MatrixCMS API and Technical Reference

TABLE OF CONTENTS

1 FEATURES AND CONFIGURATION

The MatrixCMS implementation supports the creation and parsing of three CMS data types:

- Signed-Data Content Type (RFC 5652)
- Authenticated-Enveloped-Data Content Type (RFC 5083)
- Compressed-Data Type (RFC 3274)

The Signed-Data (SD) and Authenticated-Enveloped-Data (AED) types use Elliptic Curve public key operations for the key agreement and digital signature operations.

The Signed-Data and Authenticated-Enveloped-Data types support SHA-2 digest algorithms SHA256, SHA384, and SHA512.

The Authenticated -Enveloped-Data type supports AES_GCM and AES_CBC_CMAC encryption-with-authentication algorithms.

The Compressed-Data (CD) type does not compress or decompress the data but does assume zlib as the compression algorithm.

Each data type can be parsed or created in a single-pass atomic mode or an Init/Update/Final streaming mode.

The following compile time settings can be found in the *utilities/cms/matrixCmsConfig.h* file.

USE_MCMS_STREAMING_SD_CREATE	Enable the SD streaming creation APIs
USE_MCMS_ATOMIC_SD_CREATE	Enable the SD atomic creation APIs
USE_MCMS_STREAMING_SD_PARSE	Enable the SD streaming parsing APIs
USE_MCMS_ATOMIC_SD_PARSE	Enable the SD atomic parsing APIs
USE_MCMS_STREAMING_AED_CREATE	Enable the AED streaming creation APIs
USE_MCMS_ATOMIC_AED_CREATE	Enable the AED atomic creation APIs
USE_MCMS_STREAMING_AED_PARSE	Enable the AED streaming parsing APIs
USE_MCMS_ATOMIC_AED_PARSE	Enable the AED atomic parsing APIs
MCMS_EMPTY_AED_AUTH_ATTRIBS	Empty authenticated attributes must be enabled if stream parsing AED with AES_GCM. This is because the authenticated data must be available to initialize the cipher but it appears after the data in the ASN.1.
MCMS_EMPTY_CBC_CMAC_PARAMS	This is a customer specific setting to exclude the parameters from the AES_CBC_CMAC ASN.1 encodings. Defining this option will imply an empty value for the Initialization Vector for the AES_CBC_CMAC algorithm as the requirement calls for.
USE_MCMS_STREAMING_CD_CREATE	Enable the CD streaming creation APIs

USE_MCMS_ATOMIC_CD_CREATE	Enable the CD atomic creation APIs
USE_MCMS_STREAMING_CD_PARSE	Enable the CD streaming parsing APIs
USE_MCMS_ATOMIC_CD_PARSE	Enable the CD atomic parsing APIs

1.1 Building the CMS Library

Matrix packages that include CMS functionality can be identified by the presence of the *./utilities/cms* directory. There is a *Makefile* at that directory level that will generate a *libmatrixcms.a* static library.

In order to achieve a successful compile, the required crypto algorithms must be enabled in *./crypto/cryptoConfig.h*. The set of algorithms that are disabled by default and must be enabled:

```
USE_ECC
USE_AES_CMAC
USE_AES_WRAP
USE_AES_GCM
```

AESNI Incompatibility

The Intel hardware accelerated AES algorithm, AESNI, does not function correctly with CMS in streaming modes. Additionally, AESNI does not support AES with 192 bit key sizes. You must disable AESNI if working with CMS in streaming mode or if using AES192.

1.2 Running cmsTest

A comprehensive CMS test application is included in the *./utilties/cms/test* directory and depends on the *libmatrixcms.a* library have been built.

Once compiled, invoke *./cmsTest* to run.

NOTE: If running in streaming parse mode, crypto trace will not be a suitable run-time setting due to the ASN.1 "parse errors" that will occur while testing with the partial data.

2 SIGNED-DATA CONTENT TYPE API

The Signed-Data Content Type is defined in RFC 5652. It defines a standard ASN.1 encapsulation mechanism to transport a digital signature. A digital signature is a private key encryption of a digest hash of some given data. This data type allows authentication of arbitrary data.

