Apple Cryptographic Service Provider Functional Specification

2005-03-10

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Apple Cryptographic Service Provider Functional Specification

March 10, 2005

1. Introduction

This document describes, at a very high level, the capabilities of the Apple CSP (known herein as "AppleCSP") for Mac OS X. These capabilities are described in terms of key types and formats and supported algorithms.

This document also introduces some nomenclature which is used to describe various attributes of CSSM keys. Terms defined here are shown in **bold** both in their definition and usage.

2. Key Types and Formats

In general, the AppleCSP allows the creation and use of keys in either **reference** or **raw** type. For purposes of this document, a key's **type** is defined by the value of its CSSM_KEYHEADER::BlobType field. A **reference** key has a **type** equal to CSSM_KEYBLOB_REFERENCE. A **raw** key has a **type** equal to CSSM_KEYBLOB_RAW. When keys are generated, the caller can specify one of the two blob **types** via the *KeyAttr flags in the API call (e.g., PublicKeyAttr in CSSM_GenerateKeyPair()).

The CDSA specs allow a number of **formats** for each **type**. A key's **format** is defined by the CSSM_KEYHEADER::CSSM_KEYBLOB_FORMAT field. Generally in the AppleCSP, when generating a key, the caller can either specify a particular format, or else let the CSP determine the default format for the given algorithm and key type by specifying CSSM_KEYBLOB_RAW_FORMAT_NONE for the format. (A key's **class** is one of {Public,Private,Symmetric}. All **reference** keys have format CSSM_KEYBLOB_REF_FORMAT_INTEGER.

The following tables define the supported **formats** for **raw** keys of specified **class** and algorithm.

Symmetric keys (all supported algorithms):

Format = CSSM_KEYBLOB_RAW_FORMAT_OCTET_STRING

Asymmetric keys:

- RSA public keys:
  CSSM_KEYBLOB_RAW_FORMAT_PKCS1 (default, BSAFE compatible)
  CSSM_KEYBLOB_RAW_FORMAT_X509 (openssl compatible)

- RSA private keys:
  CSSM_KEYBLOB_RAW_FORMAT_PKCS1 (openssl compatible)
  CSSM_KEYBLOB_RAW_FORMAT_PKCS8 (default, BSAFE compatible)

- DSA public keys:
  CSSM_KEYBLOB_RAW_FORMAT_X509 (default, openssl compatible)
  CSSM_KEYBLOB_RAW_FORMAT_FIPS186 (BSAFE compatible)

- DSA private keys:
  CSSM_KEYBLOB_RAW_FORMAT_FIPS186 (default, BSAFE compatible)
  CSSM_KEYBLOB_RAW_FORMAT_PKCS8 (SMIME compatible)
  CSSM_KEYBLOB_RAW_FORMAT_OPENSSL (openssl compatible)

- Diffie-Hellman public keys:
  CSSM_KEYBLOB_RAW_FORMAT_PKCS3 (default)
  CSSM_KEYBLOB_RAW_FORMAT_X509 (i.e., SubjectPublicKeyInfo)

- Diffie-Hellman private keys:
  CSSM_KEYBLOB_RAW_FORMAT_PKCS3 (default)
  CSSM_KEYBLOB_RAW_FORMAT_PKCS8 (PKCS8, X9.42 parameters)

- FEE private and public keys:
  CSSM_KEYBLOB_RAW_FORMAT_NONE (default, DER encoded)
  CSSM_KEYBLOB_RAW_FORMAT_ OCTET_STRING (native, raw bytes)

In order to generate a key with a format other than the default format described above, you have to add an appropriate attribute to the key generate context via CSSM_UpdateContextAttributes(). To specify a non-default format when generating public keys, use CSSM_ATTRIBUTE_PUBLIC_KEY_FORMAT. To specify a non-default format when generating private keys, use CSSM_ATTRIBUTE_PRIVATE_KEY_FORMAT.

