated with target service provider module.

7. In response, CSSM invokes *CSSM_SPI_ModuleUnload*( ). The function should disable events and de-register the CSSM *event-notification* function. The add-in service module may perform cleanup operations, reversing the initialization performed in *CSSM_SPI_ModuleLoad*( ).

Chapter 38
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

38.1 Security Services

The primary components of an add-in module are the security services that it offers. An add-in module may provide one or more 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.

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 respective Parts of this CDSA Technical Standard.

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.

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 available to applications in the relations managed by the Module Directory Services (MDS).

38.2 Module Administration Components

38.2.1 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.

CSSM requires successful certificate-based trust verification of all service modules when processing a *CSSM_ModuleLoad*() request. CSSM also requires trust verification of all Elective Module Managers when processing a *CSSM_ModuleAttach*() request.

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

CDSA defines a layered bilateral authentication procedure by which CSSM and an add-in module can authenticate each other to achieve a mutual trust. An add-in module is strongly encouraged to verify its own components. 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.

38.2.2 Module-Granted Use Exemptions

Service module vendors can choose to provide enhanced services to selected applications or classes of applications. A module-defined use policy is in addition to the general CSSM integrity policy. Categories of enhanced services are defined as use exemptions. CSSM_USEE_TAG declares the currently defined set of exemption classes. These focus primarily on exemptions for using cryptographic services. New exemption classes can be defined in association with any category of security services.

Service providers should record the exemptions they grant by listing them in an appropriate MDS relation. Currently, Cryptographic Service Providers (CSPs) advertise their exemptions by listing them in the *UseeTag* attribute of the MDS CSP Primary relation. This is for information only. The verifiable authorization to grant the exemption must be recorded in the service module's signed manifest credentials. If a module grants exemptions, then the module's signed manifest must include a manifest attribute attesting to this authority. The manifest attribute for currently defined exemptions is a name-value pair with name CDSA_USEE. The associated value is a string of base-64 encoded numbers separated by colons. An example of CDSA_USEE tag in the manifest (which corresponds to the base64 encoding of the CSSM_USEE_TAG) is:

```
CDSA_USEE: AAAAAg==:AAAABQ==:AAAAAw==
```

Applications can query MDS to retrieve the *UseeTags* associated with any CSP. The numbers which are base-64 encoded are the same numbers that are defined for CSSM_PRIVILEGE.

Exemptions which have numbers from the high order word of the CSSM_PRIVILEGE (see Chapter 6 on page 37) use the manifest tag CDSA_PRIV, as opposed to the tag CDSA_USEE. Other than that exception, they are used exactly like the tags listed as CDSA_USEE*.

There are two ways for an Application to request an exemption:

1. An application can request an exemption while making a call to the "P" functions, for example, *EncryptDataP, DecryptDataP, WrapKeyP* or *UnwrapKeyP*, by passing the USEE_TAG as the privilege parameter. (If *SetPrivilege* is called and then a call is made to one of the "P" functions, the USEE_TAG that is requested in the "P" function call will be sent to the service provider.)
2. An application can set the privilege for all of its calls to a service provider by calling *CSSM_Introduce*( ) and *CSSM_SetPrivilege*( ) functions. After calling *CSSM_SetPrivilege*( ), CSSM will then forward the USEE_TAG that was set to the service provider when a call is made to *EncryptData*, *DecryptData*, etc.

The appropriate USEE tag that validates the requested exemption may be in either the signed manifest of the Application (stored in the MDS Object Directory relation), and/or in the signed manifest of the CSSM. Either the Application or the CSSM (or both) must have a signed manifest with the appropriate USEE tag.

A service module may also check the application manifest directly by retrieving the application manifest from MDS and verifying the signature.

38.2.3 Service Module Requirements for USEE Tags Support

- Service modules must be signed by recognized authorities who can grant the use of the USEE tags.
- Service modules must verify that the direct calling module is a CSSM or CSSM or an EMM which has been introduced by a CSSM.
- Service module may support multiple USEE privileges.
- Service module must verify that the USEE tag supplied by the caller matches one of the supported USEE tags.
- Service module must list the supported USEE tags in a manifest attribute (e.g. CDSA_USEE).
- Use of cryptographic operations must be verified for each instance and must fall within the restrictions implied by the USEE tag.
- Every module of a CSP that contains cryptographic interfaces must be included in the CSP manifest.
- Additional components that influence the operation of the CSP should be included in the manifest.
- Multi-service modules that also implement CSP functionality must enforce the rules defined here for interfaces to cryptographic functions.
- A CSP must not allow any of its component modules' cryptographic interfaces to be directly called by an application module.

38.2.4 Initialization and Cleanup

Every module must include functions for module initialization and cleanup. Initialization should take place when CSSM calls either *CSSM_SPI_ModuleLoad*( ) or *CSSM_SPI_ModuleAttach*( ). Cleanup is performed in *CSSM_SPI_ModuleDetach*( ) and *CSSM_SPI_ModuleUnload*( ).

Chapter 39

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 loaded.

The module developer must create an installation program to register the module's identity, capabilities, and signed manifest credentials with the Module Directory Services (MDS).

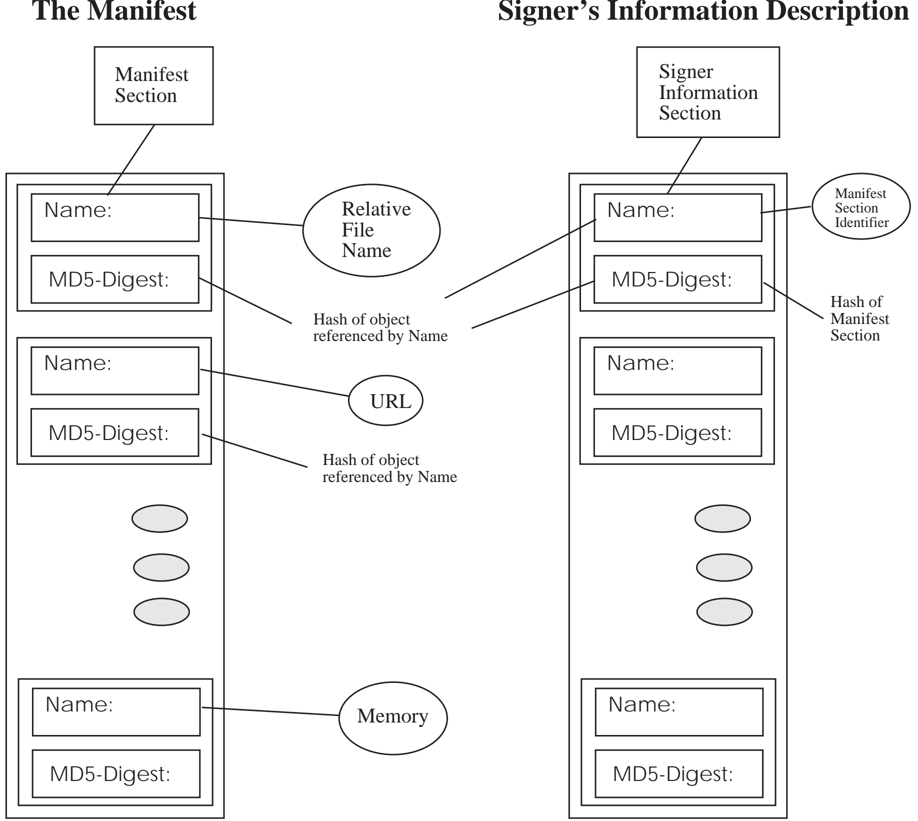
Finally, the module developer must ensure that the appropriate sequence of component verification and module initialization steps occur prior to dynamic binding of the module with CSSM.

39.1 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 is shown in Figure 45-1 and Figure 45-2. 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 that 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* that 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 form a single set of credentials. The credentials are stored as an attribute of the module. All module attributes are stored in relations managed by MDS. The module's manifest is stored as a binary blob representation of the manifest file in the MDS relation.



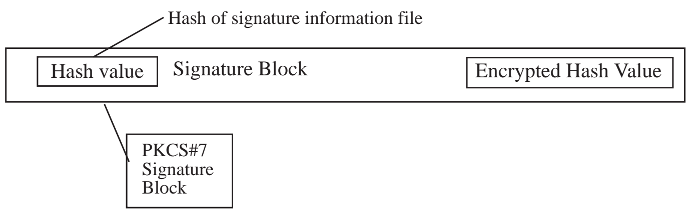


Figure 39-1 Credentials of an Add-In Service Module

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 45-2. 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.

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 39-2 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:

- (Optionally) A manifest attribute named CDSA_USEE and a set of use exemption strings
- The SHA-1 digital hashing algorithm identifier
- A SHA-1 hash of the object code file
- (Optionally) A reference to 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 and digests 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."

39.1.1 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 6-3. 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, and all chains sign the product. All three certificate chains are included in the credentials for this add-in module. When CSSM #1 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 either public key PK2 or public key PK3, which are recognized as points of trust by CSSM #1. Hence the add-in module's credentials will be successfully verified. CSSM #2 would verify the add-in module using public key PK5.

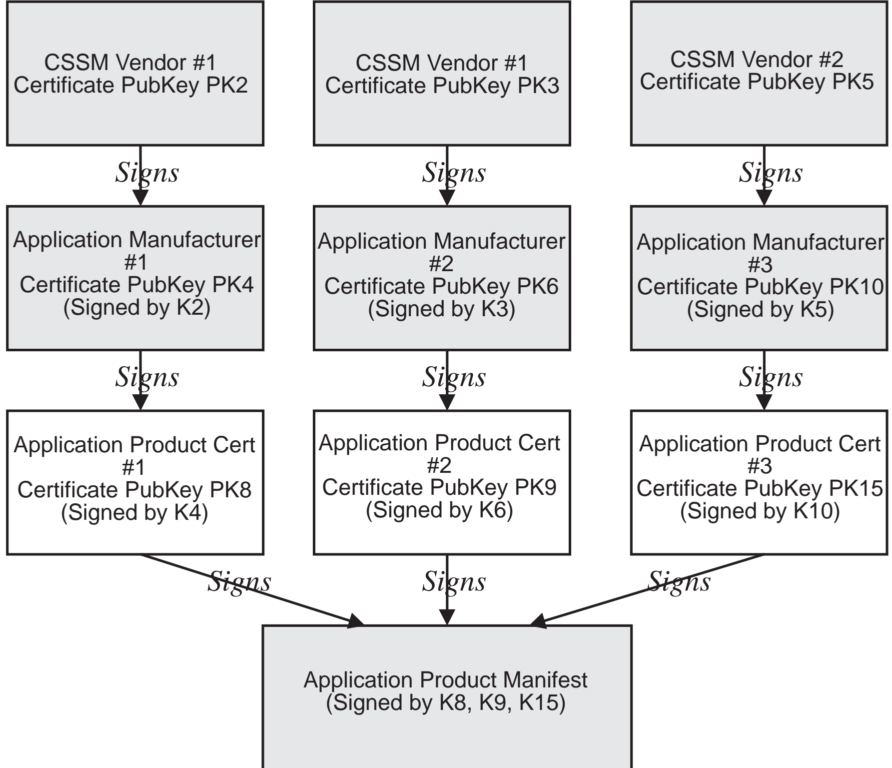


Figure 39-3 Signature File for Add-In Module with Authentication Capability

39.1.2 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.

39.1.3 Issuing an Add-In Module Product Certificate

A product certificate should be issued for each distinct product. The add-in module vendor defines what constitutes a distinct product. 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.

39.1.4 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, load-time integrity checking can be performed.

CSSM invokes the module's *CSSM_SPI_ModuleLoad*() 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 perform these checks at load-time and periodically during execution based on elapsed time or usage. Failure to perform this verification during module load 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, the service module credentials should be created. Signed manifest credentials consist of three sub-blocks:

- A manifest block
- A signer's information block
- A signature block

The manifest block contains:

- A hash of each executable software component of the service module

- (Optionally) A reference to each separately link-able software component comprising the service module.

After the manifest block is created, the signer information block is created. The signer's information file must contain:

- References to one or more sections of a manifest block
- A use exemption list for all the exemption classes the signer is granting to the module
- The hash of each reference section of a manifest block.

Finally the signature block is created. The signature block must contain:

- a signed hash of the signer's information block
- all of the certificate chains that are trusted by the service module.

The signing operation must be performed using the private key associated with the product certificate.

The signed manifest credentials must be included in a module-specific record in the *MDS Common* relation. The record(s) is created by the module's installation program.

39.2 Installing a Service Module

Before an application can use a module, the module's name, location and description must be registered by an installation program. The installation program creates one or more records in one or more MDS relations. To insert new records, the installation program must have write-access to the MDS CDSA Directory database. MDS controls write-access to all MDS databases. Write-access is granted only to signed, authorized applications.

The MDS CDSA Directory database contains numerous relations storing descriptions of CSSM, EMMs, and all types of CDSA service provider modules. Every service module must have at least one record in the *MDS Common* relation. This relation stores the general attributes applicable to all module types. These attributes include:

- The module's globally-unique identifier (GUID)
- A logical/descriptive module name
- The module's manifest
- File system reference to locate the module's executable code
- Version information.

The GUID is a structure used to differentiate between library modules. It is the primary database key for locating the module's records stored in MDS. The sub-components of a GUID are discussed in more detail below.

The logical name is a string chosen by the module developer to describe the module. The module location is determined at installation time. The installation program must set this value so the module can be located when CSSM attempts to load the service module is response to a application's *CSSM_ModuleLoad*( ) request. More detailed description of the module and its services are stored in other MDS relations. These options are described below.

39.2.1 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 as the primary database key to locate module information in MDS. When loading the library, the application uses the GUID to identify the requested module.

39.2.2 The Module Description

At install time, the installation program must register (advertise) its availability on the system. This is accomplished by adding one or more records to the MDS *Common* relation and optionally adding module-specific records to selected relations in the MDS CDSA Directory database. MDS defines relations based on security service types.

Service modules providing cryptographic services should consider inserting records in the following MDS relations:

- Primary CSP Relation
- CSP Capabilities Relation
- CSP Encapsulated Products Relation
- CSP Smartcard Relation.

Service modules providing data storage library services should consider inserting records in the following MDS relations:

- Primary DL Relation
- DL Encapsulated Products Relation

Service modules providing certificate library services should consider inserting records in the following MDS relations:

- Primary CL Relation
- CL Encapsulated Products Relation

Service modules providing trust policy services should consider inserting records in the following MDS relations:

- TP Primary Relation
- TP Policy OIDS Relation
- TP Encapsulated Products Relation

Multi-service modules should add records in all application categories of security services.

New elective security services areas must define new MDS relations to store appropriate description for those new categories of service. This requires schema modification and privileged access to the MDS database. See the section on installing elective module managers in this specification for the definition of the required procedures.

All records inserted in the MDS database are available to applications and other CDSA components via MDS queries. Consult the MDS description and schema definitions contained in this specification when developing an installation program for a service module.

39.3 Attaching a Service Module

39.3.1 Runtime Life Cycle of the Module

Applications dynamically select the service modules that will provide security services to the application. Selection and session establishment is a multiple step process. When the module's services are no longer required, de-selection is also a multi-step process. The runtime life cycle of the service module and the sequence of function calls required among applications, CSSM and the service module are as follows:

1. An application invokes *CSSM_ModuleLoad*( ) — after invoking *CSSM_Init*( ) and before invoking *CSSM_ModuleAttach*( ). The function *CSSM_ModuleLoad*( ) initializes the add-in service module. Initialization includes

   - registering the application's module-event handler

     and

   - enabling events with the add-in service module.

   The application may provide an event handler function to receive notification of insert, remove and fault events.

2. CSSM verifies the add-in service module's manifest credentials. If the credentials and the module code are successfully verified, CSSM invokes *CSSM_SPI_ModuleLoad*( ) to complete the module initialization process. *CSSM_SPI_ModuleLoad*( ) must be implemented by the service provider module. In response to this function, the service provider module must

   - Enable subservice notification events (Insert/Remove)
   - Register the CSSM callback function for processing events
   - Perform any module initialization steps (transparent to CSSM)
   - Generate the Insert event for static service modules

   The CSSM calls *CSSM_SPI_ModuleLoad*( ) each time the application invokes the *CSSM_ModuleLoad*( ) call.

3. The application invokes *CSSM_ModuleAttach*( ) to complete selection of the module.

4. CSSM carries out the attach process by calling the module interface *CSSM_SPI_ModuleAttach*( ).

   The service module should also:

   - Cross-check the caller (making authentication bilateral between CSSM and the service provider module)update internal state to manage the new session
   - Verify version compatibility
   - If this service module has not performed self-check during *SPI_ModuleLoad*( ) it should be performed the first time *SPI_ModuleAttach*( ) is called
   - Perform additional initialization operations
   - Return a table of protected function pointers to be proxied by CSSM in response to specific application calls. Note that the function table should contain only those functions that can be invoked by the application

   The *CSSM_SPI_ModuleAttach*( ) function is called each time an application invokes the *CSSM_ModuleAttach*( ) call.

The function table is returned to the CSSM for each attach handle. The service provider may vary the contents of the table based on the application's exemptions (if any).

5. When the application no longer requires the module's services, the application invokes *CSSM_ModuleDetach*( ). This closes the session and voids the attach handle associated with that session.
6. CSSM notifies the service provider modules of the application's detach request by invoking the *CSSM_SPI_ModuleDetach*( ) function. This SPI is invoked once for each application call to *CSSM_ModuleDetach*( ). In response to this function call, the service provider should clean-up any state information associated with the detaching session.
7. The application invokes *CSSM_ModuleUnload*( ) to deregister the application callback functions and to release all sessions associated with target service provider module.
8. In response, CSSM invokes *CSSM_SPI_ModuleUnload*( ). The service module must implement this function. The *CSSM_SPI_ModuleLoad*( ) function will be called each time the application invokes the *CSSM_ModuleUnload*( ) call. When an equal number of *CSSM_SPI_ModuleLoad*( ) and *CSSM_SPI_ModuleUnload*( ) calls have been made, the function should disable events and deregister the CSSM event-notification function. The add-in service module may perform cleanup operations, reversing the initialization performed in *CSSM_SPI_ModuleLoad*( ).

39.3.2 Bilateral Authentication

On *ModuleLoad*, CSSM and the 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.

The basic steps in bilateral authentication during module load 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

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.

In the event that a module's manifest describes more than one module, the module's GUID must be present in the manifest section. The GUID should be presented as a *tag:value* pair. The tag is "CDSA_GUID" and the value should be a string representation of the GUID. This allows the authenticating parties to correctly identify which module is of current concern. The verification of modules referred to by manifest sections other than the currently relevant manifest section is not necessary for bilateral authentication.

An example of a module GUID representation in a manifest is as follows:

```
CDSA_GUID: {01234567-9abc-def0-1234-56789abcdef0}
```

There should be a "CDSA_MODULE" tag in the manifest of a CSSM. The value of this tag should be "CSSM" so that other modules can verify that the calling module is a CSSM. EMMs should have the CDSA_MODULE tag with a value of "EMM" so entities (such as CSSM and addins) may verify that the module is an EMM. Other optional values for CDSA_MODULE tag are "APP" and "ADDIN" in the respective manifests.

39.3.3 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 as an input parameter to *CSSM_SPI_ModuleAttach*().

Memory management upcalls are pointers to the memory management functions used by CSSM. If needed, the call will be routed to the calling application. They are provided to a module via CSSM as a structure of function pointers. The functions will be the equivalents 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.

39.4 Modules Control Access to Objects

Service provider modules manage objects that are manipulated through the service provider's APIs. Each service provider can control access to these objects on a per request basis according to a policy enforced by the provider. Most of the access-controlled objects are persistent, but they can exist only for the duration of the current application execution. Examples include:

- Authorization to use a cryptographic key stored by a CSP
- Authorization to use a particular secret managed by a CSP
- Authorization to write records to a particular data store

A service provider must make an access control decision when faced with a request of the form "I am subject S. Do operation X for me." The decision requires the service provider to answer two questions:

- Is the requester really the subject S?
- Is S allowed to do X?

The first question is answered by authentication. The second question is answered by authorization.

39.4.1 Authentication as Part of Access Control

There are various forms of authentication. Traditionally, the term is applied to authentication of the human user. A human is often authenticated by something he or she knows, such as a passphrase, a PIN, etc. More secure authentication involves multiple factors:

- something the human knows
- something the human possesses
- something the human is, in the form of a biometric authentication.

It is also possible to authenticate an entity using public cryptography and digital certificates. The entity holding a keypair can be a hardware device or instance of some software. The device or the software acts on behalf of a human user. Each entity is identified by a public signature key. Authentication is performed by challenging the entity to create a digital signature using the private key. The signature can be verified and the entity is authenticated. The digital certificate and the digital signature are credentials presented by the entity for verification in the authentication process.

Each service provider defines a policy regarding the type and number of authentication credentials accepted for verification by the service module. The credentials can be valid for some fixed period of time or can be valid indefinitely, until rescinded by an appropriate revocation mechanism.