Currently, the MatrixCMS library supports ECDSA SHA-2 algorithms for creating and validating signatures.

2.1 Signed Data Creation

There are two available mechanisms to create a Signed-Data type. The first is the atomic version in which the data contents are given in a single parameter to the `matrixCmsCreateSignedData` function.

The second mechanism is a streaming version that uses an Init/Update/Final API. The APIs for this method are `matrixCmsInitCreateSignedData`, `matrixCmsUpdateCreateSignedData`, and `matrixCmsFinalCreateSignedData`. Each of these three APIs will return a portion of the full Signed-Data Content Type to the caller who can append them in a single file (or memory buffer) or send them to the receiving entity as they are created.

The signers X.509 certificate is always included in the Signed-Data Type in this current implementation.

2.1.1 Detached vs. Attached signed content

In some use-cases the plaintext data that is being signed is not included in the Signed-Data Content Type itself. This is called a **detached** mode of operation and is the default for atomic SD creation. In this mode, it is assumed the data will be exchanged via some other mechanism. To include the plaintext data in the SD for atomic creations, include the `MCMS_FLAGS_SD_NODETACH` define in the flags parameter.

If running in detached mode it is not necessary to pass in the entire data contents. The data may be pre-hashed and passed as the contents with the flag `MCMS_FLAGS_SD_CONTENT_PREHASHED`.

Detached mode is not available to the stream creation APIs. This is because there should be no reason to stream-generate a SD if there is not a large quantity of data to include in the data type.