Note that any key which is generated by the CSP, in any legal type and format, can be used in any call which accepts that key class and algorithm. E.g., whenever an RSA public key is called for, the app may pass in a raw or reference key.; if passing in a raw key, it can be in any format in the above table.

Also note that if performance is of interest, the use of raw RSA and DSA keys is deprecated. Better performance is obtained using reference keys. There is no such advantage for symmetric keys.

3. Key Generation

The following tables list the supported algorithms for the specified key generation functions:

CSSM_GenerateKey

Algorithm	Key size in bytes
CSSM_ALGID_DES	8
CSSM_ALGID_3DES_3KEY	24
CSSM_ALGID_RC2	1..128
CSSM_ALGID_RC4	1..512
CSSM_ALGID_RC5	1..255
CSSM_ALGID_SHA1HMAC (for MAC only)	20..256
CSSM_ALGID_MD5HMAC (for MAC only)	0..2048
CSSM_ALGID_AES	128, 192, 256
CSSM_ALGID_ASC	1..64
CSSM_ALGID_BLOWFISH	4..56
CSSM_ALGID_CAST	5..16

CSSM_GenerateKeyPair

Algorithm	Key size in bits
CSSM_ALGID_RSA	>= 512; key size mod 16 == 0
CSSM_ALGID_DSA	>= 512
CSSM_ALGID_FEE	See below
CSSM_ALGID_DH	512 <= keySize <= 2048

FEE Key Generation

FEE keys can only be generated in a limited number of discrete sizes. Additionally, two parameters may optionally be specified when generating FEE key pairs. These parameters define the elliptic curve upon which subsequent elliptic algebra is performed. The parameters are known as “prime type” and “curve type”. (A discussion of the significance of these parameters is outside the scope of this document.) These parameters, both of which have default values for any given (legal) key size, can be specified as additional attributes in the key generation context passed to CSSM_GenerateKeyPair. The attribute types and values can be found in <Security/cssmapple.h>. The prime type is specified via attribute type CSSM_ATTRIBUTE_FEE_PRIME_TYPE; the curve type is specified via attribute type CSSM_ATTRIBUTE_FEE_CURVE_TYPE.

A complete list of legal key sizes and curve parameters follows. The default curve each key size with multiple sets of curve parameters is shown; when specifying key size, prime type, and curve parameters, only enough info to unambiguously identify the curve is needed.

Key Size (bits)	Prime Type	Curve Type
31	Mersenne	Montgomery
31	Mersenne	Weierstrass (default)
127	Mersenne	Montgomery
128	FEE	Weierstrass
161	FEE	Weierstrass (default)
161	General	Weierstrass
192	General	Weierstrass

Diffie-Hellman Key Generation

Diffie-Hellman (D-H) key pair generation involves either the specification or the generation of D-H algorithm parameters. The contents and meaning of these algorithm parameters is beyond the scope of this document; suffice it to say the D-H algorithm parameters are encapsulated in an opaque blob in the form of a CSSM_DATA. This parameter blob can be passed into the CSP at CSSM_CSP_CreateKeyGenContext time (via the Params argument); it can be calculated explicitly by the CSP via CSSM_GenerateAlgorithmParameters, which returns the calculated parameters to the caller (as well as adding the calculated parameters to the key generation context), or it can be implicitly calculated by the CSP at D-H key generation time if no parameters have been specified in the context. For two parties to perform D-H key exchange, they must use the same D-H algorithm parameters. Distribution of D-H parameters, in the clear and in public, does not compromise the security of the D-H key exchange mechanism in any way and is in fact very common. Calculation of D-H algorithm parameters is a rather time-consuming process - it takes about 90 seconds on an 800 MHz G4 to calculate algorithm parameters for 1024-bit D-H keys - so systems which perform D-H key exchange typically do not calculate these parameters very often - certainly not once for each key pair.