CDSA defines a general form of access credential a caller can present to service providers when operating on objects, whose access is controlled by the service provider. A credential set consists of:

- Zero or more digital certificates
- Zero or more samples

If the service provider caches authentication and authorization state information for a session, a caller may not be required to present any certificates or samples for subsequent accesses. Typically at least one sample is required to authenticate a caller and to verify the caller's authorization to perform a CDSA operation.

The general credential structure is used as an input parameter to functions in various categories of security services. A caller can provide samples through the access credentials structure in one of several modes or forms:

- Immediate values contained in the credentials structure - for example, a PIN, or a passphrase
- By reference to another authentication agent who will acquire and verify the credentials - for example, a biometric device and agent to acquire and verify biometric data from the caller, a protected PIN pad or some external authentication mechanism such as PAM.
- By providing a callback function that the service provider can invoke to obtain a sample on-demand - for example, invoking a function challenging the caller to sign a nonce
- Any combination of these forms

The service provider uses credentials to answer the authentication question.

39.4.2 Authorization as Part of Access Control

Once any necessary authentication samples have been gathered, authorization can proceed. Just providing a password or biometric sample does not imply that the user providing the sample should get the access he or she is requesting.

An authorization decision is based on an authorization policy. In CDSA, an authorization policy is expressed in a structure called an Access Control List (ACL). An ACL is a set of ACL entries, each entry specifying a subject that is allowed to have some particular access to some resource. Traditional ACLs (from the days of early time sharing systems) identify a subject by login name. The ACLs we deal with can identify a subject by login name, but more generally, the Subject is specified by the identification templates that is used to verify the samples presented through the credentials.

The ACL associated with a resource is the basis for all access control decision over that resource. Each entry within the ACL contains:

- Subject - a typed identification template (a type designator is part of the Subject, because multiple Subject types are possible)
- Delegation flag - indicating whether the subject can delegate the access rights (this only applies to public key templates)
- Authorization tag - defining the set of operations for which permission is granted to the Subject (the definition of authorization tags is left to the Service Provider developer, but in the interest of increased interoperability, we define tags for the basic operations represented by the defined standard API).
- Validity period - the time period for which the ACL entry is valid
- Entry tag - a user-defined string value associated with and identifying the ACL entry

The ACL entry does not explicitly identify the resource it protects. The service provider module must manage this association.

The basic authentication process verifies one or more samples against templates in the ACL. Each ACL entry with a verified subject yields an authorization available to that subject.

Beyond the basic process, it is possible to mark an ACL entry with the permission to delegate. The delegation happens by one or more authorization certificates. These certificates act to connect the authorization expressed in the ACL entry from the public key subject of that entry to the authorization template in the final (or only) certificate of the chain. That is, an authorization certificate acts as an extension cord from the ACL to the actual authorized subject. Delegation by certificate is an option when scaling issues mitigate against direct editing of the ACL for every change in authorized subject.

Service provider modules are responsible for managing their ACLs. When a new resource is created at least one ACL entry must be created. The implementation of ACLs is internal to the service provider module. The CDSA interface defines a CSSM_ACL_ENTRY_PROTOTYPE that is used by the caller and the service provider to exchange ACL information.

When a caller requests the creation of a new resource, the caller should present two items:

- A set of access credentials
- An initial ACL entry

The access credentials are required if the service provider module restricts the operation of resource creation. In many situations, a service provider allows anyone to create new resources. For example, some CSPs allow anyone to create a key pair. If resource creation is controlled, then

the caller must present a set of credentials for authorization. Authentication will be performed based upon the set of samples introduced through the credentials. Upon successful authentication, the resulting authorization computation determines if the caller is authorized to create new resources within a controlled resource pool or container. If so, the new resource is created.

When a resource is created, the caller also provides an initial ACL entry. This entry is used to control future access to the new resource and its associated ACL (see Section 4.4 on page 31). The service provider can modify the caller-provided initial ACL entry to conform to any innate resource access policy the service provider may define. For example, a smartcard may not allow key extraction. When creating a key pair on the smartcard, a caller can not give permission to perform the CDSA operation CSSM_WrapKey. The attempt will result in an error.

39.4.3 Resource Owner

How a given resource controller actually records the ownership of the resource is up to the developer of that code, but the "Owner" of a resource can be thought of as being recorded in a one-entry ACL of its own. Therefore, conceptually there are two ACLs for each resource: one that can grow and shrink and give access to users to the resource, and another that always has only one entry and specifies the owner of the resource (and the resource ACL). On resource creation, the caller supplies one ACL entry. That one entry is used to initialize both the Owner entry and the resource ACL. This is to accommodate the common case in which a resource will be owned and used by the same person. In other cases, either the Owner or the ACL can be modified after creation.

Only the "Owner" is authorized to change the ACL on the resource, and only the "Owner" is authorized to change the "Owner" of the resource. Effectively, the "Owner" acts as "the ACL on the ACL" for the full lifetime of the resource. In terms of an ACL entry, it only contains the "subject" (i.e., identifies the "Owner"), and the "delegate" flag (initially set to "No delegation"). The "Authorization tag" is assumed to convey full authority to edit the ACL, and the "Validity period" is assumed to be the lifetime of the resource. There is no "Entry tag" associated with the "Owner". Note that an "Owner" may be a threshold subject; identifying many "users" who are authorized to change the ACL. Note also that "Ownership" does not convey the right to delete the resource; that right may or may not be conveyed by the ACL.

CDSA defines functions to modify an ACL during the life of the associated resource. ACL updates include:

- Adding new entries
- Replacing/updating existing entries
- Deleting an existing entry
- Changing the "Owner"

Modifying an ACL is a controlled operation. Credentials must be presented and authenticated to prove that the caller is the "Owner".

39.5 Error Handling

When an error occurs inside a module, the function should return the error number used to describe the error.

The error numbers returned by a module 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 function. They are described in this specification in the section on data structures for core services. 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_BASE_ERROR + CSSM_ERRCODE_MODULE_EXTENT, where XX stands for the service category abbreviation (CSP, TP, AC, CL, DL). A module developer is responsible for making the definition and interpretation of their module-specific error codes available to applications.

39.6 Data Structure for Add-in Modules

39.6.1 CSSM_SPI_ModuleEventHandler

This defines the event handler interface CSSM defines and implements to receive asynchronous notification of events of type CSSM_MODULE_EVENT from a service provider module. Example events include insertion or removal of a hardware service module, or fault detection.

This event structure is passed to the service module during *CSSM_SPI_ModuleLoad*. This is the single event handle the service module should use to notify CSSM of these event types for all of the attached session with the loaded module. CSSM forwards the event to the entity that invoked the corresponding *CSSM_ModuleLoad*() function. The handler specified in *CSSM_SPI_ModuleEventHandler* can be invoked multiple times in response to a single event (such as the insertion of a smartcard).

```
typedef CSSM_RETURN (CSSMAPI *CSSM_SPI_ModuleEventHandler)
    (const CSSM_GUID *ModuleGuid,
    void* CssmNotifyCallbackCtx,
    uint32 SubserviceId,
    CSSM_SERVICE_TYPE ServiceType,
    CSSM_MODULE_EVENT EventType)
```

Definition

ModuleGuid
The GUID of the service module raising the event.

CssmNotifyCallbackCtx
A CSSM context specified during *CSSM_SPI_ModuleLoad*().

SubserviceId
The subserviceId of the service module raising the event.

ServiceType
The service mask of the sub-service identified by *Subservice*.

EventType
The CSSM_MODULE_EVENT that has occurred.

39.6.2 CSSM_CONTEXT_EVENT_TYPE

This list defines event in the lifecycle of cryptographic contexts. When such an event occurs, CSSM delivers the appropriate event signal to the affected cryptographic service provider. Signals of this type are issued to the module by invoking the *EventNotify* interface, which the cryptographic service provider module specifies during module attach processing.

```
typedef enum cssm_context_event{
    CSSM_CONTEXT_EVENT_CREATE = 1,
    CSSM_CONTEXT_EVENT_DELETE = 2,
    CSSM_CONTEXT_EVENT_UPDATE = 3,
} CSSM_CONTEXT_EVENT;
```

39.6.3 CSSM_MODULE_FUNCS

This structure is used by add-in service modules to return function pointers for all service provider interfaces that can be invoked by CSSM. This includes interfaces to real security services and interfaces to administrative functions used by CSSM and the service provider to maintain the environment. This structure accommodates function tables for service modules implementing new elective categories of services that have not been defined yet. Many operating environments provide platform-specific support for strong type checking applied to function pointers. This specification allows for the use of such mechanisms when available. In the general case a function pointer is considered to be a value of type CSSM_PROC_ADDR.

```
typedef struct cssm_module_funcs {
    CSSM_SERVICE_TYPE ServiceType;
    uint32 NumberOfServiceFuncs;
    const CSSM_PROC_ADDR *ServiceFuncs;
} CSSM_MODULE_FUNCS, *CSSM_MODULE_FUNCS_PTR;
```

Definition

ServiceType
A CSSM_SERVICE_TYPE value indicating the category of security services available through the function table. For known categories of service, this type is used to determine the ordering of function pointers within the function table.

NumberOfServiceFuncs
The number of function pointers for the security service functions contained in the table of *ServiceFuncs*.

ServiceFuncs
Memory address of the beginning of the function pointer table for the security service functions identified by *ServiceType*.

39.6.4 CSSM_UPCALLS

This structure is used by CSSM to provide service functions to add-in service modules and elective module managers. The provided functions include:

- Memory management functions provided by an application
- CSSM provided services to query the state of an attach-session between an application and a particular service module

A service module or an elective module manager can invoke this service at anytime during the life cycle of an attach-session.

```
typedef void * (CSSMAPI *CSSM_UPCALLS_MALLOC)
    (CSSM_HANDLE AddInHandle,
    uint32 size);
typedef void (CSSMAPI *CSSM_UPCALLS_FREE)
    (CSSM_HANDLE AddInHandle,
    void *memblock);
typedef void * (CSSMAPI *CSSM_UPCALLS_REALLOC)
    (CSSM_HANDLE AddInHandle,
    void *memblock,
    uint32 size);
typedef void * (CSSMAPI *CSSM_UPCALLS_CALLOC)
    (CSSM_HANDLE AddInHandle,
    uint32 num,
    uint32 size);
typedef struct cssm_upcalls {
    CSSM_UPCALLS_MALLOC malloc_func;
    CSSM_UPCALLS_FREE free_func;
    CSSM_UPCALLS_REALLOC realloc_func;
    CSSM_UPCALLS_CALLOC calloc_func;
    CSSM_RETURN CSSMAPI (*CcToHandle_func)
        (CSSM_CC_HANDLE Cc,
        CSSM_MODULE_HANDLE_PTR ModuleHandle);
    CSSM_RETURN (CSSMAPI *GetModuleInfo_func)
        (CSSM_MODULE_HANDLE Module,
        CSSM_GUID_PTR Guid,
        CSSM_VERSION_PTR Version,
        uint32 *SubServiceId,
        CSSM_SERVICE_TYPE *SubServiceType,
        CSSM_ATTACH_FLAGS *AttachFlags,
        CSSM_KEY_HIERARCHY *KeyHierarchy,
        CSSM_API_MEMORY_FUNCS_PTR AttachedMemFuncs,
        CSSM_FUNC_NAME_ADDR_PTR FunctionTable,
        uint32 *NumFunctionTable);
} CSSM_UPCALLS, *CSSM_UPCALLS_PTR;
```

Definition

malloc_func
The application-provided function for allocating memory in the application's memory space.

free_func
The application-provided function for freeing memory allocated in the application's memory space using *malloc_func*.

realloc_func
The application-provided function for re-allocating memory in the application's memory space that was previously allocated using *malloc_func*.

calloc_func
The application-provided function for allocating a specified number of memory units in the application's memory space.

CcToHandle
A CSSM service function returning the module attach handle associated with a cryptographic context handle.

GetModuleInfo_func
A CSSM service function for use by Elective Module Managers and service modules to obtain the state information associated with the module handle. This information is initialized when an application calls *CSSM_ModuleAttach*( ). Returned information includes the application-specified memory management functions, callback function, and callback context associated with the module handle.

Chapter 40

Add-In Module Interface Functions

A service module must implement the following administrative interfaces to proper interact with CSSM. This service provider must explicitly export these function interfaces for invocation by CSSM using services provided by their common base operating environment.

NAME

CSSM_SPI_ModuleLoad

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSSM_SPI_ModuleLoad
    (const CSSM_GUID *CssmGuid,
    const CSSM_GUID *ModuleGuid,
    CSSM_SPI_ModuleEventHandler CssmNotifyCallback,
    void* CssmNotifyCallbackCtx)
```

DESCRIPTION

This function completes the module initialization process between CSSM and the add-in service module. Before invoking this function, CSSM verifies the add-in service module's manifest credentials. If the credentials verify this module is loaded (physically if required) and *CSSM_SPI_ModuleLoad*( ) is invoked.

The *CssmGuid* identifies the caller and should be used by the module to locate the caller's signed manifest credentials and to complete integrity verification and secure linkage checks on the caller. The *ModuleGuid* identifies the invoked module and should be used by the module to locate its credentials and to complete an integrity self-check.

The *CssmNotifyCallback* and *CssmNotifyCallbackCtx* defines a callback and callback context respectively. The module must retain this information for later use. The module should use the callback to notify CSSM of module events of type CSSM_MODULE_EVENT in any ongoing, attached sessions.

PARAMETERS

CssmGuid (input)
The CSSM_GUID of the caller. Used to locate the caller's signed manifest credentials.

ModuleGuid (input)
The CSSM_GUID of the invoked service provider module. Used to locate the module's signed manifest credentials.

CssmNotifyCallback (input)
A function pointer for the CSSM event handler that manages events of type CSSM_MODULE_EVENT.

CssmNotifyCallbackCtx (input)
The context to be returned to CSSM as input on each callback to the event handler defined by *CssmNotifyCallback*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

Refer to the error codes defined in **Part 2**.

SEE ALSO

CSSM_SPI_ModuleUnload( )
CSSM_SPI_ModuleAttach( )

NAME

CSSM_SPI_ModuleUnload

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSSM_SPI_ModuleUnload
    (const CSSM_GUID *CssmGuid,
    const CSSM_GUID *ModuleGuid,
    CSSM_SPI_ModuleEventHandler CssmNotifyCallback,
    void* CssmNotifyCallbackCtx)
```

DESCRIPTION

This function disables events and de-registers the CSSM *event-notification* function. The add-in service module may perform cleanup operations, reversing the initialization performed in *CSSM_SPI_ModuleLoad*( ).

PARAMETERS

CssmGuid (input)
The CSSM_GUID of the caller.

ModuleGuid (input)
The CSSM_GUID of the invoked service provider module.

CssmNotifyCallback (input)
A function pointer for the CSSM event handler that manages events of type CSSM_MODULE_EVENT.

CssmNotifyCallbackCtx (input)
The context to be returned to CSSM as input on each callback to the event handler defined by *CssmNotifyCallback*.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

Refer to the error codes defined in **Part 2**.

SEE ALSO

CSSM_SPI_ModuleLoad( )
CSSM_SPI_ModuleDetach( )

NAME

CSSM_SPI_ModuleAttach

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSSM_SPI_ModuleAttach
    (const CSSM_GUID *ModuleGuid,
    const CSSM_VERSION *Version,
    uint32 SubserviceID,
    CSSM_SERVICE_TYPE SubServiceType,
    CSSM_ATTACH_FLAGS AttachFlags,
    CSSM_MODULE_HANDLE ModuleHandle,
    CSSM_KEY_HIERARCHY KeyHierarchy,
    const CSSM_GUID *CssmGuid,
    const CSSM_GUID *ModuleManagerGuid,
    const CSSM_GUID *CallerGuid,
    const CSSM_UPCALLS *Upcalls,
    CSSM_MODULE_FUNCS_PTR *FuncTbl)
```

DESCRIPTION

This function is invoked by CSSM once for each invocation of *CSSM_ModuleAttach*() specifying the module identified by *ModuleGuid*. Four entities are stakeholders in this function and each is identified by a CSSM_GUID value:

- Service Module - the executing service provider performing the *CSSM_SPI_ModuleAttach*() operation. The module is identified by *ModuleGuid*.
- CSSM - the CSSM that invoked the Service Module. CSSM is identified by *CssmGuid*.
- *ModuleManagerGuid*- the module that will be routing calls to the service provider. This value will be the same as *CssmGuid* if CSSM is managing the calls to this service provider.
- Caller - the entity who invoked CSSM through the function *CSSM_ModuleAttach*(). The caller is identified by *CallerGuid*.

The service provider module should perform an integrity check of CSSM. *CssmGuid* can be used to locate CSSM's signed manifest credentials. The service provider can require an integrity check of the Caller. *CallerGuid* can be used to locate the Caller's signed manifest credentials. The *KeyHierarchy* flag identifies the class of embedded public keys CSSM will use to check the integrity of the service provider. If the manifest for the target module does not encounter an embedded key for all the key classes in *KeyHierarchy*, integrity cross-check fails.

The service module must verify compatibility with the system version level specified by *Version*. If the version is not compatible, then this function fails. The service module should perform all initializations required to support the new attached session and should return a function table for the SPI entry points that can be invoked by CSSM in response to API invocations by *Caller*. CSSM uses this function table to dispatch requests on for the attach session created by this function. Each attach session has its own function table.

PARAMETERS

ModuleGuid (input)
: The CSSM_GUID of the invoked service provider module.

Version (input)
: The major and minor version number of the required level of system services and features. The service module must determine whether its services are compatible with the required version.

SubserviceId (input)
The identifier for the requested subservice within this module. If only one service is provided by the module, then *subserviceId* can be zero.

SubServiceType (noutput)
A CSSM_SERVICE_MASK indicating the type of services provided by the service module and the ordering of the function table returned in the output parameter *FuncTbl*.

AttachFlags (input)
A mask representing the caller's request for session-specific services.

ModuleHandle (input)
The CSSM_HANDLE value assigned by CSSM and associated with the attach session being created by this function.

KeyHierarchy (input)
The CSSM_KEY_HIERARCHY flag directing CSSM which embedded key(s) to use when verifying integrity of the named modules.

CssmGuid (input)
The CSSM_GUID of the CSSM invoking this function.

ModuleManagerGuid (input)
The CSSM_GUID of the module that will route calls to the service provider.

CallerGuid (input)
The CSSM_GUID of the Caller who invoked *CSSM_ModuleAttach*( ), which resulted in CSSM invoking this function.

Upcalls (input)
A set of function pointers the service module must use to obtain selected CSSM services and to manage application memory. The memory management functions are provided by the application invoking *CSSM_ModuleAttach*( ). CSSM forwards these function pointers with CSSM service function pointers to the module.

FuncTbl (output)
A CSSM_MODULE_FUNCS table containing pointers to the service module functions the Caller can use. CSSM uses this table to proxy calls from an application caller to the add-in service module.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

Refer to the error codes in **Part 2**.

SEE ALSO

CSSM_SPI_ModuleDetach( )
CSSM_SPI_ModuleLoad( )

NAME

CSSM_SPI_ModuleDetach

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSSM_SPI_ModuleDetach
    (CSSM_MODULE_HANDLE ModuleHandle)
```

DESCRIPTION

This function is invoked by CSSM once for each invocation of *CSSM_ModuleDetach*() specifying the attach-session identified by *ModuleHandle*. The function entry point for *CSSM_SPI_ModuleDetach* is included in the module function table CSSM_MODULE_FUNCS returned to CSSM as output of a successful *CSSM_SPI_ModuleAttach*.

The service module must perform all cleanup operations associated with the specified attach handle.

PARAMETERS

ModuleHandle (input)
The CSSM_HANDLE value associated with the attach session being terminated by this function.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

Refer to the error codes in **Part 2**.

SEE ALSO

CSSM_SPI_ModuleAttach()
CSSM_SPI_ModuleUnload()

Chapter 41

CSSM Upcalls for Service Provider Modules

The CSSM upcalls presented in this Chapter are part of a small set of management functions used by CSSM and service provider modules to exchange internal state information. The upcall functions are not external interfaces. The CSSM provides these function entry points by initializing a CSSM_UPCALL structure and passing that structure to the service provider module during *ModuleLoad* and *ModuleAttach* processing.

Four upcall functions not included in this section are *malloc*, *free*, *calloc*, and *realloc*. CSSM provides these memory management functions so that service provider modules can use the application's memory to exchange data.