2.2 matrixCmsCreateSignedData

```
int32 matrixCmsCreateSignedData(psPool_t *pool, unsigned char *content,
            int32 contentLen, int32 contentType, psX509Cert_t *cert,
            psPubKey_t *key, int32 hashId, unsigned char **outputBuf,
            int32 *outputLen, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
content	input	The data that will be signed OR the pre-hashed data
contentLen	input	The byte length of data
contentType	input	The OID type of data that is being signed. Must be CMS_PKCS7_DATA, CMS_PKCS7_SIGNED_DATA, CMS_PKCS9_AUTH_ENVELOPED_DATA, or CMS_PKCS9_COMPRESSED_DATA
cert	input	The certificate of the signing entity
key	input	The private key of the signing entity
hashId	input	The digest algorithm for the desired signature
outputBuf	output	The DER encoded Signed-Data Type
outputLen	output	The byte length of `outputBuf`
flags	input	Creation control flags. See the discussion below

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
< 0	Failure.
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_SUCCESS	Success.

This is the atomic Signed-Data Content Type creation function.

The `psX509Cert_t *cert` parameter will have been obtained using `psX509ParseCertFile` or `psX509ParseCert`. The certificate parse function MUST be called with a flags value of `CERT_STORE_UNPARSED_BUFFER | CERT_STORE_DN_BUFFER` to keep the needed encoded portions of the certificate available to CMS. For information on encoding a certificate chain instead of a single certificate, see Creation Control Flags below.

The `content` and `contentLen` will identify the data to be signed. The `contentType` should identify the data that is being signed and this OID value will be written as the eContentType member of the encapContentInfo encoding. If the data being signed is a generic blob use the `CMS_PKCS7_DATA` identifier. Otherwise, choose the `CMS_PKCS7_SIGNED_DATA`, `CMS_PKCS9_AUTH_ENVELOPED_DATA`, or `CMS_PKCS9_COMPRESSED_DATA` if the signed data is itself a CMS data type. The See the Creation Control Flags section below for more information regarding detached and pre-hashed content.

The `psPubKey_t *key` parameter is the signing private key associated with the certificate and will have been obtained using `psEcdsaParsePrivKey`.

The `hashId` parameter shall be one of `MCMS_SHA256_ALG`, `MCMS_SHA384_ALG`, or `MCMS_SHA512_ALG`.

The `outputBuf` data must be freed using `psFree` when no longer needed.

Creation Control Flags

The `flags` parameter controls the options on whether the plaintext signed data will be attached, how the signer's certificate is identified within the Signed-Data, and whether the ContentInfo header will be written to the SignedData encoding.

The first configuration option is to determine whether the plaintext data will be attached. By default, it will not be attached. When the data is not attached it is not necessary to pass in the entire contents of the data. The caller may choose to only pass in the hash digest of the data in this case. If the `content` parameter is the pre-hashed digest value the value `MCMS_FLAGS_SD_CONTENT_PREHASHED` must be included in the flags parameter.

To attach the full plaintext data, include the `MCMS_FLAGS_SD_NODETACH` value in `flags`.

To encode a certificate chain instead of a single certificate, include the `MCMS_FLAGS_SD_CERT_CHAIN` value in `flags`. In addition, the `cert` parameter must point to the child-most certificate, with the `next` member of each certificate pointing to its issuer certificate. It is possible to automatically setup the links properly by concatenating the chain certificates into a PEM file in child-to-parent order and parsing the file with `psX509ParseCertFile`.

The second configuration option is to determine how the signer's certificate will be identified in the SD. The options are between using the X.509 issuer Distinguished Name and Serial Number or the X.509 Subject Key Identifier extension. The default is IssuerAndSerialNumber and there is no flags value to identify this choice. Supplying the value `MCMS_FLAGS_SD_SUBJECT_KEY_ID` to the flags will create the SD with the SubjectKeyIdentifier instead.

The final configuration option is to determine whether the outer ContentInfo ASN.1 header is written to the output. If the ContentInfo should be excluded, add the `MCMS_FLAGS_NO_CONTENT_INFO` flag.

The table below shows some viable combinations of flags for creating Signed-Data types.

Flag combinations	**Meaning**
MCMS_FLAGS_SD_NODETACH	Full plaintext data passed to content and will be included in the Signed-Data type. IssuerAndSerialNumber will be used as the certificate identification as the SignerIdentifier.
MCMS_FLAGS_SD_CONTENT_PREHASHED	Detached mode. Pre-hashed data passed to content. IssuerAndSerialNumber will be used as the certificate identification as the SignerIdentifier.
MCMS_FLAGS_SD_NODETACH | MCMS_FLAGS_SD_SUBJECT_KEY_ID	Full plaintext data passed to content and will be included in the Signed-Data type. The SignerIdentifier will use the SubjectKeyId extension of the certificate for identification
0	Detached mode. Full plaintext data passed to content but it will not be included in the data. IssuerAndSerialNumber will be used as the certificate identification as the SignerIdentifier.
MCMS_FLAGS_SD_SUBJECT_KEY_ID	Detached mode. Full plaintext data passed to content but it will not be included in the data. The SignerIdentifier will use the SubjectKeyId extension of the certificate for identification
MCMS_FLAGS_SD_CONTENT_PREHASHED | MCMS_FLAGS_SD_SUBJECT_KEY_ID	Detached mode. Pre-hashed data passed to content. The SignerIdentifier will use the SubjectKeyId extension of the certificate for identification

2.3 matrixCmsInitCreateSignedData