CSSM_DeriveKey

Supported Key Derivation algorithms (AlgorithmID):

- CSSM_ALGID_PKCS5_PBKDF2
- CSSM_ALGID_DH
- CSSM_ALGID_PKCS12_PBE_ENCR
- CSSM_ALGID_PKCS12_PBE_MAC
- CSSM_ALGID_PKCS5_PBKDF1_MD5
- CSSM_ALGID_PKCS5_PBKDF1_MD2
- CSSM_ALGID_PKCS5_PBKDF1_SHA1
- CSSM_ALGID_PBE_OPENSSL_MD5

Derived key algorithm (DeriveKeyType)

The legal key algorithms and sizes are the same as those shown for CSSM_GenerateKey, above.

CSSM_DeriveKey parameters

Parameters for key derivation algorithm-specific. The following is a description of the algorithm-specific parameters for each key derivation algorithm.

- CSSM_ALGID_PKCS5_PBKDF1_MD5
- CSSM_ALGID_PKCS5_PBKDF1_MD2
- CSSM_ALGID_PKCS5_PBKDF1_SHA1
- CSSM_ALGID_PBE_OPENSSL_MD5

  The first three algorithms implement the algorithm defined in PKCS5 version 1.5. The OPENSSL_MD5 algorithm implements a derivation algorithm first implemented in openssl, it's the "native" Openssl key wrapping algorithm, since deprecated but still in widespread use.

  These take an initialization vector in the Params argument. Password is obtained either from the incoming CSSM_CRYPTO_DATA.Seed, or from a Secure Passphrase key in the context (see "Secure Passphrases, below). Salt and iteration count are passed in directly; both are required.

- CSSM_ALGID_PKCS12_PBE_ENCR
- CSSM_ALGID_PKCS12_PBE_MAC

  These algorithms implement the key derivation algorithm defined in PKCS12. There are slightly different "flavors" for deriving MAC key and encryption keys, hence the two ALGID values. These take an initialization vector in the Params argument. Password is obtained either from the incoming CSSM_CRYPTO_DATA.Seed, or from a Secure Passphrase key in the context (see "Secure Passphrases, below). Salt and iteration count are passed in directly; salt is optional, iteration count is required. The passphrase is expressed in UTF8 format. The CSP converts this to Unicode per the PKCS12 specification prior to key derivation.

- CSSM_ALGID_DH

  Diffie Hellman key derivation requires the presence of a private key, passed into CSSM_DeriveKey as BaseKey. A public key is also required; this can be passed either as a CSSM_ATTRIBUTE_PUBLIC_KEY attribute (manually added to the key derivation context) or directly in the Param data, as a raw (unformatted) PKCS3-style Diffie-Hellman key.

- CSSM_ALGID_PKCS5_PBKDF2

  This algorithm implements key derivation as defined in PKCS5 version 2. It takes its parameters in the form of a CSSM_PKCS5_PBKDF2_PARAMS, a

pointer to which is placed in the Param.Data field. Param.Length must be set to sizeof(CSSM_PKCS5_PBKDF2_PARAMS).

The CSSM_PKCS5_PBKDF2_PARAMS .PseudoRandomFunction field must currently be set to CSSM_PKCS5_PRF_HMAC_SHA1. This algorithm does not have any CSMS_KEY as an input, so the BaseKey argument of CSSM_CSP_CreateDeriveKeyContext is NULL.

Secure Passphrases

Key Derivation algorithms which require a passphrase can obtain the passphrase data from a special CSSM_KEY called a Secure Passphrase Key. From the point of view of the CSP being described here, this is simply any key – passed into CSSM_DeriveKey via the BaseKey argument - with algorithm CSSM_ALGID_SECURE_PASSPHRASE. Such a key – which is created elsewhere – contains user-specific passphrase data in UTF8 form. In fact this is currently actually only used in securityd's private instance of the AppleCSP; securityd creates and maintains these Secure Passphrase keys on behalf of the CSPDL, hence on behalf of applications. This mechanism allows for the existence of persistent passphrase objects, the contents of which never appear in the application's address space. Applications using the AppleCSP directly have no reason to use Secure Passphrase keys.