NAME

cssm_CcToHandle

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_CcToHandle
    (CSSM_CC_HANDLE Cc,
    CSSM_MODULE_HANDLE_PTR ModuleHandle)
```

DESCRIPTION

This function returns the module attach handle identifying the service module that is managing the specified cryptographic context.

The entry point to this function is provided to a service module in a table of *upcall* functions passed to the service provider during module attach processing.

If the PVC checking for service providers is on, the service provider has to introduce itself before calling this function.

PARAMETERS

Cc (input)
A handle identifying a cryptographic context.

ModuleHandle (output)
A service provider's module attach handle. This value will be set to CSSM_INVALID_HANDLE if the function fails.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See the General Error Codes and Common Error Codes and Values.

NAME

cssm_GetModuleInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI cssm_GetModuleInfo
    (CSSM_MODULE_HANDLE Module,
    CSSM_GUID_PTR Guid,
    CSSM_VERSION_PTR Version,
    uint32 *SubServiceId,
    CSSM_SERVICE_TYPE *SubServiceType,
    CSSM_ATTACH_FLAGS *AttachFlags,
    CSSM_KEY_HIERARCHY *KeyHierarchy,
    CSSM_API_MEMORY_FUNCS_PTR AttachedMemFuncs,
    CSSM_FUNC_NAME_ADDR_PTR FunctionTable,
    uint32 NumFunctionTable)
```

DESCRIPTION

This function returns the state information associated with the module handle. The information returned by this function is that set by the call to *CSSM_ModuleAttach*(). The entry point to this function is provided to a service module in a table of upcall functions passed to the service provider during module attach processing.

If the PVC checking for service providers is on, the service provider has to introduce itself before calling this function.

PARAMETERS

Module (input)
: The handle to a service provider module.

Guid (output)
: A CSSM_GUID associated with the module handle.

Version (output)
: The version number set on *ModuleAttach*.

SubServiceId (output)
: The slot number of the reader to which the module is attached.

SubServiceType (output)
: A CSSM_SERVICE_TYPE value identifying the class of security service

AttachFlags (output)
: This parameter provides the caller with session specific information associated with the module handle.

KeyHierarchy (output)
: The key hierarchy supplied when the module was attached.

AttachedMemFuncs (output)
: The memory functions supplied when the module was attached.

FunctionTable (input/output/optional)
: A table of function-name and API function-pointer pairs. The caller provides the name of the functions as input. The corresponding API function pointers are returned on output. The function table allows dynamic linking of CDSA interfaces, including interfaces to Elective Module Managers, which are transparently loaded by CSSM during *CSSM_ModuleAttach*(). The caller of this function should allocate the memory for the number of slots required.

NumFunctionTable (input)
The number of entries in the *FunctionTable* parameter. If no *FunctionTable* is provided, this value must be zero.

RETURN VALUE

A CSSM_RETURN value indicating success or specifying a particular error condition. The value CSSM_OK indicates success. All other values represent an error condition.

ERRORS

See General Error Codes and Common Error Codes and Values.

Technical Standard

Part 15:

Appendices, Glossary and Index

The Open Group

Appendix A

CSSM Error Handling

A.1 Introduction

This Appendix 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 return a value of type CSSM_RETURN. The value CSSM_OK indicates that the function was successful. Any other value is an error value from a service provider or CSSM. There are two types of error values that can be returned from a CSSM API function when the return value is not CSSM_OK:

- Error values which CSSM has defined specifically (e.g. CSSMERR_CSSM_INTERNAL_ERROR)
- Error values which are particular to a module — a custom error value.

CSSM reserves a set of pre-defined numeric offset values for its use; these offset values must be used as defined in the CSSM specification. A separate set of predefined offset values is reserved for module developers to indicate custom error values.

The calling application must determine how to handle an error returned by an API. Detailed descriptions of the error values are available in the corresponding Chapters of this specification,

Error values should not be overwritten, if at all possible. Overwriting the error value may destroy valuable error handling and debugging information. This means an add-in module of type A can return an error value defined by an add-in module of type B.

A.2 Error Values and Error Codes Scheme

Error Value refers to the 32-bit CSSM_RETURN value.

Error Code
refers to the *code* portion of the Error Value.

The code is an offset from the error base for a specific CSSM module type.

This Appendix includes:

- A listing of Configurable CSSM Error Code Constants
- A listing of CSSM Error Code Constants
- A listing of General Error Values that may be returned by the CSSM core module from any CSSM API
- Listings of common error codes. Common error codes are codes that can be associated with multiple module types.
- A listing of Configurable CSSM Error Code Constants for specific module types

Module-specific error value definitions reside in the associated module definition chapter of the specification.

A.3 Error Codes and Error Value Enumeration

A.3.1 Configurable CSSM Error Code Constants

These constants define constants that are configurable.

- `#define CSSM_BASE_ERROR CSSM_ERRORCODE_MODULE_EXTENT`

  The implementation-defined CSSM error code base value.

- `#define CSSM_ERRORCODE_MODULE_EXTENT (0x00000800)`

  The implementation-defined number of error codes allocated for each module type. This number must be greater than CSSM_ERRORCODE_COMMON_EXTENT, and should allow at least half the space for spec-defined error codes. The minimum value for this constant is 0x00000800.

- `#define CSSM_ERRORCODE_CUSTOM_OFFSET (0x00000400)`

  The platform-specific offset at which custom error codes are allocated:
  must be greater than CSSM_ERRCODE_COMMON_EXTENT
  and less than CSSM_ERRORCODE_MODULE_EXTENT.
  The minimum value for this constant is 0x00000400.

- `#define CSSM_ERRORCODE_COMMON_EXTENT (0x100)`

  The number of error codes allocated to indicate "common" errors.

A.3.2 CSSM Error Code Constants

```
#define CSSM_CSSM_BASE_ERROR (CSSM_BASE_ERROR)
#define CSSM_CSSM_PRIVATE_ERROR (CSSM_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
#define CSSM_CSP_BASE_ERROR (CSSM_CSSM_BASE_ERROR+CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_CSP_PRIVATE_ERROR (CSSM_CSP_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
#define CSSM_DL_BASE_ERROR (CSSM_CSP_BASE_ERROR + CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_DL_PRIVATE_ERROR (CSSM_DL_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
#define CSSM_CL_BASE_ERROR (CSSM_DL_BASE_ERROR + CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_CL_PRIVATE_ERROR (CSSM_CL_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
#define CSSM_TP_BASE_ERROR (CSSM_CL_BASE_ERROR + CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_TP_PRIVATE_ERROR (CSSM_TP_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET )
#define CSSM_KR_BASE_ERROR (CSSM_TP_BASE_ERROR + CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_KR_PRIVATE_ERROR (CSSM_KR_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
#define CSSM_AC_BASE_ERROR (CSSM_KR_BASE_ERROR + CSSM_ERRORCODE_MODULE_EXTENT)
#define CSSM_AC_PRIVATE_ERROR (CSSM_AC_BASE_ERROR + CSSM_ERRCODE_CUSTOM_OFFSET)
```

A.3.3 General Error Values

The following error values can be returned from any CSSM API.

```
#define CSSMERR_CSSM_INVALID_ADDIN_HANDLE \
    (CSSM_CSSM_BASE_ERROR + CSSM_ERRORCODE_COMMON_EXTENT + 1)
```

The given service provider handle is not valid

```
#define CSSMERR_CSSM_NOT_INITIALIZED \
    (CSSM_CSSM_BASE_ERROR + CSSM_ERRORCODE_COMMON_EXTENT + 2)
```

A function is called without initializing CSSM

```
#define CSSMERR_CSSM_INVALID_HANDLE_USAGE \
    (CSSM_CSSM_BASE_ERROR + CSSM_ERRORCODE_COMMON_EXTENT + 3)
```

Handle does not match with service type

```
#define CSSMERR_CSSM_PVC_REFERENT_NOT_FOUND \
    (CSSM_CSSM_BASE_ERROR + CSSM_ERRORCODE_COMMON_EXTENT + 4)
```

A reference to the calling module was not found in the list of authorized callers.

```
#define CSSMERR_CSSM_FUNCTION_INTEGRITY_FAIL \
    (CSSM_CSSM_BASE_ERROR + CSSM_ERRORCODE_COMMON_EXTENT + 5)
```

Function address is not within the verified module

A.3.4 Common Error Codes For All Module Types

The following codes can be returned by multiple module types.

```
#define CSSM_ERRCODE_INTERNAL_ERROR (0x0001)
```

General system error; indicates that an operating system or internal state error has occurred and the system may not be in a known state

```
#define CSSM_ERRCODE_MEMORY_ERROR ( 0x0002 )
```

A memory error occurred

```
#define CSSM_ERRCODE_MDS_ERROR  ( 0x0003 )
```

The MDS could not be accessed to complete the operation

```
#define CSSM_ERRCODE_INVALID_POINTER (0x0004)
```

An input/output function parameter or input/output field inside of a data structure is an invalid pointer

```
#define CSSM_ERRCODE_INVALID_INPUT_POINTER (0x0005)
```

An input function parameter or input field in a data structure is an invalid pointer

```
#define CSSM_ERRCODE_INVALID_OUTPUT_POINTER (0x0006)
```

An output function parameter or output field in a data structure is an invalid pointer

```
#define CSSM_ERRCODE_FUNCTION_NOT_IMPLEMENTED (0x0007)
```

The function is not implemented by the service provider

```
#define CSSM_ERRCODE_SELF_CHECK_FAILED (0x0008)
```

The module failed a self-check

```
#define CSSM_ERRCODE_OS_ACCESS_DENIED (0x0009)
```

The operating system denied access to a required resource

```
#define CSSM_ERRCODE_FUNCTION_FAILED (0x000A)
```

The function failed for an unknown reason.

```
#define CSSM_ERRCODE_MODULE_MANIFEST_VERIFY_FAILED (0x000B)
```

Module's manifest verification failed

```
#define CSSM_ERRCODE_INVALID_GUID (0x000C)
```

Invalid GUID

A.3.5 Common Error Codes for ACLs

The following code enumeration values apply to any module that supports ACLs.

```
#define CSSM_ERRCODE_OPERATION_AUTH_DENIED (0x0020)
```

The access credentials were insufficient to permit the requested action to complete

```
#define CSSM_ERRCODE_OBJECT_USE_AUTH_DENIED (0x0021)
```

The access credentials were insufficient to permit the associated object to be used

```
#define CSSM_ERRCODE_OBJECT_MANIP_AUTH_DENIED (0x0022)
```

The access credentials were insufficient to permit manipulation of the object that is the target of an operation (i.e. *CSSM_WrapKey*( ), *CSSM_UnwrapKey*( )).

```
#define CSSM_ERRCODE_OBJECT_ACL_NOT_SUPPORTED (0x0023)
```

An ACL is specified for a new object and the service provider does not support ACLs for objects of that type

```
#define CSSM_ERRCODE_OBJECT_ACL_REQUIRED (0x0024)
```

An ACL is not specified for a new object and the service provider requires an ACLs for objects of that type

```
#define CSSM_ERRCODE_INVALID_ACCESS_CREDENTIALS (0x0025)
```

The access credentials are invalid

```
#define CSSM_ERRCODE_INVALID_ACL_BASE_CERTS (0x0026)
```

The base certificates were corrupt or not recognized as the indicated type

```
#define CSSM_ERRCODE_ACL_BASE_CERTS_NOT_SUPPORTED (0x0027)
```

The type of at least one certificate in the base certificates is not supported

```
#define CSSM_ERRCODE_INVALID_SAMPLE_VALUE (0x0028)
```

The sample value is corrupt or not recognized as the indicated type

```
#define CSSM_ERRCODE_SAMPLE_VALUE_NOT_SUPPORTED (0x0029)
```

The type of at least one sample is not supported

```
#define CSSM_ERRCODE_INVALID_ACL_SUBJECT_VALUE (0x002A)
```

The subject value is corrupt or not recognized as the indicated type

```
#define CSSM_ERRCODE_ACL_SUBJECT_TYPE_NOT_SUPPORTED (0x002B)
```

The type of the subject value is not supported

```
#define CSSM_ERRCODE_INVALID_ACL_CHALLENGE_CALLBACK (0x002C)
```

The challenge or subject callback function pointer is invalid

```
#define CSSM_ERRCODE_ACL_CHALLENGE_CALLBACK_FAILED (0x002D)
```

The challenge or subject callback to the client failed

```
#define CSSM_ERRCODE_INVALID_ACL_ENTRY_TAG (0x002E)
```

The entry tag value is not valid.

```
#define CSSM_ERRCODE_ACL_ENTRY_TAG_NOT_FOUND (0x002F)
```

No ACL entry was found with the specified entry tag

```
#define CSSM_ERRCODE_INVALID_ACL_EDIT_MODE (0x0030)
```

The edit mode specified when changing an ACL entry is not valid

```
#define CSSM_ERRCODE_ACL_CHANGE_FAILED (0x0031)
```

The ACL update operation failed

```
#define CSSM_ERRCODE_INVALID_NEW_ACL_ENTRY (0x0032)
```

The ACL entry specified for an initial or modified value is invalid

```
#define CSSM_ERRCODE_INVALID_NEW_ACL_OWNER (0x0033)
```

The ACL owner specified for a modified value is invalid

```
#define CSSM_ERRCODE_ACL_DELETE_FAILED (0x0034)
```

ACL entry to be deleted was not found

```
#define CSSM_ERRCODE_ACL_REPLACE_FAILED (0x0035)
```

ACL entry to be replaced was not found

```
#define CSSM_ERRCODE_ACL_ADD_FAILED (0x0036)
```

Unable to add new ACL entry

A.3.6 Common Error Codes for Specific Data Types

Error values with the following code enumeration values may be returned from any function that takes as input a Cryptographic Context Handle.

```
#define CSSM_ERRCODE_INVALID_CONTEXT_HANDLE (0x0040)
```

Invalid context handle

Error values with the following code enumeration values may be returned from any function that takes as input a version.

```
#define CSSM_ERRCODE_INCOMPATIBLE_VERSION (0x0041)
```

Version is not compatible with the current version

Error values with the following code enumeration values may be returned from any function that takes as input the associated input pointer types.

```
#define CSSM_ERRCODE_INVALID_CERTGROUP_POINTER (0x0042)
```

Invalid pointer for certificate group

```
#define CSSM_ERRCODE_INVALID_CERT_POINTER (0x0043)
```

Invalid pointer for certificate

```
#define CSSM_ERRCODE_INVALID_CRL_POINTER (0x0044)
```

Invalid pointer for certificate revocation list

```
#define CSSM_ERRCODE_INVALID_FIELD_POINTER (0x0045)
```

Invalid pointer for certificate or CRL fields (OID/value pairs)

Error values with the following code enumeration values may be returned from any function that takes as input a CSSM_DATA.

```
#define CSSM_ERRCODE_INVALID_DATA (0x0046)
```

The data in an input parameter is invalid

Error values with the following code enumeration values may be returned from any function that takes as input an encoded unsigned CRL.

```
#define CSSM_ERRCODE_CRL_ALREADY_SIGNED (0x0047)
```

Attempted to modify a CRL that has already been signed

Error values with the following code enumeration values may be returned from any function that takes as input a number of fields.

```
#define CSSM_ERRCODE_INVALID_NUMBER_OF_FIELDS (0x0048)
```

Invalid argument for number of certificate fields

Error values with the following code enumeration values may be returned from any function whose operation includes verification of a certificate or CRL.

```
#define CSSM_ERRCODE_VERIFICATION_FAILURE (0x0049)
```

Certificate or CRL verification failed

Error values with the following code enumeration values may be returned from any function that takes as input a DB handle.

```
#define CSSM_ERRCODE_INVALID_DB_HANDLE (0x004A)
```

Invalid database handle

Error values with the following code enumeration values may be returned from any function that deals with privilege.

```
#define CSSM_ERRCODE_PRIVILEGE_NOT_GRANTED (0x004B)
```

Requested privilege has not been granted to the module requesting the privilege

Error values with the following code enumeration values may be returned from any function that takes as input a CSSM_DL_DB_LIST.

```
#define CSSM_ERRCODE_INVALID_DB_LIST (0x004C)
```

Invalid CSSM_DL_DB_LIST.

```
#define CSSM_ERRCODE_INVALID_DB_LIST_POINTER (0x004D)
```

Invalid database list pointer

Error values with the following code enumeration values may be returned from any function that takes as input a certificate template, certificate or CRL.

```
#define CSSM_ERRCODE_UNKNOWN_FORMAT (0x004E)
```

Unknown format or invalid structure for certificate template, certificate or CRL

```
#define CSSM_ERRCODE_UNKNOWN_TAG (0x004F)
```

Unknown OID specified in certificate template, certificate or CRL field

Error values with the following code enumeration values may be returned from any function that takes as input the associated handle.

```
#define CSSM_ERRCODE_INVALID_CSP_HANDLE (0x0050)
```

Invalid CSP handle

```
#define CSSM_ERRCODE_INVALID_DL_HANDLE (0x0051)
```

Invalid DL handle

```
#define CSSM_ERRCODE_INVALID_CL_HANDLE (0x0052)
```

Invalid CL handle

```
#define CSSM_ERRCODE_INVALID_TP_HANDLE (0x0053)
```

Invalid TP handle

```
#define CSSM_ERRCODE_INVALID_KR_HANDLE (0x0054)
```

Invalid KR handle

```
#define CSSM_ERRCODE_INVALID_AC_HANDLE (0x0055)
```

Invalid AC handle

Error values with the following code enumeration values may be returned from any function that takes as input a passthrough ID.

```
#define CSSM_ERRCODE_INVALID_PASSTHROUGH_ID (0x0056)
```

Invalid passthrough ID

Error values with the following code enumeration values may be returned from any function that takes as input a CSSM_NET_ADDRESS.

```
#define CSSM_ERRCODE_INVALID_NETWORK_ADDR (0x0057)
```

The network address is invalid

Error values with the following code enumeration values may be returned from any function that takes as input a CSSM_CRYPTO_DATA.

```
#define CSSM_ERRCODE_INVALID_CRYPTO_DATA (0x0058)
```

The CSSM_CRYPTO_DATA structure is invalid

Appendix B

Application Memory Functions

B.1 Introduction

Memory management between applications and CSSM, and between applications and add-in service providers, differs by necessity. Before selecting and loading a particular version of CSSM, applications are required to provide memory management functions as input to each get-operation. These functions operate on heap space owned by the application. Corresponding free-operations are provided to relieve applications of the burden of walking complex data structures. Once a version of CSSM has been selected and loaded by the application, some memory structures can be allocated on CSSM's heap. Corresponding free-operations are defined, but they must be used by the application to free those structures when they are no longer needed.

When a service provider is selected and attached using *CSSM_ModuleAttach*(), the application must provide a set of memory management functions and heap space for data structures that will be returned by the add-in module as part of its service. When requesting specific services, an application can provide pre-allocated memory as input for the function call. This requires that the application know the memory size required by the service provider. This is not always possible. Supplying a heap and memory management frees the application from specifying memory block sizes. The memory that the application receives is in its heap space. When the application no longer requires the memory, it is responsible for freeing it.

A memory function table is passed from the application to add-in service modules through the *CSSM_xxx_Attach* functions associated with each add-in.

B.2 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 void * (CSSMAPI *CSSM_MALLOC)
    ( uint32 size,
    void * allocref);

typedef void (CSSMAPI *CSSM_FREE)
    (void * memblock,
    void * allocref);

typedef void * (CSSMAPI *CSSM_REALLOC)
    (void * memblock,
    uint32 size,
    void * allocref);

typedef void * (CSSMAPI *CSSM_CALLOC)
    (uint32 num,
    uint32 size,
    void * allocref);
```

```
typedef struct cssm_memory_funcs {
    CSSM_MALLOC malloc_func;
    CSSM_FREE free_func;
    CSSM_REALLOC realloc_func;
    CSSM_CALLOC calloc_func;
    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 the memory heap the function operates on.

Cryptographic Service Provider Behavior

C.1 Introduction

C.1.1 Guidelines for Each Service Provider type

As a growing number of services become available for the CDSA infrastructure, it is important that providers of the same type behave in a common and predictable manner. A set of guidelines for each service provider type must be established to ensure common behavior. Without them, using CDSA provides little or no inherent advantage to application and middleware developers.

This Appendix is designed to serve as a set of behavior guidelines for CDSA Cryptographic Service Provider (CSP) modules. It is written as an "edge specification" relative to the Cryptographic Service Provider SPI, in that it specifies the apparent behavior of the CSP through the SPI interface, but does not specify how the underlying implementation must be done. In this case, the edge is the boundary between the CSSM and the Service Provider, although it also makes references to the boundary between the application and the CSSM, so in this respect it is also an interface specification.

No examples of interface usage are provided in this Appendix, since it is not a *"how-to"* guide.

C.1.2 Typographic Conventions

When referring to a field inside of a complex structure, the following notation is used:

STRUCTURE_TYPE::FieldName

For example, the *AlgorithmId* field in the CSSM_KEYHEADER structure is referenced as:

CSSM_KEYHEADER::AlgorithmId

C.2 Formats

C.2.1 Key Formats

CDSA Cryptographic Service Providers (CSP) represent keys in three ways:

- plaintext blobs
- indirect references
- wrapped blobs

Plaintext blobs include PKCS #1 RSA public keys, key references include string names, and wrapped key blobs include PKCS #8 encrypted private key blobs.

The following sections list the CSSM_KEYHEADER field values, and the corresponding *CSSM_KEY::KeyData* contents that they describe.