4. Cryptographic Operations

Symmetric Encryption/Decryption

Note "PKCS5/7 padding" means that either PKCS5 or PKCS7 padding may be specified. The two are identical for 8-byte block ciphers.

Algorithm	Modes (CSSM_ALGMODE_xxx)	Notes
CSSM_ALGID_DES	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	
CSSM_ALGID_3DES_3KEY_EDE		
	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	
CSSM_ALGID_AES	CBCPadIV8	Requires PKCS7 padding Requires a block-sized IV
	CBC_IV8	Requires a block-sized IV
	ECBPad	Requires PKCS7 padding
	ECB	

NOTE: AES allows block sizes of 16, 24, and 32 bytes. The default is 16 bytes. The application can specify other block sizes via a CSSM_ATTRIBUTE_BLOCK_SIZE attribute in the symmetric context.

NOTE: the AES implementation has been heavily optimized for the case of 16-byte keys and blocks.

CSSM_ALGID_RC2	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	
CSSM_ALGID_RC4	NONE	No options
CSSM_ALGID_RC5	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	

CSSM_ALGID_BLOWFISH	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	
CSSM_ALGID_CAST	CBCPadIV8	Requires PKCS5/7 padding Requires an 8-byte IV
	CBC_IV8	Requires an 8-byte IV
	ECBPad	Requires PKCS5/7 padding
	ECB	

Asymmetric Encryption/Decryption

Encryption algorithm	Required key algorithm
CSSM_ALGID_RSA	CSSM_ALGID_RSA
CSSM_ALGID_FEED	CSSM_ALGID_FEE
CSSM_ALGID_FEEDExp	CSSM_ALGID_FEE

The FEE encryption algorithms do not require (or accept) the specification of a mode or of a padding specification. They implement a custom padding algorithm.

The RSA algorithm allows for an optional mode and padding (mode and padding are independent). Padding can be CSSM_PADDING_NONE or CSSM_PADDING_PKCS1. Specifying CSSM_PADDING_NONE is not recommended unless the programmer really knows what they are doing; note that with CSSM_PADDING_NONE, the input size for both encrypting and decrypting must be exactly the same as the key size.

RSA Encryption provides a “blinding” option to defend against timing attacks, in which the attacker attempts to glean information about a server’s private key by getting the server to encrypt or decrypot various pieces of attacker-chosen data and measuring the time it takes for the server to do so. Enabling RSA blinding adds a pseudorandom amount of “noisy” time to a private key operationm preventing an attacker from knowing anything about the server’s private key. The Blinding option, normally disabled, is enabled by adding a CSSM_ATTRIBUTE_RSA_BLINDING attribute, data type uint32, to the encryption context. A non zero value for the attribute’s value turns on RSA blinding.

RSA encryption and decryption also allow an optional mode argument (which must be manually added to the asymmetric context via CSSM_UpdateContextAttributes()). These optional modes are CSSM_ALGMODE_PUBLIC_KEY and CSSM_ALGMODE_PRIVATE_KEY. Normally, RSA encryption is performed with a public key; RSA decryption is performed with a private key. In the absence a mode attribute specifying otherwise, these operations will fail with a CSSMERR_CSP_INVALID_KEY_CLASS error if performed with the improper key class. It

is possible to perform RSA encryption with a private key by specifying CSSM_ALGMODE_PRIVATE_KEY, and RSA decryption using a public key by specifying CSSM_ALGMODE_PUBLIC_KEY. (These operations actually constitute the "raw" RSA signature operations.)

Digital signature and verify

The following algorithms are supported, shown with the required key type.