C.2.1.1 Plaintext Keys

Table C-1 on page 939 describes the values of the *CSSM_KEYHEADER* fields and the corresponding plaintext key formats that they represent. A plaintext key format requires that *CSSM_KEYHEADER::BlobType* is equal to CSSM_KEYBLOB_RAW. When *CSSM_KEYHEADER::BlobType* is equal to CSSM_KEYBLOB_RAW, *CSSM_KEYHEADER::WrapAlgorithmId* and *CSSM_KEYHEADER::WrapMode* should be ignored.

Format CSSM_KEYBLOB_ RAW_FORMAT_*	**Valid key types**	**CSSM_KEY::KeyData**
PKCS1	RSA	An RSA public or private key formatted according to the PKCS #1 v2.0 specification.
PKCS3	Diffie-Hellman public key	A Diffie-Hellman public key calculated according to PKCS #3 v1.4. The public key format is the public value as an octet string with the most significant byte first (big-endian).
PKCS3	Diffie-Hellman private key	A Diffie-Hellman private key calculated according to PKCS #3 v1.4. The private key format is the BER encoded sequence of the p, g, and x values.
MSCAPI	All Microsoft CAPI supported	A key formatted according to the Microsoft CAPI 2.0 keyblob specification.
PGP	All PGP supported	A key formatted according to the PGP keyring specification.
FIPS186	DSA	A DSA key formatted as a DER sequence of p, q, g, and either y or x for public or private keys respectively.
BSAFE	All	A key formatted in the RSA BSafe key format.
CCA	All	A key formatted according to the IBM CCA specification.
PKCS8	All private keys	A private key formatted according to PKCS #8 v1.2, without encryption.

SPKI	All asymmetric keys	A public or private key formatted according to the SPKI specification.
OCTET_STRING	All symmetric keys	A symmetric key represented as an octet string, with the most significant byte first (big-endian).
OTHER	Vendor specific	A key formatted according to individual vendor requirements. See individual documentation for format details. This format is not recommended for most CSPs.

Table C-1 Plaintext Key Format Descriptor Values for CSSM_KEYHEADER

Every key type has a default format that is used in the cases where the caller does not specify a specific format type. Table C-2 below lists the defined CSSM algorithms and their default plaintext key formats.

AlgorithmId CSSM_ALGID_*	**Format CSSM_KEYBLOB_RAW_FORMAT_***
RSA	PKCS1
DSA	FIPS186
DH	PKCS3
All symmetric key algorithms	OCTET_STRING

Table C-2 Default Plaintext Key Formats

C.2.1.2 Key References

Table C-3 below describes the values of the CSSM_KEYHEADER fields and the corresponding key reference formats that they represent. A key reference format requires that *CSSM_KEYHEADER::BlobType* is equal to CSSM_KEYBLOB_REFERENCE. When *CSSM_KEYHEADER::BlobType* is equal to CSSM_KEYBLOB_REFERENCE, *CSSM_KEYHEADER::WrapAlgorithmId* and *CSSM_KEYHEADER::WrapMode* should be ignored.

Format CSSM_KEYBLOB_ REF_FORMAT_*	**Valid key types**	**CSSM_KEY::KeyData**
INTEGER	All	An integer value of any length, and any byte-sex.
STRING	All	A valid RFC2279 (UTF8) string that corresponds to the "Label" attribute of a key in a DL schema, and the Label parameter to the *CSSM_GenerateKey*( ) or *CSSM_GenerateKeyPair*( ) API. UTF8 degrades to ASCII for english text.
SPKI	All asymmetric keys	A key reference as defined in the SPKI specification. It has the form *(public-key (hash))* or *(private-key (hash))*

OTHER	Vendor specific	A key reference formatted according to individual vendor requirements. See individual documentation for format details.

Table C-3 Key Reference Format Descriptor Values for CSSM_KEYHEADER

Integer references, CSSM_KEYBLOB_REF_FORMAT_INTEGER, are domain specific to a specific service provider, and must not be interpreted by a client. A client must consider a reference of this type to become invalid as soon as the CSP handle used to instantiate the reference is invalidated using the *CSSM_ModuleDetach*( ) API. Even though the reference becomes invalid, the memory for the reference must still be released using the *CSSM_FreeKey*( ) API.

String references, CSSM_KEYBLOB_REF_FORMAT_STRING, are valid for the lifetime of the key that they reference. For instance, if the key is instantiated with the CSSM_KEYATTR_PERMANENT flag, then the reference is valid until *CSSM_FreeKey*( ) is called with the *Delete* parameter set to CSSM_TRUE. If the key is instantiated without the CSSM_KEYATTR_PERMANENT flag, then the key will be destroyed and the reference invalidated by the CSP when the CSP handle used to instantiate the key is detached. As with integer references, the memory for the reference must be released using the *CSSM_FreeKey*( ) API, regardless of whether or not the reference is valid.

SPKI references, CSSM_KEYBLOB_REF_FORMAT_SPKI, have the same lifetimes and deletion requirements as string references.

Vendor specific references, CSSM_KEYBLOB_REF_FORMAT_OTHER, have lifetime behavior that is specific to a certain CSP. They have the same deletion requirements as all other reference types.

The default reference type is CSP specific. The application can safely treat a reference of an unknown type as if it is an integer reference since it has the most restrictive lifetime semantics.

C.2.1.3 Wrapped Keys

Table C-4 below describes the values of the CSSM_KEYHEADER fields and the corresponding wrapped key formats that they represent. A wrapped key format requires that:

- *CSSM_KEYHEADER::BlobType* is equal to CSSM_KEYBLOB_WRAPPED,
- *CSSM_KEYHEADER::WrapAlgorithmId* is not equal to CSSM_ALGID_NONE,
- *CSSM_KEYHEADER::WrapMode* is equal to the appropriate mode, possibly CSSM_ALGMODE_NONE

Format CSSM_KEYBLOB_ WRAPPED_FORMAT_*	**Valid key types**	**CSSM_KEY::KeyData**
PKCS8	All private keys	A private key in formatted and encrypted according to PKCS #8 v1.2. This format is used to wrap asymmetric keys using a symmetric algorithm
PKCS7	All symmetric keys	A symmetric key encrypted as required by PKCS #7 v1.5. This format is used to wrap symmetric keys using either symmetric or asymmetric algorithms.

MSCAPI	All Microsoft CAPI supported	A key formatted and encrypted according to the Microsoft CAPI 2.0 keyblob specification.
OTHER	Vendor specific	A key formatted and encrypted according to individual vendor requirements. See individual documentation for format details. This format is not recommended for most CSPs.

Table C-4 Wrapped Key Format Descriptor Values for CSSM_KEYHEADER

PKCS #7 wrapping, CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS7, wraps only the raw key value of the symmetric key. None of the information in the CSSM_KEYHEADER structure is encryted with the key. All information in the key header of the key to be wrapped is copied to the wrapped key structure, and updated as necessary by the wrapping operation.

PKCS #8 wrapping, CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS8, wraps only the information specified in the PKCS #8 specification. None of the information in the CSSM_KEYHEADER structure is encryted with the key. All information in the key header of the key to be wrapped is copied to the wrapped key structure, and updated as necessary by the wrapping operation.

ASN.1 Structures for PKCS #8 Wrapping

Unlike RSA, some commonly used algorithms do not have an ASN.1 structure defined for use with PKCS #8. PKCS #1 defines the RSAPrivateKey structure for wrapping RSA private keys with PKCS #8. The following text defines the ASN.1 structure used to wrap keys for other algorithms.

Diffie-Hellman Private Keys

Wrapping a Diffie-Hellman private key uses the DHParameter structure defined in PKCS #3 to embed the p and g values into the *privateKeyAlgorithmIdentifier* field of the PrivateKeyInfo structure. The private value, x, is placed in the *privateKey* field BER encoded as BER type INTEGER.

DSA Private Keys

Wrapping a DSA private key uses the Dss-parms structure defined in PKCS #11 v2.01 to embed the p, q, and g values into the *privateKeyAlgorithmIdentifier* field of the PrivateKeyInfo structure. The private value, x, is placed in the *privateKey* field BER encoded as BER type INTEGER.

C.2.2 Requesting Key Format Types

The CSSM API provides a method for requesting a key format that allows varying degrees of control. The method involves specifying two levels of detail that build on each other.

The first level of detail is the easiest to use, and implement in a CSP, but provides only limited amounts of control over the key format. All CSP APIs that instantiate a key object have a parameter *KeyAttr* (*PublicKeyAttr*and*PrivateKeyAttr* for *CSSM_GenerateKeyPair*()). A series of flags is defined to allow the application to define the class of key format to return. The application may choose to receive a CSSM_KEY structure containing a plaintext key, a key reference, or "no data". The CSP may choose to return the key in any format within the specified class.

Table C-5 below lists the defined format flags and the corresponding format class. If more than one of the flags is set, the CSP will return the error CSSM_CSP_INVALID_KEYATTR. If the CSP can not support the requested class of key format, it will return CSSM_CSP_KEY_FORMAT_UNSUPPORTED.

Key Attribute Format Flag	CSP to return format class
CSSM_KEYATTR_RETURN_DATA	Plaintext key
CSSM_KEYATTR_RETURN_REF	Key reference
CSSM_KEYATTR_RETURN_NONE	No data

Table C-5 Key Attribute Format Flags and Corresponding Format Class

The second level of detail for specifying the key format provides fine control over the value returned. Fine control requires that one of the attribute types listed in Table C-6 below is located in the Cryptographic Context (CC) used by the CSP to instantiate the key object. The attribute used depends on the type of key being instantiated. The format types allowed for all unwrapped keys are those that begin with CSSM_KEYBLOB_RAW_FORMAT_* and CSSM_KEYBLOB_REF_FORMAT_* for plaintext keys and key references respectively. The format specifier in the CC attribute must be valid for the format class specified in the *KeyAttr* mask.

Format types allowed for wrapped keys are those that begin with CSSM_KEYBLOB_WRAPPED_FORMAT_*. Note that the *CSSM_WrapKey*( ) API only uses the CC attribute without a key format class flag because the class is implied by the operation.

Cryptographic Context Attribute CSSM_ATTRIBUTE_*	**Valid for APIs**
PUBLIC_KEY_FORMAT	*CSSM_GenerateKeyPair*( )
PRIVATE_KEY_FORMAT	*CSSM_GenerateKeyPair*( ) *CSSM_UnwrapKey*( )
SYMMETRIC_KEY_FORMAT	*CSSM_GenerateKey*( ) *CSSM_UnwrapKey*( ) *CSSM_DeriveKey*( )
WRAPPED_KEY_FORMAT	*CSSM_WrapKey*( )

Table C-6 APIs and the Appropriate Key Format Attributes

If the application chooses "no data", the CSP instantiates the key in its persistent storage but returns no reference. The CSP will return the error CSSM_CSP_INVALID_KEYATTR if "no data" is requested without setting the CSSM_KEYATTR_PERMANENT flag. All key format attributes in a CC are ignored if "no data" is returned. A key wrapping operation may not specify "no data".

C.3 Events

One of the ways that a service provider and the CSSM communicate is by using events. All service providers send events to the CSSM, but CSPs also have the ability to receive events from the CSSM.

C.3.1 Receiving Context Events

The CSSM sends an event to the CSP whenever the client manipulates a cryptographic context (CC). The events indicate that a context is being created, deleted, or updated. The CSSM uses the *CSP_EventNotify*( ) SPI function to send context events to the CSP. For this reason, the *CSP_EventNotify*( ) SPI must be implemented by all CSPs. The reactions to all but the *delete* event are CSP specific, but suggestions are given here.

The *create* event, CSSM_CONTEXT_EVENT_INSERT, is sent to the CSP whenever the client calls one of the CSSM_CSP_Create*Context APIs. The CSSM passes a copy of the new CC and the new CC handle along with the event. At this point, the CSP has the ability to check the CC and reject its contents by returning an error. If the CC is rejected, then the CSSM will fail the context creation call. The CSP is not required to check the CC at this point. If the check is not made when the CC is created, then the check must be performed the first time it is used.

The *update* event, CSSM_CONTEXT_EVENT_UPDATE, is sent to the CSP whenever the client calls the *CSSM_UpdateContextAttributes*( ) or *CSSM_SetContext*( ) APIs. The CSSM sends a copy of the modified CC to the CSP along with the event. At this point, the CSP has the ability to check the modified CC and reject the contents by returning an error. If the CC is rejected, the updates are not applied by the CSSM. The CSP is not required to check the CC at this point. If the check is not made when the context is created, then the check must be performed the first time it is used with the updated attributes. The CSP is not required to accept changes to a CC that is currently being used to perform a staged operation. It can return the error CSSM_CSP_CONTEXT_BUSY to indicate that the changes can not be accepted. If it accepts the changes while the CC is being used, then the changes are not used by the CSP until the current operation completes and the client starts another operation using the context.

The *delete* event, CSSM_CONTEXT_EVENT_DELETE, is sent to the CSP whenever the client calls the *CSSM_DeleteContext*( ) API. The CSSM sends a copy of the current CC to the CSP along with the event. Deleting a CC has the effect of canceling any operations that use it. The CSP must insure that all state information associated with the CC in the CSP is destroyed. The CSP does not have the ability to reject this operation. Deleting a CC always succeeds. An error can be returned to the CSSM, but the operation will always complete.

C.3.2 Sending Insert and Remove Events

A service provider sends *insert* and *remove* events to indicate that a subservice is available or unavailable for use by a client respectively. Clients cannot attach to a subservice until an *insert* event for that subservice has been sent to the CSSM. Once a *remove* event has been sent for a subservice, all module handles to that subservice are invalidated by the CSSM and the client can no longer attach to the subservice until another *insert* event is sent.

A service provider module is required to send an *insert* event, CSSM_NOTIFY_INSERT, to the CSSM for each subservice that it provides regardless of whether or not that subservice can be removed. This allows the CSSM to implement a common interaction model for all service provider modules. If the module implements multiple module types for the same subservice ID, they should both be included in the same *insert* event. If two module types are implemented for the same subservice ID, the services should be related. For instance, a smartcard module will probably provide both CSP and DL services for the same subservice ID. This is a valid re-use of a subservice ID. A module that interfaces to a database and a totally independent cryptographic

library should place those services on different subservice IDs. The CSSM will fail events that break this rule.

The following example shows both correct and incorrect use of the events and subservice IDs. In the examples, *CssmNotifyCallback*() is a function with type *CSSM_API_ModuleEventHandler*() and ADDIN_GUID is the GUID of the service provider module.

```
/* Typical usage. A single service type with a single subservice ID. */
CssmNotifyCallback( &ADDIN_GUID,
                    CssmNotifyCallbackCtx,
                    0,
                    CSSM_SERVICE_CSP,
                    CSSM_NOTIFY_INSERT ); /* Correct */

/* Usage by a module that exports multiple service types using the */
/* SAME subservice ID.                                             */

/* Correct usage, services are related and allowed to share an SSID  */
CssmNotifyCallback( &ADDIN_GUID,
                    CssmNotifyCallbackCtx,
                    1,
                    CSSM_SERVICE_CSP | CSSM_SERVICE_DL,
                    CSSM_NOTIFY_INSERT );

/* Incorrect usage, the second event fails, services are not related */
CssmNotifyCallback( &ADDIN_GUID,
                    CssmNotifyCallbackCtx,
                    2,
                    CSSM_SERVICE_CSP,
                    CSSM_NOTIFY_INSERT );
CssmNotifyCallback( &ADDIN_GUID,
                    CssmNotifyCallbackCtx,
                    2,
                    CSSM_SERVICE_DL,
                    CSSM_NOTIFY_INSERT );
```

If a service provider implements subservices that are always available (like most software-only service providers), then the *insert* events must be sent to the CSSM when the service provider is handling the *CSSM_SPI_ModuleLoad*() interface. The events should be sent every time the *CSSM_SPI_ModuleLoad*() interface is called. It may cause duplicate events to be sent to some event handling functions in the client, but it allows modules loading the service provider for the first time to obtain a complete picture of the available subservices.

Remove events, CSSM_NOTIFY_REMOVE, are used much more sparingly than the corresponding *insert* events. They are used to indicate that a removable device is no longer available. The definition of "remove" is service provider specific but the result is always the same, the subservice is no longer available to the client. *Remove* events should not be sent to the CSSM when the *CSSM_SPI_ModuleUnload*() interface is called. Even though reversing all module loads will cause a subservice (actually, the whole service provider) to become unavailable to the client, it is not a service provider initiated action. Client initiated actions will never trigger a *remove* event.

C.3.3 Sending Fault Events

A *fault* event, CSSM_NOTIFY_FAULT, indicates that a dangerous situation, usually an integrity error, has been detected by the service provider and that the CSSM must immediately sever all connections and cause the operating system to remove the binary from the process space. This has the effect of canceling all *attach* and *load* operations performed on the service provider. The client must attempt to reestablish all connections to the service provider if it requires those services.

The CSSM will most likely avoid calling any interfaces in the service provider after receiving a *fault* event to avoid executing any unstable, or altered code. Therefore, the service provider should insure that it has cleaned up all internal resources before sending the *fault* event to the CSSM.

C.4 Memory Management

CSP service providers have the most complex memory management tasks compared to the other standard CDSA service provider types. Most service providers will allocate the memory for all of the return data, and provide appropriate APIs to release that memory. CSPs have situations where either the client or the service provider can do the memory allocation. The follow subsections describe the methods that should be used by CSPs to handle memory.

C.4.1 Types of Memory Allocation

The CDSA architecture defines two ways for service providers to manage memory. The first is through the use of a local memory heap. The functions that manipulate the local heap are called the *local memory functions*. The second method is to use a set of memory allocation functions in the CSSM that will allocate memory suitable for being returned to the client. These functions are called the *application* memory *functions"*, and are given to the service provider by the CSSM using the *CSSM_SPI_ModuleAttach*() interface.

Service providers should use the local heap for all internal memory requirements. Internal memory requirements include any memory required to perform an operation that will not be returned to the client. This provides higher performance, and reduces exposure of internal data outside of the service provider.

Any memory that will be returned to the client must be allocated using the application memory functions given to the service provider by the CSSM. The memory obtained this way is guaranteed to be allocated in a way that will allow the client to release them when they are no longer needed.

C.4.2 Allocation of Key Information

All functions that return a key allocate a memory buffer for the CSSM_KEY::KeyData field, unless the return type is CSSM_KEYATTR_RETURN_NONE. The CSP will always allocate this buffer, regardless of the state of the CSSM_KEY::KeyData.Data pointer. The client does not have the option to allocate this buffer. The memory must be allocated using the application memory functions.

The *CSSM_FreeKey*() API releases the resources in a CSSM_KEY structure, and optionally deletes the key material from the CSPs internal storage. The CSP will always release the buffer referenced by the CSSM_KEY::KeyData.Data pointer, and set the pointer to NULL. The memory must be released using the application memory functions.

C.4.3 Allocation of Single Output Buffers

In cases where the output from a function is a single buffer, the application has the option of allocating its own memory. The CSP determines whether or not to allocate memory using the application memory functions according to Table C-7 below. The value of CSSM_DATA::Data is shown on the left, and the value of CSSM_DATA::Length is shown across the top.

Length / **Data**	0	>0
NULL	Allocate required memory	Error = CSSM_CSP_INVALID_INPUT_POINTER
!NULL	Error = CSSM_CSP_INVALID_INPUT_POINTER	Use supplied memory Error = CSSM_CSP_INVALID_INPUT_LENGTH if not suitable size

Table C-7 Actions Taken when Returning Values in a Single Buffer

C.4.4 Allocation of Vector-of-Buffers

A vector-of-buffers is an array of CSSM_DATA structures. They are used for both input and output in multiple CSP APIs. The CSP must treat the individual memory buffers in the vector as if they make up a single continuous memory buffer.

When filling an output vector, each component buffer is completely filled before placing any data in the next buffer of the array. None of the length values for the component buffers can be modified by the CSP. The output length parameter that accompanies each output vector must be used to indicate the total number of bytes placed in the output vector.

The CSP should only allocate memory if all of the entries in the vector have a NULL data pointer and a zero length. If memory is allocated, a single output buffer with enough space for the output should be allocated and assigned to the first array element's data pointer. The memory must be allocated using the application memory functions.

C.5 CSP Query Mechanisms

CSPs provide three interfaces for querying the current state of the service provider, or one of the operations being performed.

C.5.1 Querying Key Sizes

The *CSP_QueryKeySizeInBits*( ) SPI (*CSSM_QueryKeySizeInBits*( ) API) is used to get the logical and effective size of a key. The logical size of the key is the number of bits in the key data for a symmetric key, or the size of the key component commonly used to reference the size of an asymmetric key. The effective size of a key is the number of bits in a key that are actually used in the cryptographic operation. For instance, the logical size of a DES key is 64, and the effective size of the same key is 56. Differences in logical and effective sizes will be notes in Section C.9 on page 966.

The key can be supplied to the function directly or in a cryptographic context. The size of the key must be calculated by looking at the key data itself. The key header information should not be used to calculate the values.

C.5.2 Querying Output Sizes

The *CSP_QuerySize*( ) SPI (*CSSM_QuerySize*( ) API) is used to query the size of the output buffer required to hold the result of a cryptographic operation. The CSP must take into account any applicable information including the block size, number of bytes required to complete a block, padding data, the amount of the data being input to the function, and any other algorithm specific information. If the algorithm has a block size, then the size reported may be up to one block size larger than the actual result.

Table C-8 below describes the behavior for each supported cryptographic operation type. The CSP returns the error CSSM_CSP_INVALID_CONTEXT if a query is made for an unsupported operation.

Operation Type	Encrypt Parameter	Result
Encrypt	CSSM_TRUE	Returns the output size of the next operation. If the cryptographic context (CC) used to perform the operation is currently being used to perform a staged operation, then the output is based on the current state of the algorithm. If the CC is not being used to perform a staged operation, then the result is based on a single stage operation.
Decrypt	CSSM_FALSE	
Digest	No effect	Always returns the length of the digest output regardless of the algorithm state. This value is also available by fetching the CSSM_ATTRIBUTE_OUTPUT_SIZE attribute for the corresponding capability information in the MDS.
Generate MAC	CSSM_TRUE	Always returns the length of the signature output regardless of the algorithm state. For MAC algorithms that always generate the same size signature, this value is also available by fetching the CSSM_ATTRIBUTE_OUTPUT_SIZE attribute for the corresponding capability information in the MDS.
Verify MAC	CSSM_FALSE	Always returns zero.

Sign	CSSM_TRUE	Always returns the length of the signature output regardless of the algorithm state. For signature algorithms that always generate the same size signature (i.e. DSA), this value is also available by fetching the CSSM_ATTRIBUTE_OUTPUT_SIZE attribute for the corresponding capability information in the MDS.
Verify	CSSM_FALSE	Always returns zero.

Table C-8 Behavior of CSP_QuerySize for all supported Operation Types

C.5.3 Querying State of the CSP Subservice

The current state of the CSP subservice can be determined using the *CSP_GetOperationalStatistics*( ) SPI (*CSSM_CSP_GetOperationalStatistics*( ) API). Table C-9 describes the meaning of each field in the CSSM_CSP_OPERATIONAL_STATISTICS structure.

Field Name	Description
UserAuthenticated	This field will be set to CSSM_TRUE if the client has authenticated to the subservice using the *CSP_Login*( ) SPI (*CSSM_CSP_Login*( ) API).
DeviceFlags	This field is described in detail in Table C-10 on page 950.
TokenMaxSessionCount	Indicates the maximum number of simultaneous attach handles that can be issued for the subservice.
TokenOpenedSessionCount	Indicates the number of attach handles that have already been issued for the subservice.
TokenMaxRWSessionCount	Indicates the maximum number of simultaneous read/write attach handles that can be issued for the subservice. By default, attach handles are created in read/write mode.
TokenOpenedRWSessionCount	Indicates the number of read/write attach handles that have already been issued for the subservice.
TokenTotalPublicMem	Indicates the amount of storage space that can be used to store objects that are usable without the client authenticating to the subservice using the *CSP_Login*( ) SPI.
TokenFreePublicMem	Indicates the amount of storage space that has been used to store objects that are usable without the client authenticating to the subservice using the *CSP_Login*( ) SPI.
TokenTotalPrivateMem	Indicates the amount of storage space that can be used to store objects that are not usable without the client authenticating to the subservice using the *CSP_Login*( ) SPI.
TokenFreePrivateMem	Indicates the amount of storage space that has been used to store objects that are not usable without the client authenticating to the subservice using the *CSP_Login*( ) SPI.

Table C-9 CSSM_CSP_OPERATIONAL_STATISTICS Structure

Any of the Token* fields can have the value CSSM_VALUE_NOT_AVAILABLE to indicate that the CSP does not use these values, or that it will not reveal the values. The values are primarily applicable to hardware cryptographic devices.

The *DeviceFlags* field can be a mask of the values listed in Table C-10 below.

Field Name	Description
CSSM_CSP_TOK_WRITE_PROTECTED	The meaning of this flag varies by service provider. In some service providers, no objects may be created, deleted, or modified. In others, permanent objects may not be manipulated but transient objects may be freely created.
CSSM_CSP_TOK_LOGIN_REQUIRED	The CSP requires the client to authenticate to the subservice before cryptographic operations other than random number generation and message digest calculation.
CSSM_CSP_TOK_USER_PIN_INITIALIZED	If the subservice requires authentication before cryptographic operations can be performed, it indicates that the authentication data for the client has been initialized.
CSSM_CSP_TOK_PROT_AUTHENTICATION	The subservice provides a protected mechanism for collecting authentication information.
CSSM_CSP_TOK_USER_PIN_EXPIRED	If the subservice requires authentication before cryptographic operations can be performed, it indicates that the authentication data for the client has expired and must be changed before it can be used.
CSSM_CSP_TOK_SESSION_KEY_PASSWORD	Indicates that the subservice supports separate authentication data for individual symmetric keys.
CSSM_CSP_TOK_PRIVATE_KEY_PASSWORD	Indicates that the subservice supports separate authentication data for individual private keys.
CSSM_CSP_STORES_PRIVATE_KEYS	Indicates that private keys can be stored permanently by the subservice.
CSSM_CSP_STORES_PUBLIC_KEYS	Indicates that public keys can be stored permanently by the subservice.
CSSM_CSP_STORES_SESSION_KEYS	Indicates that session keys can be stored permanently by the subservice.
CSSM_CSP_STORES_CERTIFICATES	Indicates that certificates can be stored permanently by the subservice. This flag can only be set by a multi-service module that also implements a DL interface.
CSSM_CSP_STORES_GENERIC	Indicates that generic data objects can be stored permanently by the subservice. This flag can only be set by a multi-service module that also implements a DL interface.

Table C-10 CSSM_CSP_OPERATIONAL_STATISTICS::DeviceFlags Field

C.6 Client Authentication and Authorization

Client authentication and authorization in Cryptographic Service Providers (CSPs) has two forms:

- Client login ACLs
- Individual key ACLs

Many CSPs will implement one form or the other, but they are not mutually exclusive. Each authentication mechanism has its own functional properties that make it useful.

C.6.1 Client Login ACLs

Client login ACLs protect resources in the CSP that affect the subservice as a whole. All authorizations granted to a subservice using the *CSP_Login*( ) SPI (*CSSM_CSP_Login*( ) API) can be utilized using any CSP handle issued to the same process[2]. Authorizations granted using client login ACLs are cancelled in the following situations:

1. The *CSP_Logout*( ) SPI (*CSSM_CSP_Login*( ) API) is called the same number of times as the *CSP_Login*( ) SPI
2. All CSP handles to the subservice in the process have been invalidated using the *CSSM_SPI_ModuleDetach*( ) SPI (*CSSM_ModuleDetach*( ) API).

If the CSP supports client login ACLs, then it must set the CSSM_CSP_TOK_LOGIN_REQUIRED flag in the CSSM_CSP_OPERATIONAL_STATISTICS::DeviceFlags field.

Client login ACLs control the visibility of some keys stored in the CSP. Keys with the CSSM_KEYATTR_PRIVATE flag set in their CSSM_KEY::KeyHeader.KeyAttr field are not visible to the client until it has authenticated to the subservice. As a result, functions such as *CSP_ObtainPrivateKeyFromPublicKey*( ) will not succeed in finding private keys with the CSSM_KEYATTR_PRIVATE flag set until the client has authenticated using *CSP_Login*( ).

Client login ACLs also control the ability to use the subservice for cryptographic operations. If the CSSM_CSP_TOK_LOGIN_REQUIRED flag is set, then the client must authenticate to the subservice before operations other than random number generation and message digesting can be performed.

C.6.1.1 Managing Client Login ACLs

ACL Contents

The contents of a client login ACL can be fetched using the *CSP_GetLoginAcl*( ) SPI (*CSSM_CSP_GetLoginAcl*( ) API). The CSP must filter the set of returned ACL entries based on the *SelectionTag* parameter. The CSP must allocate all memory for the ACL information structures using the application memory functions, and fill in the *NumberOfAclInfos* with the number of allocated structures. If either the *AclInfos* or *NumberOfAclInfos* parameters are NULL, then the CSP will return the error CSSM_CSP_INVALID_OUTPUT_POINTER. If there are no matching ACL information structures, then the CSP sets the *NumberOfAclInfos* parameter to zero and leaves the *AclInfos* value untouched. If the CSP does not support client login ACLs, then it

2. A *process* is defined as all threads of execution sharing the same addressable memory space.

returrns the error CSSM_CSP_OBJECT_ACL_NOT_SUPPORTED.

The contents of a client login ACL can be modified by the ACL owner using the *CSP_ChangeLoginAcl*( ) SPI (*CSSM_CSP_ChangeLoginAcl*( ) API). If the requested edit mode is not supported by the CSP, then it returns CSSM_CSP_INVALID_ACL_EDIT_MODE.

ACL Owner

The subject of the ACL owner is the entity that has the right to change the contents of the client login ACL. The value of the ACL owner subject is initialized in a CSP specific manner outside the scope of the CSP SPI.

The contents of a client login owner ACL can be fetched using the *CSP_GetLoginOwner*( ) SPI (*CSSM_CSP_GetLoginOwner*( ) API). The CSP must allocate all memory for the ACL owner information structure using the application memory functions. If the *Owner* parameter is NULL, then the CSP returns the error CSSM_CSP_INVALID_OUTPUT_POINTER. If the CSP does not support client login ACLs, then it returns the error CSSM_CSP_OBJECT_ACL_NOT_SUPPORTED.

The contents of a client login ACL owner can be modified by the subject of the ACL owner using the *CSP_ChangeLoginOwner*( ) SPI (*CSSM_CSP_ChangeLoginOwner*( ) API). If the requested edit mode is not supported by the CSP, then it returns CSSM_CSP_INVALID_ACL_EDIT_MODE.

C.6.2 Individual Key ACLs

Individual key ACLs protect and control the usage of individual symmetric or private keys. All authorizations granted to a client for a key are only valid until a single operation using the key has completed. Authorizations granted using individual key ACLs are cancelled in the following situations:

1. The operation using the key completes
2. The cryptographic context that uses the key is deleted

If the CSP supports individual key ACLs, then it must set at least one of the CSSM_CSP_TOK_SESSION_KEY_PASSWORD and CSSM_CSP_TOK_PRIVATE_KEY_PASSWORD flags in the CSSM_CSP_OPERATIONAL_STATISTICS::DeviceFlags field to indicate what key types can be protected using individual key ACLs. Public keys never require ACLs.

Individual key ACLs control the ability of clients to use the key for a specific purpose. Since individual key ACLs control the use of the key they protect, the operations listing in the ACL must match the list of operations found in the CSSM_KEY::KeyHeader.KeyUsage field (or specified to a key creation API).

ACL Contents

The contents of an individual key ACL can be fetched using the *CSP_GetKeyAcl*( ) SPI (*CSSM_CSP_GetKeyAcl*( ) API). The CSP must filter the set of returned ACL entries based on the *SelectionTag* parameter. The CSP must allocate all memory for the ACL information structures using the application memory functions, and fill in the *NumberOfAclInfos* with the number of allocated structures. If either the *AclInfos* or *NumberOfAclInfos* parameters are NULL, then the CSP returns the error CSSM_CSP_INVALID_OUTPUT_POINTER. If there are no matching ACL information structures, then the CSP sets the *NumberOfAclInfos* parameter to zero and leaves the *AclInfos* value untouched. If the CSP does not support individual key ACLs, then it returns the error CSSM_CSP_OBJECT_ACL_NOT_SUPPORTED.

The contents of an individual key ACL can be modified by the ACL owner using the *CSP_ChangeKeyAcl*( ) SPI (*CSSM_CSP_ChangeKeyAcl*( ) API). If the requested edit mode is not supported by the CSP, then it returns CSSM_CSP_INVALID_ACL_EDIT_MODE.

ACL Owner

The subject of the ACL owner is the entity that has the right to change the contents of the client login ACL. The value of the ACL owner subject is initialized to the ACL subject specified in the initial ACL for the key.

The contents of a client login owner ACL can be fetched using the *CSP_GetKeyOwner*( ) SPI (*CSSM_CSP_GetKeyOwner*( ) API). The CSP must allocate all memory for the ACL owner information structure using the application memory functions. If the *Owner* parameter is NULL, then the CSP returns the error CSSM_CSP_INVALID_OUTPUT_POINTER. If the CSP does not support client login ACLs, then it returns the error CSSM_CSP_OBJECT_ACL_NOT_SUPPORTED.

The contents of an individual key ACL owner can be modified by the subject of the ACL owner using the *CSP_ChangeKeyOwner*( ) SPI (*CSSM_CSP_ChangeKeyOwner*( ) API). If the requested edit mode is not supported by the CSP, then it returns CSSM_CSP_INVALID_ACL_EDIT_MODE.

C.6.3 Protected Authentication Paths

Some CSPs have the ability to collect authentication information using a method that does not reveal the information to the host system. These methods are usually only found in CSPs that are implemented in hardware or as hybrid hardware/software modules. When an ACL indicates that the subject of an authorization group is a *protected* value, such as a *protected password*, the client must supply a CSSM_ACCESS_CREDENTIALS structure listing the subject type but no value, or a CSSM_ACCESS_CREDENTIALS structure that only specifies a challenge callback. A CSP never calls a challenge callback to obtain a *protected* value.

C.7 Module Directory Service Information

The Module Directory Service (MDS) contains information about each service provider. The information is used by the CSSM and its clients to locate a service provider and determine its capabilities. This section lists each MDS relation related to a CSP and describes how the record or records for the service provider must be populated. If the service provider implements dynamic subservices, there is also a description of how each relation must be updated when handling module events.

C.7.1 Common Relation

Every service provider, regardless of type, must register a single record in this relation. When a client calls *CSSM_ModuleLoad*( ) the CSSM searches this relation for a record with a matching *ModuleId* GUID value. When it finds the record it uses the *ModuleName*, *ModulePath*, and *Manifest* fields to verify and load the module.

Field Name	Contents
ModuleId	GUID (in string format) uniquely identifying the service provider.
Manifest	Signed manifest describing the service provider binary. If the service provider implements the Module Services and Administration (MSA) interfaces (i.e. *CSSM_SPI_Load*( ), *CSSM_SPI_Attach*( )) and the module interface functions (i.e. *CSP_SignData*( )) in different binaries, then this manifest describes the binary that implements the MSA functions.
ModuleName	File name of the service provider binary without the system path information.
Path	List of system paths where the module might be found. The format of the list is system dependent. The paths will be searched in order until the module is found. This path is used to find all binaries required for the service provider.
Version	CSSM interface version implemented by the service provider. This must be a text representation of the CSSM_MAJOR and CSSM_MINOR constants in the form "X.Y".
Desc	Human readable text string describing the service provider.
DynamicFlag	Boolean flag indicating whether or not the service provider implements dynamic services. Zero indicates static services and a non-zero value indicates dynamic services. See Section C.3.2 on page 943 (Sending Insert and Remove Events) for more information on dynamic services.
MultiThreadFlag	Boolean flag indicating whether or not the CSSM should allow multiple threads to call the service provider at the same time. Zero indicates that the CSSM should synchronize all threads making requests to the service provider, non-zero indicates that any number of threads are allowed to execute at the same time. **Note:** This functionality is deprecated and may be removed in future implementations of the CSSM. All service providers should insure thread-safe operation on their own and set this value to a non-zero value. In the situation where multiple CSSM modules are present in the same process, there is no way for the modules to synchronize their operation and guarantee that only a single thread is executing in the service provider. The result is that the flag limits the number of threads to one per loaded CSSM.
ServiceMask	Service Mask of all service types supported by the service provider. The value of the mask is the bitwise logical OR of constants defined for the CSSM_SERVICE_MASK type.

Table C-11 Contents of CDSA Common MDS Relation

This record should remain static once the service provider is installed.

C.7.2 CSP Primary Relation

Service provider modules that export a CSP interface must register a single record in this relation for each CSP subservice that they export. When a client calls *CSSM_ModuleAttach*( ) for a CSP, the CSSM searches this relation for a record with a matching *ModuleId* GUID and *SSID* subservice ID value. When it finds the record it uses the *ModuleName* and *Manifest* fields along with the *ModulePath* field from the CDSA Common Relation to verify and load the subservice binary for the module. If the service provider is implemented using a single binary, then the CSSM simply uses the library that is already loaded instead of attempting to reload the binary a second time.

Software CSPs that implement a single static subservice should use a *i* SSID value of zero. CSPs that implement dynamic subservices can use any *SSID* including zero. A CSP should typically allocate a new *SSID* value for each logical service. For instance, a smartcard CSP should allocate an *SSID* value for each card reader slot present on the system.

Field Name	Description
ModuleID	GUID (in string format) uniquely identifying the service provider. This value must be the same as the *ModuleId* field in the CDSA Common Relation record.
SSID	Subservice ID. This value can be any unsigned 32-bit value. If the CSP implements a single static subservice, this value should be zero. If the CSP implements one or more dynamic subservices, then this value can be any value including zero.
Manifest	Signed manifest describing the service provider binary that implements the subservice. If the service provider implements the Module Services and Administration (MSA) interfaces (i.e. *CSSM_SPI_Load*( ), *CSSM_SPI_Attach*( )) and the module interface functions (i.e. *CSP_SignData*( )) in different binaries, then this manifest describes the binary that implements the CSP functions for the subservice. If the entire service provider is implemented as a single binary, the most common case, then this value should be a copy of the value in the CDSA Common Relation record.
ModuleName	File name of the service provider binary without the system path information. The *ModulePath* value in the CDSA Common relation record will be used as the search path to find the binary.
Version	CSSM interface version implemented by the service provider. This must be a text representation of the CSSM_MAJOR and CSSM_MINOR constants in the form "X.Y".
Vendor	Service provider vendor name in ASCII text.
CspType	CSP implementation type, such as software, hardware, or hybrid. The value corresponds to constants of type CSSM_SERVICE_MASK.
CspFlags	CSP descriptions flags (32-bits). The flags are a bitwise logical OR of the constants defined for CSSM_CSP_FLAGS.
CspCustomFlags	CSP specific flags (32-bits).
UseeTags	Array of 32-bit integers containing the privilege values supported by the CSP. The integers correspond to symbols of type CSSM_PRIVILEGE.
SampleTypes	An array of 32-bit integers representing the sample types accepted by the CSP. The integers correspond to symbols of type CSSM_SAMPLE_TYPE
AclSubjectTypes	An array of 32-bit integers representing the ACL subject types accepted by the CSP. The integers correspond to symbols of type CSSM_ACL_SUBJECT_TYPE

AuthTags	An array of 32-bit integers representing the authorization tag values defined by the CSP. The integers correspond to symbols of type CSSM_ACL_AUTHORIZATION_TAG

Table C-12 Contents of the CSP Primary MDS Relation

This record should remain static once the service provider is installed. An exception to this rule might be when a user installs a new reader slot supported by a smartcard CSP.

C.7.3 CSP Encapsulated Product Relation

Service provider modules that export a CSP interface must register a single record in this relation for each CSP subservice that they export. It contains information about the underlying functionality provided by the specific CSP subservice. Service providers that are not wrappers around other products should create these records to insure that a complete set of information is available to clients. It is not used by the CSSM. Some of the information includes the product version, standard and standard version implemented by the service provider, and token reader device information (if applicable).

Field Name	Description
ModuleID	GUID (in string format) uniquely identifying the service provider. This value must be the same as the *ModuleId* field in the corresponding CSP Primary Relation record.
SSID	Subservice ID. This must match the *SSID* value of the corresponding CSP Primary Relation record.
ProductDesc	ASCII text description of the product encapsulated by the CSP implementation.
ProductVendor	ASCII text description of the vendor that produces the product encapsulated by the CSP implementation.
ProductVersion	Version string in dotted high/low format (i.e. 2.0) indicating the version of the product encapsulated by the CSP implementation.
ProductFlags	Product flags (32-bits). The definition of the flag values is product specific.
CustomFlags	Custom flags (32-bits). The definition of the flag value is product specific.
StandardDesc	String describing the standard that the implementation encapsulates (i.e. PKCS #11)
StandardVersion	Version string in dotted high/low format (i.e. 2.0) indicating the version of the standard implemented by the encapsulated product.
ReaderDesc	If the product encapsulates a hardware device that includes a reader device, this is an ASCII text description of the reader. This field should be empty (" ") if there is no hardware reader.
ReaderVendor	If the product encapsulates a hardware device that includes a reader device, this is an ASCII text description of the reader vendor. This field should be empty ("") if there is no hardware reader device.
ReaderVersion	Version string in dotted high/low format (i.e. 2.0) indicating the version of the hardware reader device. This field should be empty (" ") if there is no hardware reader device.
ReaderFirmwareVersion	Version string in dotted high/low format (i.e. 2.0) indicating the firmware version of the hardware reader device. This field should be empty (" ") if

	there is no hardware reader device.
ReaderFlags	Reader flags (32-bits). The flags are a bitwise logical OR of the constants defined for CSSM_CSP_READER_FLAGS.
ReaderCustomFlags	Reader custom flags (32-bits). The definition of the flag value is product specific.
ReaderSerialNumber	ASCII representation of the reader device serial number. This field should be empty (" ") if there is no hardware reader device.