Signature algorithm	Required key algorithm
CSSM_ALGID_SHA1WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_MD5WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_MD2WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_SHA256WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_SHA384WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_512WithRSA	CSSM_ALGID_RSA
CSSM_ALGID_RSA (raw; see note)	CSSM_ALGID_RSA
CSSM_ALGID_SHA1WithDSA	CSSM_ALGID_DSA
CSSM_ALGID_DSA (raw)	CSSM_ALGID_DSA
CSSM_ALGID_FEE_MD5	CSSM_ALGID_FEE
CSSM_ALGID_FEE_SHA1	CSSM_ALGID_FEE
CSSM_ALGID_SHA1WithECDSA	CSSM_ALGID_FEE
CSSM_ALGID_FEE (raw)	CSSM_ALGID_FEE
CSSM_ALGID_ECDSA (raw)	CSSM_ALGID_FEE

Note: “Raw” sign and verify is used to sign or verify a digest which is calculated by the app separately from the sign/verify operation. Raw RSA sign and verify require the specification of the digest algorithm at Sign/Verify time. This is done via the “DigestAlgorithm” to the CSSM_SignData and CSSM_VerifyData functions.

Digest

CSSM_ALGID_MD2	Outsize = 16 bytes
CSSM_ALGID_MD5	Outsize = 16 bytes
CSSM_ALGID_SHA1	Outsize = 20 bytes
CSSM_ALGID_SHA256	Outsize = 32 bytes
CSSM_ALGID_SHA384	Outsize = 48 bytes
CSSM_ALGID_SHA512	Outsize = 64 bytes

The latter three algorithms are variants of a generic class of digest algorithm colloquially known as SHA2.

MAC

CSSM_ALGID_SHA1HMAC	Outsize = 20 bytes
CSSM_ALGID_MD5HMAC	Outsize = 16 bytes
CSSM_ALGID_SHA1HMAC_LEGACY	Outsize = 20 bytes

The CSSM_ALGID_SHA1HMAC_LEGACY algorithm is backwards compatible with the SHA1HMAC algorithm from the Cheetah implementation of the CSP. This implementation, in the BSAFE library, was actually faulty in that it produced different results for the same data input if updates were performed on different boundaries. Unfortunately, Cheetah keychains are not usable unless we can verify their MACs with this legacy algorithm. Obviously future use of this algorithm to actually generate MAC values is deprecated.

Wrap/Unwrap key

A Key Wrap operation is essentially the encrypting of the KeyData field of one key (the Unwrapped Key) with another key (the Wrapping Key) and placing the result in a new CSSM_KEY struct (the Wrapped Key). There are many ways to do this, with many encryption algorithms and many formats of the wrapped key. The encryption algorithm is specified by the application in the incoming encryption context. The format of the wrapped key can optionally be specified by adding a CSSM_ATTRIBUTE_WRAPPED_KEY_FORMAT attribute to the encryption context. In the absence of such a format specification the CSP will pick a default format appropriate to the type of keys being used as described below.

A rule of thumb is that you can wrap any key with any other key, but certainly not all formats work with all keys. You should probably make a serious attempt to fully understand what's going on and what all the various options and parameters mean before attempting to use any of this.

A special form of Key Wrapping is called Null Wrap. No encryption is involved; this converts a reference key to a raw key. In this case the application passes in a symmetric encryption context with no key and an AlgorithmID of CSSM_ALGID_NONE.

The Wrapped Key Formats supported by the CSP are described here:

- CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS7

  Only valid for wrapping symmetric keys. See PKCS7.

- CSSM_KEYBLOB_WRAPPED_FORMAT_PKCS8

  Only valid for wrapping private asymmetric keys. See PKCS8.

- CSSM_KEYBLOB_WRAPPED_FORMAT_OPENSSL

  Only valid for wrapping private asymmetric keys. Implements Openssl's legacy key wrapping format, currently deprecated in favor of PKCS8 but still in widespread use.