Table C-13 Contents of CSP Encapsulated Product MDS Relation

This record should updated to reflect correct information whenever the corresponding CSP Primary Relation record is updated.

C.7.4 CSP Smartcard Relation

Service providers that export a CSP interface to a hardware device must register a record in the CSP Smartcard relation for each applicable subservice. All hardware devices, regardless of form-factor or dynamic properties must create these records. CSPs that are implemented in software are not required to create records in this relation.

Field Name	Comment
ModuleID	GUID (in string format) uniquely identifying the service provider. This value must be the same as the *ModuleId* field in the corresponding CSP Primary relation record.
SSID	Subservice ID. This must match the SSID value of the corresponding CSP Primary relation record.
ScDesc	ASCII text description of the subservice device.
ScVendor	ASCII text description of the subservice device vendor.
ScVersion	Version string in dotted high/low format (i.e. 2.0) indicating the product version of the subservice device. This field should be empty (" ") if there is no version information available.
ScFirmwareVersion	Version string in dotted high/low format (i.e. 2.0) indicating the firmware version of the subservice device. This field should be empty (" ") if there is no version information available.
ScFlags	Subservice device flags (32-bits). Indicates the set of built-in features of the subservice device. The features should be implemented in hardware.
ScCustomFlags	Custom subservice flags (32-bits). The definition of the flag value is product specific.
ScSerialNumber	ASCII text representation of the subservice device serial number.

Table C-14 Contents of CSP Smartcard MDS Relation

The contents of the CSP Smartcard relation record for a subservice can be updated at any time by the CSP. It must be updated to reflect current information before the *insert* event is sent to the CSSM, and it must remain valid until the *remove* event is sent or the module is unloaded from all processes. See Section C.3.2 on page 943 (Sending Insert and Remove Events) for more information on insert events.

C.7.5 CSP Capabilities Relation

A CSP service provider will enter multiple record in the CSP Capabilities relation for each cryptographic operation supported by a subservice. Table C-15 describes the fields for each attribute. The interpretation of the *AttributeValue* field varies based on the *AttributeType* field. Later subsections will describe the list of attributes that are required for each type of cryptographic operation, as well as other optional attributes. The *UseeTag* field allows a CSP to reflect additional capabilities based on a client's privileges granted in its signed manifest. The proper method for reflecting the changes in available capabilities is described in later sections.

Field Name	Comment
ModuleID	GUID (in string format) uniquely identifying the service provider. This value must be the same as the *ModuleId* field in the corresponding CSP Primary relation record.
SSID	Subservice ID. This must match the SSID value of the corresponding CSP Primary relation record.
UseeTag	32-bit privilege tag associated with the attribute values. The values correspond to constants of type CSSM_USEE_TAG.
ContextType	Class of cryptographic operation described by the attribute (32-bit). The value corresponds to constants of type CSSM_CONTEXT_TYPE.
AlgType	Cryptographic algorithm supported by the CSP (32-bit). The value corresponds to constants of type CSSM_ALGORITHMS.
GroupId	32-bit identifier grouping all of the attributes associated with a single cryptographic capability.
AttributeType	CSP attribute tag to identify the attribute value (32-bit). The value corresponds to constants of type CSSM_ATTRIBUTE_TYPE.
AttributeValue	Array of 32-bit values. Some attribute types can have multiple values.
Description	ASCII text, human readable description of the algorithm (*AlgType*). All attributes with the same *GroupId* value should have the same *Description* value.

Table C-15 Contents of CSP Capabilities MDS Relation

The contents of the CSP Capabilities relation records for a subservice can be updated at any time by the CSP. It must be updated to reflect current information before the *insert* event is sent to the CSSM, and it must remain valid until the *remove* event is sent or the module is unloaded from all processes. See Section C.3.2 on page 943 (Sending Insert and Remove Events) for more information on insert events.

C.7.5.1 Assigning GroupId Values

Attributes that describe the same cryptographic capability are bound together using a group identifier, the *GroupId* field. The group identifier is an arbitrary unsigned integer that has no meaning other than to bind related attributes.

For simplicity, and to allow easier searches for all attributes of a single capability, all capabilities must have a unique *GroupId* value. The values should range from zero to N-1 without gaps in the sequence, where N is the number of capabilities supported by the CSP.

C.7.5.2 Privileged Capabilities

Some CSP service providers may elect to restrict some of their capabilities so that they can only be used by clients with the appropriate privileges granted in their signed manifests. The *UseeTag* field is used to indicate that the attribute only applies when the client's privilege is equal to a specific privilege value. The value CSSM_USEE_NONE indicates that the capability is available using any privilege value. This means that any client, including those that have not been granted special privileges in their signed manifest, can use the capability.

To indicate privilege restrictions on capabilities, the attributes are entered into the MDS in two "phases". The first phase is the attributes for capabilities supported without special privileges from the client (CSSM_USEE_NONE), otherwise known as the base attributes. The second phase is the deltas to those capabilities available to clients that have been granted special privilege (i.e. CSSM_USEE_FINANCIAL). The two phases build on each other, so that a CSP can reflect upgrades in a capability due to client privilege.

Table C-16 illustrates a set of attributes for a CSP that implements RSA key generation, SHA-1 message digests and RSA key exchange. By looking at the attributes with a CSSM_USEE_NONE *UseeTag* value, you can tell that SHA-1 message digests can always be calculated regardless of the privilege used by a client attached to the CSP. You can also tell that the CSP implements 512-bit RSA key generation and 512-bit key exchange at any time. By looking at the attributes with a CSSM_USEE_KEYEXCH *UseeTag* value, you can tell that the CSP allows 1024-bit RSA key generation and 1024-bit key exchange when the client has been granted that privilege. Notice that the deltas in capabilities, the additional key size, use the same *GroupId* value as the base set of attributes.

UseeTag	GroupId	ContextType	AlgType	AttributeType	AttributeValue
NONE	0	KEYGEN	RSA	KEYSIZE	512
KEYEXCH	0	KEYGEN	RSA	KEYSIZE	1024
NONE	1	DIGEST	SHA1	OUTPUT_SIZE	20
NONE	2	ASYMMETRIC	RSA	KEYSIZE	512
KEYEXCH	2	ASYMMETRIC	RSA	KEYSIZE	1024

Table C-16 Example Representation of Capabilities Set for CSP in MDS

C.7.5.3 Required Capability Attributes

Each type of cryptographic capability has a set of required attribute types, and an additional set of optional values depending on the specific algorithm. The following subsections describe the required and optional attributes for each capability type.

If a capability does not have any required attributes and no optional attributes are present, a single attribute must be inserted with the following fixed values.

Field Name	Value
AttributeType	CSSM_ATTRIBUTE_NONE
AttributeValue	0

Table C-17 Fixed Attribute Values for No Required Attributes

Random Number Generation Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
BLOCK_SIZE	If the RNG algorithm uses a block generation method, this value can be used to indicate the size of each block.
ROUNDS_RANGE	If the RNG can be configured to use a variable number of rounds to generate random data, this attribute indicates the valid range of round values that are supported by the CSP.
ROUNDS	This attribute must be specified if the ROUNDS_RANGE attribute is present. It indicates the default rounds value used by the CSP if this attribute is not found in the cryptographic context.

Table C-18 Capability Attributes for Random Number Generation

Message Digest Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
OUTPUT_SIZE	This is the output length of the message digest algorithm in bytes.

Table C-19 Capability Attributes for Message Digest Capabilities

Symmetric Key Generation Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits.
Optional Attributes	
KEYUSAGE	Indicates the key usage mask values that can be specified for the new key.

Table C-20 Capability Attributes for Symmetric Key Generation

Symmetric Block Cipher Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
MODE	Indicates the algorithm mode described by this set of attributes. Each algorithm mode should be described by its own capability differentiated using the *GroupId* field.
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits.
BLOCK_SIZE	Block size of the cipher. If the cipher is used without a padding method, the length of the input must be a multiple of this value. If a padding method is used, the total output length from the cipher will be a multiple of this value.
Optional Attributes	
PADDING	List of padding methods supported for this cipher and mode.
ROUNDS_RANGE	If the cipher can be configured to use a variable number of rounds to transform data, this attribute indicates the valid range of round values that are supported by the CSP.
ROUNDS	This attribute must be specified if the ROUNDS_RANGE attribute is present. It indicates the default rounds value used by the CSP if this attribute is not found in the cryptographic context.
IV_SIZET	If the algorithm and mode requires and initialization vector, this value indicates the required length of the value that must be supplied by the client.

Table C-21 Capability Attributes for Symmetric Block Cipher

Symmetric Stream Cipher Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits.
Optional Attributes	
MODE	Indicates the algorithm mode described by this set of attributes. Each algorithm mode should be described by its own capability differentiated using the *GroupId* field.
ROUNDS_RANGE	If the cipher can be configured to use a variable number of rounds to transform data, this attribute indicates the valid range of round values that are supported by the CSP.
ROUNDS	This attribute must be specified if the ROUNDS_RANGE attribute is present. It indicates the default rounds value used by the CSP if this attribute is not found in the cryptographic context.
KEYUSAGE	Indicates the key usage mask values that can be specified for the new key.

Table C-22 Capability Attributes for Symmetric Stream Cipher

Message Authentication Code Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
MODE	Indicates the algorithm mode described by this set of attributes. Each algorithm mode should be described by its own capability differentiated using the *GroupId* field.
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits.
BLOCK_SIZE	Block size of the cipher used to generate the MAC. Optional Attributes
Optional attributes	
ROUNDS_RANGE	If the cipher can be configured to use a variable number of rounds to transform data, this attribute indicates the valid range of round values that are supported by the CSP.
ROUNDS	This attribute must be specified if the ROUNDS_RANGE attribute is present. It indicates the default rounds value used by the CSP if this attribute is not found in the cryptographic context.
OUTPUT_SIZE	Length of the MAC value generated by the algorithm.
IV_SIZE	If the algorithm and mode requires an initialization vector, this value indicates the required length of the value that must be supplied by the client.

Table C-23 Capability Attributes for Message Authentication Code

Asymmetric Key Generation Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits
Optional Attributes	
KEYUSAGE	Indicates the key usage mask values that can be specified for the new keys. The mask is a combination of the values that are valid for the public and private keys.

Table C-24 Capability Attributes for Asymmetric Key Generation

Asymmetric Encryption Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	Description
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.

KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits.
Optional Attributes	
MODE	Indicates the algorithm mode described by this set of attributes. Each algorithm mode should be described by its own capability differentiated using the *GroupId* field.

Table C-25 Capability Attributes for Asymmetric Encryption

Asymmetric Signature Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	**Description**
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits
Optional Attributes	
MODE	Indicates the algorithm mode described by this set of attributes. Each algorithm mode should be described by its own capability differentiated using the *GroupId* field.

Table C-26 Capability Attributes for Asymmetric Signature

Key Derivation Capabilities

Optional Attributes CSSM_ATTRIBUTE_*	**Description**
KEY_LENGTH_RANGE	If the algorithm supports multiple key sizes, this is the range of key sizes supported in bits.
KEY_LENGTH	If the algorithm uses a fixed key size, this is the logical key size in bits. lb s l l. Optional Attributes ROUNDS_RANGE!T{ If the key derivation algorithm can be configured to use a variable number of rounds to transform data, this attribute indicates the valid range of round values that are supported by the CSP.
ROUNDS	This attribute must be specified if the ROUNDS_RANGE attribute is present. It indicates the default rounds value used by the CSP if this attribute is not found in the cryptographic context.
KEY_TYPE	List of algorithm identifiers (CSSM_ALGORITHMS) indicating the types of keys that can be created by the derivation mechanism.

Table C-27 Capability Attributes for Key Derivation

C.8 CSP Multi-Service Modules with DL Interface

CSPs are unique among CDSA service providers in that they take on extra capabilities when combined with service providers of other types. Examples of service provider type combinations that produce extra capabilities are CSPs with Key Recovery (KR), and CSPs with a Data Storage Library (DL).

The capability changes for a CSP combined with a KR service provider are described in the CDSA KR specification and will not be covered here. The most common service provider type is CSP with a DL interface. This combination is covered here in detail.

C.8.1 Purpose of CSP Multi-Service Modules

The purpose of CSP multi-service modules with a DL interface (CSP/DL) is to give the CSP the ability to manage multiple key storage repositories, and provide more robust key management APIs. CSP/DLs are not intended to provide a fully capable DL module and an independent CSP in the same binary. As a result, CSP/DLs will usually provide a limited DL interface as required to implement key management. The DL interfaces in CSP/DL service providers are not required to support all database management functions. Specific functions that are not required include database creation, deletion, and schema modification.

C.8.2 Identifying Multi-Service Modules

CSP/DLs are identified using the MDS. The client must search the CDSA Common relation in the MDS for modules that have a *ServiceMask* field that has the CSSM_SERVICE_CSP and CSSM_SERVICE_DL flags set.

C.8.3 Assigning Subservice Identifiers

Subservice identifiers in CSP/DL service providers are assigned based on "cooperating service". Cooperating services are those that provide different interfaces to the same logical entity. For hardware CSPs, the logical entity may be a smartcard, PCMCIA card, or other general-purpose cryptographic device. For software only CSPs, the meaning of "logical entity" is left up to the service provider designer. The rule for cooperating services is that if a service provider supports a CSP and a DL subservice with the same subservice identifier, then they are cooperative and reference the same logical entity. If the service provider supports two different service provider types that do not cooperate, the subservice identifiers used by each type must not overlap.

C.8.4 Client Authentication and Authorization

The authentication and authorization for a CSP/DL is combined into a shared state for both service provider interfaces to a subservice. As a result, if the client has called the *CSP_Login*() SPI (*CSSM_CSP_Login*() API), it achieves the same effect as calling the *DL_Authenticate*() SPI (*CSSM_DL_Authenticate*() API) and vice-versa.

C.8.5 Managing Multiple Key Storage Databases

A client that uses a CSP/DL service provider has the ability to specify a specific database (DB) handle to a CSP interface that is to be used for storage of the new key. The client may only specify DB handles that were issued by the CSP's own DL interface. This allows the client to have control over where keys are stored, but allows the CSP to insure that the data is stored in a secure manner.

To specify a specific DB for key storage, the client must place the CSSM_DL_DB_HANDLE structure into the cryptographic context (CC) that will create the new key as a

CSSM_ATTRIBUTE_DL_DB_HANDLE attribute using the *CSSM_UpdateContextAttributes*( ) API. A CSP/DL service provider must always check for the CSSM_ATTRIBUTE_DL_DB_HANDLE attribute before storing a key. If it is not present in the CC, the CSP must use a default storage DB. The CSP/DL must always make sure that the DB handle is valid, and that it was issued by its own DL interface. CSP-only service providers do not check for the CSSM_ATTRIBUTE_DL_DB_HANDLE attribute, and will never return an error based on its value if it is present in the CC.

A CSP/DL must always have a default DB that is used for key storage in case the client does not specify a specific DB in the CC. The default DB is the first one in the list returned by the *DL_GetDbNames*( ) SPI (*CSSM_DL_GetDbNames*( ) API).

C.9 Algorithm Reference

C.9.1 Conventions

Table C-28 contains a list of algorithm actions, and the associated abbreviation used in this Appendix.

Abbreviation	Algorithm Operation	CSSM API(s)
H	Message digest (hashing)	CSSM_DigestData
M	Message Authentication Code (MAC)	CSSM_GenerateMac CSSM_VerifyMac
E	Symmetric key encryption	CSSM_EncryptData CSSM_DecryptData
P	Public key encryption/ private key decryption	CSSM_EncryptData(Public key) CSSM_DecryptData(Private key)
R	Signature with message recovery (private key encryption / public key decryption)	CSSM_EncryptData(Private key) or CSSM_SignData(Private key), CSSM_DecryptData(Public key)
S	Signature w/Appendix	CSSM_SignData(Private key) CSSM_VerifyData(Public key)
X	Key exchange	CSSM_WrapKey(Public key) or CSSM_WrapKey(Symmetric key), CSSM_UnwrapKey(Private key) or CSSM_UnwrapKey(Symmetric key)
K	Key generation	CSSM_GenerateKey (symmetric) CSSM_GenerateKeyPair (asymmetric)
D	Key derivation	CSSM_DeriveKey

Table C-28 Abbreviations for Algorithm Uses

C.9.2 Basic Algorithm Usage

Algorithms are specified to CSPs using a Cryptographic Context (CC). CCs contain information such as the algorithm class, algorithm ID, and a collection of attributes required by the algorithm or in addition to the required attributes to modify the default behaviors of the algorithm. In some cases, not all of the possible attributes for an operation can be specified using one of the CC creation APIs. In these cases, the additional attributes are added to the context using the *CSSM_UpdateContextAttributes*( ) API.

C.9.2.1 Digital Signatures

Digital signatures are typically generated using a two step process: computing the message digest of a piece of data, and performing a private key operation to obtain the final result. In the most common situation, both steps are performed with a single signature operation. In some applications, such as PKCS #7 packages with multiple signers, it is more efficient to perform both steps separately. In this Appendix, signature algorithms that perform both steps are called **combination signatures**, and algorithms that perform only the private key operation are called **encrypt-only** signatures.

Some signature algorithms are capable of being used in a variety of modes. The mode determines how the message digest is tagged and how the digest is transformed before performing the private key operation. Transformation of the message digest includes padding, masking, etc. Each algorithm that supports multiple modes will have a default mode that is assumed by the CSP if one is not explicitly specified as a CC attribute of type CSSM_ATTRIBUTE_MODE.

Combination Signatures

Combination signature algorithms have algorithm ID names of the form:

```
CSSM_ALGID_<DigestAlg>With<PrivateKeyAlg>
```

DigestAlg identifies the name of the message digest algorithm, such as SHA-1 or MD5. *PrivateKeyAlg* identifies the algorithm used to perform the private key operation. For instance CSSM_ALGID_MD5WithRSA and CSSM_ALGID_SHA1WithDSA indicate combination signature algorithms.

For a combination algorithm, the CSP performs all necessary message digest transformations according to the CSSM_ATTRIBUTE_MODE attribute in the signature CC. In addition, the *DigestAlgorithm* parameter to either *CSSM_SignData*( ) or *CSSM_VerifyData*( ) must be set to CSSM_ALGID_NONE.

Encrypt-only Signatures

Encrypt-only signatures are generated by transforming a message digest value and then performing the private key operation in separate steps. The signature algorithm does not compute the message digest value.

The CSSM_ATTRIBUTE_MODE attribute and the DigestAlgorithm parameter (to either *CSSM_SignData*( ) or *CSSM_VerifyData*( )) govern the extent of the message digest transformations performed by the CSP. The mode determines how the input digest is transformed. The digest algorithm indicates whether the CSP has to perform the entire transformation or should assume that the client has done the work of tagging the message digest. Tagging a message digest in most signature modes means that an indicator of the digest algorithm is appended or prepended to the digest value. If the digest algorithm is specified to be CSSM_ALGID_NONE, then the CSP assumes that the client has already tagged the digest value. If the digest tagging is performed by the CSP, then it must always check that the digest length is

appropriate for the specified digest algorithm.

Encrypt-only signatures can only be performed using single-stage APIs (*CSSM_SignData*() and *CSSM_VerifyData*()).

C.9.3 Algorithm Parameters

Many algorithms require a set of parameters to operate. For instance, the PKCS #3 Diffie-Hellman algorithm requires both parties to use a common prime (P) and base (G) value pair. In situations like this, the parameters are represented using a data block held in the CSSM_ATTRIBUTE_ALG_PARAMS attribute of the cryptographic context (CC).

When a client needs to generate the algorithm parameters, it uses the *CSP_GenerateAlgorithmParams*() SPI (*CSSM_GenerateAlgorithmParams*() API) to make the CSP generate a suitable set of parameters and place them in the CC. The CSP must generate the appropriate parameter values based on the context type, algorithm, etc. and add or update the CSSM_ATTRIBUTE_ALG_PARAMS attribute by returning an array of CSSM_CONTEXT_ATTRIBUTE structures to the CSSM. The CSP must allocate the memory for the array of attributes to be updated. The structures must be allocated using the application memory functions so that they can be deallocated by the CSSM. See <REFERENCE UNDEFINED>(typemalloc) for more information about application memory functions.

The formats for the algorithm parameters, if any, will be given along with the algorithm description in Section C.9.4, below.

C.9.4 Algorithm List

C.9.4.1 RSA

The RSA algorithm, CSSM_ALGID_RSA, is an asymmetric algorithm used for single-stage encryption and decryption, encrypt-only digital signature generation and verification (with and without message recovery), and key wrapping. RSA can be used in a number of modes, as listed in Table C-29 below.