- CSSM_KEYBLOB_WRAPPED_FORMAT_APPLE_CUSTOM

  Apple-custom format used for encrypting keychain items. The wrapping key can not be an AES key (or any other key whose algorithm requires an initialization vector of larger than 8 bytes).

If no WRAPPED_KEY_FORMAT attribute is found in the incoming context, the default format is calculated based on the incoming Unwrapped Key as follows:

- Wrapped Key is symmetric: default format = PKCS7

- Wrapped Key is a public asymmetric: default format = APPLE_CUSTOM
- Wrapped Key is FEE Private asymmetric: default format = APPLE_CUSTOM
- Wrapped key is other Private asymmetric: default format = PKCS8

Another optional attribute during a WrapKey operation specifies the format of the raw key just prior to encrypting it (or in the case of a NULL wrap, the format of the raw key to be returned). This differs from the Wrapped Key Format. Depending on whether the Unwrapped Key is a Private, Public, or Symmetric, this attribute type is CSSM_ATTRIBUTE_{PRIVATE,PUBLIC,SESSION}_KEY_FORMAT, respectively. If this attribute is not specified then the following default values for blob format are used:

- Wrapped Format = PKCS8 : blob format is PKCS8 for RSA keys, FIPS186 for DSA keys, default format for other private keys.

- Wrapped Format = OPENSSL : blob format is PKCS1 for RSA keys, OPENSSL for DSA keys, default format for all other keys.

Performing a CSSM_UnwrapKey follows most of the above rules and regulations; an Unwrap Key operation consists of decrypting the key material of a Wrapped Key with another key called the Unwrapping Key, and placing the result in a third key called, you guessed it, the Unwrapped Key. The wrap format of the Wrapped Key must be valid and present in the Wrapped Key's KeyHeader.BlobFormat. No defaults, no guessing; this has to be accurate.

A Null Unwrap is used to convert a raw (not wrapped, not encrypted) key into a reference key. It uses a symmetric encryption context no key and an AlgorithmID of CSSM_ALGID_NONE.

Pseudo Random Number Generation (CSSM_GenerateRandom())

The CSP supports one PRNG algorithm, ALGID_APPLE_YARROW. This algorithm uses a version of Yarrow, originally written by Counterpane Labs, modified for the OS X platform. Specifying a seed in the cryptographic context is optional; if present, the seed data is added to the current entropy pool but does **not** fully specify the state of the Yarrow PRNG. (This operation, in which seed data is added to the entropy pool, is only possible when the current process is running as root.) Note that specifying the same seed in multiple calls to (CSSM_CSP_CreateRandomGenContext(), CSSM_GenerateRandom()) will **not** result in the generation of repeatable pseudorandom numbers.

Revision History

Version	Date	Changes
0.1	4/18/00	Initial distribution.
0.2	5/16/00	Updated for alpha release of CSP.
0.3	7/13/00	Added CSSM_ALGID_3DES key generation and derivation. Added CSSM_ALGID_3DES_3KEY_EDE encrypt/decrypt. Removed CSSM_ALGMODE_ECB modes from feedback ciphers. Cleaned up IV specification. Added ALGID_APPLE_YARROW PRNG.
0.4	10/25/00	Added AES symmetric encryption.
0.5	4/26/01	Added ASC; added signature algorithms; modified modes for symmetric encryption algorithms
0.6	9/5/01	Added CSSM_ALGID_SHA1HMAC_LEGACY Added raw DSA, FEE, and ECDSA signatures
0.7	4/30/02	Added Diffie-Hellman Key gen and DeriveKey Added CDSA_PADDING_NONE option for RSA Added CSSM_ALGMODE_{PUBLIC,PRIVATE}_KEY for RSA
0.8	1/12/05	Bring up to date for Tiger. Added many new key formats. Added MD2, SHA256, SHA384, SHA512, CAST, BLOWFISH. Added many key derivation algorithms. Expanded Key wrap/unwrap discussion.
0.8	3/10/05	Added SHA-2 signature algorithms.