Context Type CSSM_ALGCLASS_*	Algorithm Mode CSSM_ALGMODE_*	Valid Operations	Applicable Standard
ASYMMETRIC	NONE	P/R	PKCS #1 v2.0, RSAEP/RSADP
ASYMMETRIC	PKCS1_EME_V15	P/R/X	PKCS #1 v2.0, EME-PKCS1-v1_5
ASYMMETRIC	PKCS1_EME_OAEP	P/X	PKCS #1 v2.0, EME-OAEP
SIGNATURE	PKCS1_EMSA_V15	S/R	PKCS #1 v2.0, EMSA-PKCS1-v1_5
SIGNATURE	X9_31	S	ASNI X9.31-1998
ASYMMETRIC	ISO_9796	P/R	ISO 9796-1
SIGNATURE	ISO_9796	S/R	ISO 9796-1

Table C-29 Applicable Modes for CSSM_ALGID_RSA Context Type

The default algorithm mode if the CSSM_ATTRIBUTE_MODE attribute is not in the CC is CSSM_ALGMODE_PKCS1_EME_V15 for asymmetric contexts and CSSM_ALGMODE_PKCS1_EMSA_V15 for signature contexts.

Refer to the appropriate standards for the input length restrictions and output sizes for each mode.

When generating RSA keypairs, the length of the public-modulus may be specified in 8-bit increments. The resulting public and private keys have CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_RSA.

Signature generation and key unwrapping require a private key with CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_RSA. Signature verification and key wrapping require a public key of the same type. Encryption and decryption accept either a public or private key.

Algorithm Parameters

RSA key generation has an optional algorithm parameter, the CSSM_ATTRIBUTE_ALG_PARAM attribute, specified using the *Param* parameter of the *CSSM_CSP_CreateKeyGenContext*( ) API. The value of the parameter is the public exponent for the new key pair. The exponent must be represented as an integer with the most significant byte first. The *CSSM_GenerateAlgorithmParams*( ) API can be used to make the CSP pre-generate the public exponent for the key generation.

When using the PKCS #1 OAEP algorithm mode, CSSM_ALGMODE_PKCS1_EME_OAEP, the client must specify the algorithm parameters as an attribute of type CSSM_ATTRIBUTE_ALG_PARAMS. The CSSM_DATA::Data value must point to the start address of a CSSM_PKCS1_OAEP_PARAMS structure, and the CSSM_DATA::Length value must be the size of the parameter structure. The *CSSM_GenerateAlgorithmParams*( ) API can not be used to generate these parameters.

C.9.4.2 Combination Signatures with RSA

The RSA combination signature algorithms are used for single-stage and multi-staged digital signature generation and verification (without message recovery). The RSA combination signature algorithms can be used in a number of modes, as listed in Table C-30 below.

Context Type CSSM_ALGCLASS_*	**Algorithm Mode CSSM_ALGMODE_***	**Valid Operations**	**Applicable Standard**
SIGNATURE	PKCS1_EMSA_V15	S	PKCS #1 v2.0, EMSA-PKCS1-v1_5
SIGNATURE	X9_31	S	ANSI X9.31-1998
SIGNATURE	ISO_9796	S	ISO 9796-1

Table C-30 Applicable Modes for Combination Signatures with RSA

The default algorithm mode if the CSSM_ATTRIBUTE_MODE attribute is not in the CC is CSSM_ALGMODE_PKCS1_EMSA_V15.

Refer to the appropriate standards for the input length restrictions and output sizes for each mode.

Signature generation requires a private key with CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_RSA. Signature verification requires a public key of the same type.

C.9.4.3 DSA

The DSA algorithm, CSSM_ALGID_DSA, is an asymmetric algorithm used for single-stage encrypt-only digital signature generation and verification. RSA can be used in the modes listed in Table C-31.

Context Type CSSM_ALGCLASS_*	**Algorithm Mode CSSM_ALGMODE_***	**Valid Operations**	**Applicable Standard**
SIGNATURE	NONE	S	NIST FIPS 186-2; ANSI X9.30:1-1993

Table C-31 Applicable Modes for CSSM_ALGID_DSA

When generating DSA keypairs, the length of the prime may be specified in 64-bit increments, starting at 512. The resulting public and private keys have CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_DSA.

Signature generation requires a private key with CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_DSA. Signature verification requires a public key of the same type. The *DigestAlgorithm* parameter to the *CSSM_SignData*( ) and *CSSM_VerifyData*( ) APIs must be CSSM_ALGID_SHA1, and the length of the input must total exactly 160 bits.

Algorithm Parameters

DSA key generation has an optional algorithm parameter, the CSSM_ATTRIBUTE_ALG_PARAM attribute, specified using the *Param* parameter of the *CSSM_CSP_CreateKeyGenContext*( ) API. The parameter is a BER encoded **Dss-params** structure, described above, containing the prime, sub-prime, and base (p, q, and g) values. If the CSSM_ATTRIBUTE_ALG_PARAMS attribute is not present in the context, the parameters will be generated along with the public and private values. The *CSSM_GenerateAlgorithmParams*( ) API can be used to make the CSP pre-generate the key parameters for subsequent key pair generation.

Generating DSA parameters requires a 160-bit seed value. The CSP obtains this value using the CSSM_ATTRIBUTE_SEED attribute in the cryptographic context. If the supplied seed value is not 160 bits long, then the seed value is processed using SHA-1 and the resulting 160-bit value is used. If the CSSM_ATTRIBUTE_SEED attribute is not present in the context, then an internally generated 160-bit random value is used.

C.9.4.4 Combination Signatures with DSA

The DSA combination signature algorithms are used for single-stage and multi-staged digital signature generation and verification. The DSA combination signature algorithms do not have a choice of operating modes. They are listed in Table C-32.

Context Type CSSM_ALGCLASS_*	**Algorithm Mode CSSM_ALGMODE_***	**Valid Operations**	**Applicable Standard**
SIGNATURE	NONE	S	NIST FIPS 186-2; ANSI X9.30:1-1993

Table C-32 Applicable Modes for combination Signatures with DSA

Signature generation requires a private key with CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_DSA. Signature verification requires a public key of the same type.

C.9.4.5 Diffie-Hellman (PKCS #3)

The PKCS #3 Diffie-Hellman algorithm, CSSM_ALGID_DH, is an algorithm used for asymmetric key pair generation and symmetric key derivation. The combination of the two functions completes a key agreement between two parties. The resulting symmetric key can range in length from 8 bits to the length of the prime value, p, in increments of 8 bits.

Table C-33 lists the context types and applicable standards for PKCS #3 Diffie-Hellman key agreement.

Context Type CSSM_ALGCLASS_*	**Algorithm Mode CSSM_ALGMODE_***	**Valid Operations**	**Applicable Standard**
KEYGEN	NONE	K	PKCS #3 v1.5
DERIVEKEY	NONE	D	PKCS #3 v1.5

Table C-33 Context Types and Modes for PKCS #3 Diffie-Hellman

When generating Diffie-Hellman keypairs, the length of the prime value may be specified in 8-bit increments. The resulting public and private keys have CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_DH. The public key value, CSSM_KEY::KeyData, must be sent to the other party taking part in the key agreement process.

When deriving the symmetric key, the public value received from the other party is specified using the *Params* parameter to the *CSSM_DeriveKey*( ) API. Key derivation requires a private key with CSSM_KEYHEADER::AlgorithmId equal to CSSM_ALGID_DH.

Algorithm Parameters

PKCS #3 Diffie-Hellman key pair generation has an optional algorithm parameter, the CSSM_ATTRIBUTE_ALG_PARAM attribute, specified using the *Param* parameter of the *CSSM_CSP_CreateKeyGenContext*( ) API. The parameter is a BER encoded DHParameter structure, described in PKCS #3, containing the prime and base (p and g) values. If the CSSM_ATTRIBUTE_ALG_PARAMS attribute is not present in the context, the parameters will be generated along with the public and private values.

The *CSSM_GenerateAlgorithmParams*( ) API can be used to have the CSP pre-generate the key parameters for subsequent key pair generation. These parameters must also be sent to the other party taking place in the key agreement process. They can be fetched from the context using the *CSSM_GetContext*( ) and *CSSM_GetContextAttribute*( ) APIs.

C.9.4.6 Password Based Key Derivation (PKCS #5)

PKCS #5 Password Based Key Derivation (PBD) is a collection of algorithms for deriving a symmetric encryption key, and sometimes an initialization vector from a password and random salt value. PKCS #5 v2.0 defines two methods for deriving key material from a password: PBKDF1 and PBKDF2. PBKDF1 is compatible with the PKCS #5 v1.5 key derivation algorithm, and generates a key with a maximum length of 64-bits, and an 8-byte initialization vector (IV). PBKDF2 allows derivation of keys with an arbitrary length, but does not generate an IV.

,cX table34 "" 1 lists the PBD functions defined in PKCS #5 v2.0.

Context Type CSSM_ALGCLASS_*	Algorithm CSSM_ALGID_*	PBD Function	Parameter Structure CSSM_PKCS5_*
DERIVEKEY	PKCS5_PBKDF1_MD2	PBKDF1	PBKDF1_PARAMS
DERIVEKEY	PKCS5_PBKDF1_MD5	PBKDF1	PBKDF1_PARAMS
DERIVEKEY	PKCS5_PBKDF1_SHA1	PBKDF1	PBKDF1_PARAMS
DERIVEKEY	PKCS5_PBKDF2	PBKDF2	PBKDF2_PARAMS

Table C-34 Algorithm IDs and Parameter Structures for PKCS-5 PBD

All of the PKCS #5 PBD algorithms require salt and iteration count values. They are obtained from the CSSM_ATTRIBUTE_SALT and CSSM_ATTRIBUTE_ITERATION_COUNT attributes and specified by the application using the *Salt* and *Iterations* parameters to the *CSSM_CreateDeriveKeyContext*( ) API respectively.

For PBKDF1 based algorithms, the *salt* value must be exactly 8 bytes long. For PBKDF2, the *salt* value may have any non-zero length. The CSP returns CSSMERR_CSP_MISSING_ATTR_SALT if the *salt* attribute is not present in the context, and CSSMERR_CSP_INVALID_ATTR_SALT if the value does not meet the requirements of the algorithm.

The CSP returns CSSMERR_CSP_MISSING_ATTR_ITERATION_COUNT if the iteration count attribute is not present in the context, and CSSMERR_CSP_INVALID_ATTR_ITERATION_COUNT if the value is zero.

The type of the output key is determined by the CSSM_ATTRIBUTE_KEY_TYPE attribute in the context. The CSP returns CSSMERR_CSP_MISSING_ATTR_KEY_TYPE if the key type attribute is not present in the context, and CSSMERR_CSP_INVALID_ATTR_KEY_TYPE if the type is not supported by the CSP.

Data Structures

When using a PKCS #5 PBD algorithm, the caller must supply an appropriate parameter structure. The structure is supplied through the *Params* parameter to the *CSSM_DeriveKey*( ) API. *Params.Data* must point to the structure, and *Params.Length* must be equal to the size of the structure.

CSSM_PKCS5_PBKDF1_PARAMS

```
typedef struct CSSM_PKCS5_PBKDF1_PARAMS {
    CSSM_DATA Passphrase;       /* Input  */
    CSSM_DATA InitVector;       /* Output */
} CSSM_PKCS5_PBKDF1_PARAMS;
```

Definitions

Passphrase (input)
Buffer containing the passphrase used as the basis for the PBD operation. CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

InitVector (output)
The CSP will return an 8-byte IV in this buffer. If the caller does not supply a buffer, it is allocated as described above.

CSSM_PKCS5_PBKDF2_PARAMS

```
typedef uint32 CSSM_PKCS5_PBKDF2_PRF;
#define CSSM_PKCS5_PBKDF2_PRF_HMAC_SHA1 ( 0 )

typedef struct CSSM_PKCS5_PBKDF2_PARAMS {
    CSSM_DATA Passphrase;
    CSSM_PKCS5_PBKDF2_PRF PseudoRandomFunction;
} CSSM_PKCS5_PBKDF2_PARAMS;
```

Definitions

Passphrase (input)
Buffer containing the passphrase used as the basis for the PBD operation. CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

PseudoRandomFunction (input)
Pseudo random function (PRF) used by the PBKDF2 algorithm.

C.9.4.7 Generic Message Digests

Message digests implement a one-way compression function on a stream of data. The result is a message representative that is usually shorter that the input value. Message digests can be calculated using single-stage or multi-stage APIs.

Table C-35 lists the generic message digest algorithm identifiers defined by the current version of the CDSA standard.

Algorithm Name CSSM_ALGID_*	Valid Operations	Applicable Standard
MD2	H	IETF RFC1319, April 1992
MD4	H	IETF RFC1320, April 1992
MD5	H	IETF RFC1321, April 1992
SHA1	H	NIST FIPS 180-1; ASNI X9.30:2-1993
RIPEMD160	H	ISO/IEC 10118-3:1998

Table C-35 Generic Message Digest Algorithm Identifiers and Standards

Generic message digest algorithms do not have different modes of operation.

Refer to the appropriate standards for the input length restrictions and output sizes for each algorithm.

C.9.4.8 Generic Block Ciphers

Generic block ciphers are used for symmetric key generation, bulk encryption and decryption, computing Message Authentication Codes, and key wrapping and unwrapping. Block ciphers operate on fixed length block of data, are capable of being used in multiple feedback modes, and can be used with a number of padding modes in case the input data is not a multiple of the block length.

Commonly used block ciphers are listed in Table C-35.

Algorithm CSSM_ALGID_*	Valid Operations	Applicable Standards	Comments
DES, 3DES_3KEY_EEE, 3DES_3KEY_EDE, 3DES_2KEY_EEE, 3DES_2KEY_EDE, 3DES_1KEY_EEE	E, X, M	NIST FIPS 46-3; ANSI X9.52- 1998	The 3KEY identifiers indicate a triple-length DES key. The 2KEY identifiers indicate a double-length DES key. The EEE identifiers indicate an encrypt, encrypt, encrypt sequence is performed using consecutive DES keys within the single, double, or triple length keys. Double length operations use the first key for the first and third operation. Single length operations use the same key for all operations. EDE identifiers indicate an encrypt, decrypt, encrypt sequence using the keys as described for EEE operations.
RC2	E, X, M	IETF RFC2268, January 1998	
RC5	E, X, M	IETF RFC2040, October 1996	

Generic block ciphers can be used in a number of different modes, with multiple padding methods. The most common methods are described in Table C-36. The additional required cryptographic context attributes are also listed for each mode.

Algorithm Mode CSSM_ALGMODE_*	Applicable Standards	Additional CC Attributes CSSM_ATTRIBUTE_*	Description
ECB	NIST FIPS 46-3; NIST FIPS 81; ANSI X9.52-1998	None	Electronic Codebook (ECB) mode. Raw block encryption with no feedback between data blocks. Equivalent blocks will encrypt to the same value. The input data length must be a multiple of the cipher block size.
ECBPad		PADDING	ECB mode with data padding. The final block of the input data is padded until a multiple of the block length is obtained. Some padding modes may add an additional block to the output data.
CBC_IV8 CFB_IV8 OFB_IV8		INIT_VECTOR	Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB) modes. Block encryption with feedback between blocks. Equivalent blocks will encrypt to different values. The input data length must be a

			multiple of the cipher block size.
CBCPadIV8 CFBPadIV8 OFBPadIV8		INIT_VECTOR, PADDING	Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB) modes with data padding. The final block of the input data is padded until a multiple of the block length is obtained. Some padding modes may add an additional block to the output data.

Table C-36 Algorithm IDs and Standards for Block Ciphers

If the algorithm mode indicates *IV8*, then the application must supply an 8-byte initialization vector using the CSSM_ATTRIBUTE_INIT_VECTOR attribute, and specified by the application using the *InitVector* parameter to the *CSSM_CSP_CreateSymmetricContext*( ) API. The CSP returns CSSMERR_CSP_MISSING_ATTR_INIT_VECTOR if the attribute is not present in the context and CSSMERR_CSP_INVALID_ATTR_INIT_VECTOR if it does not meet length requirements.

If the algorithm mode indicates *Pad*, then the application must supply a padding method using the CSSM_ATTRIBUTE_PADDING attribute, and specified by the application using the *Padding* parameter to the *CSSM_CSP_CreateSymmetricContext*( ) API. The CSP returns CSSMERR_CSP_MISSING_ATTR_PADDING if the attribute is not present in the context, and CSSMERR_CSP_INVALID_ATTR_PADDING if the requested method is not supported by the CSP. If the cipher mode selected applies data padding, it can use the padding methods in <REFERENCE UNDEFINED>(table38). Note that some methods are not always desirable due to the inability to accurately remove the padding during decryption.

Padding Method CSSM_PADDING_*	**Applicable Standard**	**Description**
ZERO		The final block is padded with zeros until the proper length is reached. The method cannot be accurately stripped during decryption if the data contains trailing zero bytes.
ONE		The final block is padded with one bits until the proper length is reached. The method cannot be accurately stripped during decryption if the data contains trailing bytes containing all one bits.
PKCS5	PKCS #5 v2.0	Popular padding method that can be accurately stripped in all cases. Limited to an 8-byte block length. If the input data length is a multiple of the block length, an extra block of data will be added to the output.
PKCS7	PKCS #7 v1.5	Generalized version of PKCS #5 padding that can be applied to any block length.
RANDOM		Random bytes are used to extend the input data to the proper length. This padding cannot be stripped by the CSP.

Table C-37 Padding Modes for Block Ciphers

Additional RC2 Requirements

RC2 allows a variable number of bits in the key schedule to be used for encryption and decryption. The number of effective bits can be configured using the CSSM_ATTRIBUTE_EFFECTIVE_BITS attribute in the cryptographic context. The value of the attribute can range from 1 to 1024. If the attribute is missing from the context, the default value 1024 is used. If the value of the attribute is outside the value range, the error CSSMERR_CSP_INVALID_ATTR_EFFECTIVE_BITS is returned. The CSSM_ATTRIBUTE_EFFECTIVE_BITS attribute must be added to the context using the CSSM_UpdateContextAttributes API.

Additional RC5 Requirements

RC5 allows a variable block size and number of rounds.

The block size can be configured using the CSSM_ATTRIBUTE_BLOCK_SIZE attribute in the cryptographic context. The value of the attribute can be 32, 64, or 128. If the attribute is missing from the context, the default value 64 is used. If the value of the attribute is outside the value range, the error CSSMERR_CSP_INVALID_ATTR_BLOCK_SIZE is returned.

The number of rounds can be configured using the CSSM_ATTRIBUTE_ROUNDS attribute in the cryptographic context. If the attribute is missing from the context, the default value 16 is used. The CSSM_ATTRIBUTE_BLOCK_SIZE and CSSM_ATTRIBUTE_ROUNDS attributes must be added to the context using the *CSSM_UpdateContextAttributes*( ) API.

C.9.4.9 Generic Stream Ciphers

Generic stream ciphers are used for symmetric key generation, bulk encryption and decryption, and key wrapping and unwrapping. Stream ciphers operate on data one bit or byte at a time, and have no block length or mode restrictions.

Commonly used stream ciphers are listed in Table C-38.

Algorithm CSSM_ALGID_*	**Valid Operations**	**Applicable Standard**	**Comments**
RC4	E, X		

Table C-38 Algorithm IDs and Standards for Stream Ciphers

C.9.5 SSL 3.0 Algorithms

The CDSA Technical Standard defines a set of algorithm identifiers that represent steps in the SSL 3.0 handshake protocol. The protocol steps include pre-master key generation, master key derivation, cipher and MAC key derivation, and the SSL defined MD5-MAC and SHA-1-MAC variants.

Table C-39 lists the SSL 3.0 handshake protocol step identifiers, the context types, and the parameter structures used with the *CSSM_DeriveKey*( ) API. A detailed description of each identifier follows after Table C-39. When a parameter structure is required, the *Params* parameter to the *CSSM_DeriveKey*( ) API has the *Data* field pointing to the parameter structure, and the *Length* field is the size of the parameter structure.

Context Type CSSM_ALGCLASS_*	Algorithm ID CSSM_ALGID_*	Parameter Structure CSSM_SSL3_*
KEYGEN	SSL3PreMasterGen	None.
DERIVEKEY	SSL3MasterDerive	MASTER_KEY_DERIVE_PARAMS
DERIVEKEY	SSL3KeyAndMacDerive	KEY_AND_MAC_DERIVE_PARAMS
MAC	SSL3MD5_MAC	N/A
MAC"SSL3SHA1_MAC	N/A	

Table C-39 SSL 3.0 Algorithm IDs, Context Types and Parameter

The Data Structures are defined below.

C.9.5.1 Data Structures

CSSM_SSL3_MASTER_KEY_DERIVE_PARAMS

```
typedef struct CSSM_SSL3_MASTER_KEY_DERIVE_PARAMS {
    CSSM_DATA ClientRandom; /* Input  */
    CSSM_DATA ServerRandom; /* Input  */
    CSSM_VERSION Version;   /* Output */
} CSSM_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

Definitions

ClientRandom (input)
Buffer containing the client random data. The CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

ServerRandom
Buffer containing the server random data. The CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

Version (output)
SSL protocol version number extracted from the pre-master secret during the derivation process. Modified by the key derivation process.

CSSM_SSL3_KEY_AND_MAC_DERIVE_PARAMS

```
typedef struct CSSM_SSL3_KEY_AND_MAC_DERIVE_PARAMS {
    uint32 MACKeyLogicalSizeInBits;    /* Input */
    uint32 SecretKeyLogicalSizeInBits; /* Input */
    uint32 IVLengthInBits;             /* Input */
    CSSM_BOOL IsExport;                /* Input */
    CSSM_DATA ClientRandom;            /* Input */
    CSSM_DATA ServerRandom;            /* Input */
    CSSM_KEY ClientMACKey;             /* Output */
    CSSM_KEY ServerMACKey;             /* Output */
    CSSM_KEY ClientWriteKey;           /* Output */
    CSSM_KEY ServerWriteKey;           /* Output */
    CSSM_DATA ClientIV;                /* Output */
    CSSM_DATA ServerIV;                /* Output */
} CSSM_SSL3_KEY_AND_MAC_DERIVE_PARAMS;
```

Definitions

MACKeyLogicalSizeInBits (input)
The size of the MACing keys created during the derivation process.

SecretKeyLogicalSizeInBits (input)
The size of the encryption keys created during the derivation process.

IVLengthInBits (input)
If initialization vectors are required, then this must be set to the length of the IV in bits. If IVs are not required, then this field must be set to zero.

IsExport (input)
CSSM_TRUE if the keys are being derived for an export cipher, CSSM_FALSE otherwise.

ClientRandom (input)
Buffer containing the client random data. The CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

ServerRandom (input)
Buffer containing the server random data. The CSP returns CSSMERR_CSP_INVALID_INPUT_POINTER if the data pointer is NULL, and CSSMERR_CSP_INVALID_DATA if the length is zero.

ClientMACKey (output)
MACing key of type CSSM_ALGID_GenericSecret used for data written by the client.

ServerMACKey (output)
MACing key of type CSSM_ALGID_GenericSecret used for data written by the server.

ClientWriteKey (output)
Encryption key used for data written by the client.

ServerWriteKey (output)
Encryption key used for data written by the server.

ClientIV (output)
Initialization vector used for data written by the client.

ServerIV (output)
Initialization vector used for data written by the server.

C.9.5.2 *Pre-Master Key Generation*

SSL 3.0 Pre-Master Key Generation, *CSSM_ALGID_SSL3PreMasterGen*, is used to generate a 48-byte *pre-master* key. The client wraps the pre-master key and sends it to the server when the chosen key exchange mechanism is RSA or Fortezza.

The result of the key generation is a symmetric key of type *CSSM_ALGID_GenericSecret*. The length of the key is always 48 bytes (384 bits). The CSP returns CSSMERR_CSP_INVALID_ATTR_KEY_TYPE or CSSMERR_CSP_INVALID_ATTR_KEY_LENGTH if the requested key algorithm, or the requested length does not match those values respectively.

The pre-master secret includes a protocol version number to prevent version rollback attacks. The client can specify the protocol version by adding a CSSM_ATTRIBUTE_VERSION attribute to the key generation context. If the protocol version is not found in the context, then it defaults to 3.0.

Refer to the SSL 3.0 specification for a detailed description of pre-master key generation.

C.9.5.3 *Master Key Derivation*

SSL 3.0 Master Key Derivation, *CSSM_ALGID_SSL3MasterDerive*, is used to derive a 48-byte master secret from a pre-master secret key and random data supplied by both the client and the server. This step in the handshake protocol is performed regardless of the key exchange algorithm.

The result of the key derivation is a symmetric key of type *CSSM_ALGID_GenericSecret*. The length of the key is always 48 bytes (384 bits). The CSP returns CSSMERR_CSP_INVALID_ATTR_KEY_TYPE or CSSMERR_CSP_INVALID_ATTR_KEY_LENGTH if the requested key algorithm, or the requested length does not match those values respectively. The version number embedded in the pre-master secret is also returned in the CSSM_SSL_MASTER_KEY_DERIVE_PARAMS::Version field of the parameters structure.

The following rules are applied to the CSSM_KEYATTR_ALWAYS_SENSITIVE and CSSM_KEYATTR_NEVER_EXTRACTABLE attributes based on the attributes of the pre-master secret key and the attributes specified for the resulting key:

- If the pre-master key does not have its CSSM_KEYATTR_ALWAYS_SENSITIVE flag set, then it will not be set in the derived key. If the pre-master key has its CSSM_KEYATTR_ALWAYS_SENSITIVE flag set, then the derived key has its CSSM_KEYATTR_ALWAYS_SENSITIVE flag set to the same value as its CSSM_KEYATTR_SENSITIVE flag.
- If the pre-master key does not have its CSSM_KEYATTR_NEVER_EXTRACTABLE flag set, then it will not be set in the derived key. If the pre-master key has its CSSM_KEYATTR_NEVER_EXTRACTABLE flag is set, then the derived key has its CSSM_KEYATTR_NEVER_EXTRACTABLE flag set to the opposite value from its CSSM_KEYATTR_EXTRACTABLE flag.

C.9.5.4 *Encryption and MACing Secret Key Derivation*

SSL 3.0 encryption and MACing key derivation, *CSSM_ALGID_SSL3KeyAndMacDerive*, is used to derive client and server encryption and integrity keys from the master secret key. This is the final step in the SSL 3.0 handshake protocol. The algorithm returns two encryption keys, two integrity keys, and two initialization vectors.

The two MACing keys are always type *CSSM_ALGID_GenericSecret*. The CSSM_KEYHEADER::KeyUse fields have the CSSM_KEYUSE_SIGN (generate MAC), CSSM_KEYUSE_VERIFY (verify MAC), and CSSM_KEYUSE_DERIVE flags set.

The two encryption keys are determined by the value of the CSSM_ATTRIBUTE_KEY_TYPE attribute in the context. The CSP will return CSSMERR_CSP_MISSING_ATTR_KEY_TYPE if the key type attribute is not present in the context, and CSSMERR_CSP_INVALID_ATTR_KEY_TYPE if the type is not supported by the CSP. The CSSM_KEYHEADER::KeyUse fields have the CSSM_KEYUSE_ENCRYPT, CSSM_KEYUSE_DECRYPT, and CSSM_KEYUSE_DERIVE flags set.

All four keys always inherit the values of the CSSM_KEYATTR_SENSITIVE, CSSM_KEYATTR_ALWAYS_SENSITIVE, CSSM_KEYATTR_EXTRACTABLE, and CSSM_KEYATTR_NEVER_EXTRACTABLE flags from the base key.

When using this algorithm, the *DerivedKey* parameter is not used and must be NULL.

C.9.5.5 *MD5 and SHA-1 MACing*

SSL 3.0 MD5 and SHA-1 MACing, CSSM_ALGID_SSL3MD5_MAC and CSSM_ALGID_SSL3SHA1_MAC, algorithms are used to generate message authentication codes according to the SSL 3.0 specification.

Algorithm Name CSSM_ALGID_*	**Valid Operations**	**Applicable Standard**
SSL3MD5_MAC	M	Netscape SSL 3.0
SSL3SHA1_MAC	M	Netscape SSL 3.0

Table C-40 MD5 and SHA-1 MAC Algorithms for MACs

The length of the MAC output is variable, from 4 to 8 bytes. The length is determined by specifying the CSSM_ATTRIBUTE_OUTPUT_SIZE parameter in the context. If the attribute is not present, then the output size defaults to 8. If the attribute is presentebut outside the valid range, the CSP returns CSSMERR_CSP_INVALID_ATTR_OUTPUT_SIZE. The CSSM_ATTRIBUTE_OUTPUT_SIZE attribute must be added to the context using the *CSSM_UpdateContextAttributes*( ) API.

Both MAC algorithms require a symmetric key with type *CSSM_ALGID_GenericSecret*.

Appendix D

Signed Manifests

D.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.

D.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.

D.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 and a new Integrity Core set to describe the integrity of the referents.

D.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).

D.3.2 Dublin Core

Details of the specification of the Dublin Core set are outside the scope of this document.

D.3.3 PKWARE Archive File Format Specification

Details of the PKWARE archive format are outside the scope of this document.

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.

carve-outs
The term in general use in the United States of America to identify a set of constants corresponding to the application areas currently recognized by the United States Department of Commerce as application areas that can be granted an export license to use strong cryptography. Financial applications have been recognized for carve-out for several years. The application areas of medicine and insurance are recent additions to the carve-out list.

CDSA
See Common Data Security Architecture

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
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
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*.

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

CSSM
See Common Security Services Manager.

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.

USEE
USEE A tag defining a set of Use Exemptions (USEE). Applications present one USEE tag value when requesting privileged services. CSSM and add-in service provider modules have a set of associated USEE tags. Each tag defines one or more use exemptions that can be granted to authorized callers. Each USEE tag represents policy-based exemptions for the use of Cryptographic Services, Key Recovery Services, and other CSSM services available only to authorized callers.

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

AC_AuthCompute **417**
AC_PassThrough **422**
Asymmetric algorithms 985
carve-outs 985
CDSA 985
Certificate 985
Certificate chain 985
Certificate signing 985
Certificate validity date 985
Certification Authority (CA) 985
CL_CertAbortCache **465**
CL_CertAbortQuery **452**
CL_CertCache **459**
CL_CertCreateTemplate **439**
CL_CertDescribeFormat **470**
CL_CertGetAllFields **455**
CL_CertGetAllTemplateFields **441**
CL_CertGetFirstCachedFieldValue **461**
CL_CertGetFirstFieldValue **448**
CL_CertGetKeyInfo **454**
CL_CertGetNextCachedFieldValue **463**
CL_CertGetNextFieldValue **450**
CL_CertGroupFromVerifiedBundle **468**
CL_CertGroupToSignedBundle **466**
CL_CertSign **442**
CL_CertVerify **444**
CL_CertVerifyWithKey **446**
CL_CrlAbortCache **505**
CL_CrlAbortQuery **491**
CL_CrlAddCert **476**
CL_CrlCache **494**
CL_CrlCreateTemplate **472**
CL_CrlDescribeFormat **506**
CL_CrlGetAllCachedRecordFields **503**
CL_CrlGetAllFields **492**
CL_CrlGetFirstCachedFieldValue **498**
CL_CrlGetFirstFieldValue **487**
CL_CrlGetNextCachedFieldValue **501**
CL_CrlGetNextFieldValue **489**
CL_CrlRemoveCert **478**
CL_CrlSetFields **474**
CL_CrlSign **480**
CL_CrlVerify **482**
CL_CrlVerifyWithKey **484**
CL_FreeFields **457**
CL_FreeFieldValue **458**
CL_IsCertInCachedCrl **496**
CL_IsCertInCrl **486**
CL_PassThrough **508**
Common Data Security Architecture 985
Common Security Services Manager 985
Cryptographic algorithm 986
Cryptographic Service Providers (CSPs) 986
Cryptoki 986
CSP_EventNotify **300**
CSSM 986
cssm_CcToHandle **920**
CSSM_ChangeKeyAcl **198**
CSSM_ChangeKeyOwner **201**
CSSM_CSP_ChangeLoginAcl **194**
CSSM_CSP_ChangeLoginOwner **203**
CSSM_CSP_CreateAsymmetricContext **175**
CSSM_CSP_CreateDeriveKeyContext **177**
CSSM_CSP_CreateDigestContext **172**
CSSM_CSP_CreateKeyGenContext **179**
CSSM_CSP_CreateMacContext **173**
CSSM_CSP_CreatePassThroughContext **181**
CSSM_CSP_CreateRandomGenContext **174**
CSSM_CSP_CreateSignatureContext **168**
CSSM_CSP_CreateSymmetricContext **170**
CSSM_CSP_GetLoginAcl **192**
CSSM_CSP_GetLoginOwner **202**
CSSM_CSP_Login **190**
CSSM_CSP_Logout **191**
CSSM_DeleteContext **185**
CSSM_DeleteContextAttributes **188**
cssm_DeregisterManagerServices **882**
CSSM_FreeContext **183**
CSSM_GetAPIMemoryFunctions **119**
cssm_GetAppMemoryFunctions **879**
cssm_GetAttachFunctions **877**
CSSM_GetContext **182**
CSSM_GetContextAttribute **186**
CSSM_GetKeyAcl **196**
CSSM_GetKeyOwner **200**
CSSM_GetModuleGUIDFromHandle **114**
cssm_GetModuleInfo **921**
CSSM_GetPrivilege **113**
CSSM_GetSubserviceUIDFromHandle **115**
CSSM_Init **98**
CSSM_Introduce **106**
cssm_IsFuncCallValid **880**

CSSM_KR_CreateRecoveryEnablementContext 670
CSSM_KR_CreateRecoveryRegistrationContext 669
CSSM_KR_CreateRecoveryRequestContext....671
CSSM_KR_FreePolicyInfo 695
CSSM_KR_GetPolicyInfo 672
CSSM_KR_QueryPolicyInfo 693
CSSM_KR_SetEnterpriseRecoveryPolicy....667
CSSM_ListAttachedModuleManagers 117
CSSM_ModuleAttach....108
CSSM_ModuleDetach 110
CSSM_ModuleLoad....104
CSSM_ModuleUnload....105
cssm_ReleaseAttachFunctions 878
CSSM_SetContext 184
CSSM_SetPrivilege....111
CSSM_SPI_ModuleAttach....916
CSSM_SPI_ModuleDetach 918
CSSM_SPI_ModuleLoad....914
CSSM_SPI_ModuleUnload 915
CSSM_Terminate 102
CSSM_TP_RetrieveCredResult 348
CSSM_Unintroduce 107
CSSM_UpdateContextAttributes....187
DecryptData 252
DecryptDataFinal 261
DecryptDataInit 256
DecryptDataInitP....258
DecryptDataP 255
DecryptDataUpdate....259
DeregisterDispatchTable....873
DeriveKey 283
DigestData 217
DigestDataClone....222
DigestDataFinal 224
DigestDataInit 219
DigestDataUpdate....220
Digital certificate....986
Digital signature 986
DL_Authenticate 545
DL_ChangeDbAcl 549
DL_ChangeDbOwner....552
DL_CreateRelation....562
DL_DataAbortQuery 580
DL_DataDelete....571
DL_DataGetFirst....575
DL_DataGetFromUniqueRecordId....581
DL_DataGetNext....578
DL_DataInsert 569
DL_DataModify 572
DL_DbClose....556
DL_DbCreate....557
DL_DbDelete 560
DL_DbOpen....554
DL_DestroyRelation 564
DL_FreeNameList 567
DL_FreeUniqueRecord....583
DL_GetDbAcl....547
DL_GetDbNameFromHandle 566
DL_GetDbNames 565
DL_GetDbOwner 551
DL_PassThrough 585
EISL_CheckAddressWithinModule 786
EISL_CheckDataAddressWithinModule....787
EISL_ContinueVerification 736
EISL_CopyCertificateChain 766
EISL_CreateCertAttributeEnumerator 770
EISL_CreateCertificateChain 764
EISL_CreateCertChainWithCertificate 765
EISL_CreateCertChainWCredDataAndCert 763
EISL_CreateCertChainWithCredentialData....762
EISL_CreateManifestAttributeEnumerator....751
EISL_CreateManifestSectionAttributeEnum....778
EISL_CreateManifestSectionEnumerator....747
EISL_CreateSignatureAttributeEnumerator....757
EISL_CreateSignerInfoAttributeEnumerator...753
EISL_CreateVerifiedSignatureRoot....744
EISL_CreateVerifiedSignatureRootWithCert ...745
EISL_CreateVerifiedSigRootWithCredData 741
EISL_CreateVerSigRootWCredDataAndCert...743
EISL_DuplicateVerifiedModulePtr 738
EISL_FindCertificateAttribute 769
EISL_FindManifestAttribute....750
EISL_FindManifestSection 746
EISL_FindManifestSectionAttribute 777
EISL_FindSignatureAttribute 756
EISL_FindSignerInfoAttribute....752
EISL_GetCertficateChain....735
EISL_GetLibHandle 788
EISL_GetManifestSignatureRoot 774
EISL_GetModuleManifestSection....781
EISL_GetNextAttribute....754
EISL_GetNextCertificateAttribute....771
EISL_GetNextManifestSection 748
EISL_GetNextManifestSectionAttribute 779
EISL_GetNextSignatureAttribute 758
EISL_GetReturnAddress....785
EISL_LocateProcedureAddress 783
EISL_RecycleAttributeEnumerator 755
EISL_RecycleCertificateAttributeEnum 772
EISL_RecycleManifestSectionAttributeEnum..780
EISL_RecycleManifestSectionEnumerator....749
EISL_RecycleSignatureAttributeEnumerator...759

EISL_RecycleVerifiedCertificateChain.............**767**
EISL_RecycleVerifiedModuleCredentials **739**
EISL_RecycleVerifiedSignatureRoot..................**760**
EISL_SelfCheck **718**
EISL_VerifyAndLoadModule **775**
EISL_VerAndLoadModAndCredDataWCert...**721**
EISL_VerAndLoadModAndCredentialData.....**719**
EISL_VerAndLoadModAndCredentials............**723**
EISL_VerAndLoadModAndCredWithCert.......**725**
EISL_VerLoadedModule.....................................**776**
EISL_VerLoadedModAndCredDataWCert **729**
EISL_VerLoadedModAndCredentialData **727**
EISL_VerLoadedModAndCredentials **731**
EISL_VerLoadedModAndCredWithCert **733**
EncryptData...**241**
EncryptDataFinal...**250**
EncryptDataInit **245**
EncryptDataInitP...**247**
EncryptDataP **244**
EncryptDataUpdate **248**
EventNotifyManager...**874**
FreeKey **286**
GenerateAlgorithmParams **288**
GenerateKey **265**
GenerateKeyP...**268**
GenerateKeyPair..**269**
GenerateKeyPairP **272**
GenerateMac...**226**
GenerateMacFinal **231**
GenerateMacInit **228**
GenerateMacUpdate...**229**
GenerateRandom..**273**
GetOperationalStatistics **291**
GetTimeValue...**292**
Hash algorithm 987
Hypertext Transfer Protocol (HTTP)..................987
Initialize **869**
JAVA 987
KR_GenerateRecoveryFields **680**
KR_GetRecoveredObject **689**
KR_PassThrough **699**
KR_ProcessRecoveryFields **682**
KR_RecoveryRequest **685**
KR_RecoveryRequestAbort **692**
KR_RecoveryRetrieve...**687**
KR_RegistrationRequest **674**
KR_RegistrationRetrieve **677**
KRSP_PassPrivFunc..**697**
Leaf Certificate 987
MDS_Initialize..**629**
MDS_Install **633**
MDS_Terminate **631**
MDS_Uninstall...**634**
Message digest 987
Meta-information 987
ModuleManagerAuthenticate **871**
Nonce 987
ObtainPrivateKeyFromPublicKey **275**
Owned certificate 987
PassThrough **298**
PolicyMaker 987
Pretty Good Privacy (PGP)..................................987
Private key 988
Public key...988
QueryKeySizeInBits..**263**
QuerySize **239**
Random number generators 988
RefreshFunctionTable **875**
RegisterDispatchTable..**872**
RetrieveCounter...**295**
RetrieveUniqueId **294**
Root certificate 988
Secret key 988
Secure Electronic Transaction (SET)..................988
Secure MIME (S/MIME) 988
Secure Sockets Layer (SSL)..................................988
Security Context 989
Security-relevant event 989
Session key...989
Signature 989
Signature chain 989
SignData **205**
SignDataFinal **209**
SignDataInit...**207**
SignDataUpdate...**208**
Symmetric algorithms 989
Terminate **870**
Token..989
TP_ApplyCrlToDb **388**
TP_CertCreateTemplate.......................................**367**
TP_CertGetAllTemplateFields.............................**369**
TP_CertGroupConstruct.......................................**392**
TP_CertGroupPrune **395**
TP_CertGroupToTupleGroup **397**
TP_CertGroupVerify **364**
TP_CertReclaimAbort **358**
TP_CertReclaimKey..**356**
TP_CertRemoveFromCrlTemplate.....................**382**
TP_CertRevoke **379**
TP_CertSign...**371**
TP_ConfirmCredResult...**351**
TP_CrlCreateTemplate **377**

TP_CrlSign **385**
TP_CrlVerify **374**
TP_FormRequest **359**
TP_FormSubmit **361**
TP_PassThrough **402**
TP_ReceiveConfirmation **354**
TP_SubmitCredRequest **345**
TP_TupleGroupToCertGroup **399**
UnwrapKey **279**
UnwrapKeyP **282**
USEE 989
Verification 989
VerifyData **211**
VerifyDataFinal **216**
VerifyDataInit **213**
VerifyDataUpdate **214**
VerifyDevice **296**
VerifyMac **233**
VerifyMacFinal **238**
VerifyMacInit **235**
VerifyMacUpdate **236**
Web of trust 990
WrapKey **276**
WrapKeyP **278**