```
int32 matrixCmsInitCreateSignedData(psPool_t *pool, psX509Cert_t *cert,
                  psPubKey_t *key, int32 hashId, int32 contentType,
                  unsigned char **outputBuf, int32 *outputLen,
                  int32 flags, cmsSdStream_t **sdCtx);
```

Parameter	**Input/Output**	**Description**
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
cert	input	The certificate of the signing entity
key	input	The private key of the signing entity
hashId	input	The digest algorithm for the desired signature
contentType	input	The OID type of data that is being signed. Must be CMS_PKCS7_DATA, CMS_PKCS7_SIGNED_DATA, CMS_PKCS9_AUTH_ENVELOPED_DATA, or CMS_PKCS9_COMPRESSED_DATA

outputBuf	output	The initial portion of a BER encoded Signed-Data Type
outputLen	output	The byte length of outputBuf
flags	input	See the discussion below
sdCtx	output	The context that will be passed to the Update/Final components

Return Value	**Description**
PS_MEM_FAIL	Failure. Internal memory allocation failure
< 0	Failure.
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_SUCCESS	Success.

Begins the streaming creation of the Signed-Data Content Type. This mode can only be used when the full plaintext data will be attached to the SD. There is no detached option for stream creation because such a data type should be small enough to create with the atomic version.

On success, the `outputBuf` will contain the BER encoded Signed-Data type all the way to the constructed OCTET STRING of the EncapsulatedContentInfo. Each subsequent `matrixCmsUpdateCreateSignedData` function calls will output a component OCTET STRING of the content data that is passed to it. The `matrixCmsFinalCreateSignedData` function call will complete the signature process and return the final BER encoding to complete the full data type.

The `psX509Cert_t *cert` parameter will have been obtained using `psX509ParseCertFile` or `psX509ParseCert`. The certificate parse function MUST be called with a flags value of `CERT_STORE_UNPARSED_BUFFER | CERT_STORE_DN_BUFFER` to keep the needed encoded portions of the certificate available to CMS.

The `psPubKey_t *key` parameter is the signing private key associated with the certificate and will have been obtained using `psEcdsaParsePrivKey`.

The `hashId` parameter shall be one of `MCMS_SHA256_ALG`, `MCMS_SHA384_ALG`, or `MCMS_SHA512_ALG`.

The `contentType` should identify the data that is being signed and this OID value will be written as the eContentType member of the encapContentInfo encoding. If the data being signed is a generic blob use the `CMS_PKCS7_DATA` identifier. Otherwise, choose the `CMS_PKCS7_SIGNED_DATA`, `CMS_PKCS9_AUTH_ENVELOPED_DATA`, or `CMS_PKCS9_COMPRESSED_DATA` if the signed data is itself a CMS data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `sdCtx` context parameter must be freed at the conclusion of the streaming creation using `matrixCmsFreeStreamCreatedSignedData`

Creation Control Flags

This streaming mode can only be used if the content will be included in the Signed-Data structure. There should be no reason to require a streaming mode for detached content because the pre-hash of the data can be performed via the streaming mechanism of a SHA-2 Init/Update/Final API.

Therefore, the creation control flags are only used for certificate identification and whether the ContentInfo header is to be included when using this stream creation method. If the ContentInfo should be excluded, add the `MCMS_FLAGS_NO_CONTENT_INFO` flag.

Flag combinations	
0	IssuerAndSerialNumber will be used as the certificate identification as the SignerIdentifier.
MCMS_FLAGS_SD_SUBJECT_KEY_ID	The SignerIdentifier will use the SubjectKeyId extension of the certificate for identification

2.4 matrixCmsUpdateCreateSignedData

```
int32 matrixCmsUpdateCreateSignedData(psPool_t *pool, cmsSdStream_t *sdCtx,
                                unsigned char *content, int32 contentLen,
                                unsigned char **outputBuf, int32 *outputLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. MUST be same pool as `matrixCmsInitCreateSignedData`. NULL if unused
sdCtx	input	The context from a previous call to `matrixCmsInitCreateSignedData`
content	input	The next portion of the data that will be signed
contentLen	input	The byte length of `content`
outputBuf	output	The next portion of a BER encoded Signed-Data Type
outputLen	output	The byte length of `outputBuf`

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.

Continues the streaming creation of the Signed-Data Content Type.

On success, the `outputBuf` will contain the BER encoded OCTET STRING of the content that should be appended to the output of a previous call to `matrixCmsUpdateCreateSignedData` (or `matrixCmsInitCreateSignedData` if this is the first portion). Each subsequent `matrixCmsUpdateCreateSignedData` function calls will output a component OCTET STRING of the content data. The `matrixCmsFinalCreateSignedData` function call will complete the signature process and return the final BER encoding to complete the full data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `sdCtx` context parameter must be freed at the conclusion of the streaming creation using `matrixCmsFreeStreamCreatedSignedData`

2.5 matrixCmsFinalCreateSignedData

```
int32 matrixCmsFinalCreateSignedData(psPool_t *pool, cmsSdStream_t *sdCtx,
                          unsigned char **outputBuf, int32 *outputLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. MUST be same pool as `matrixCmsInitCreateSignedData`. NULL if unused
sdCtx	input	The context from a previous call to `matrixCmsInitCreateSignedData`
outputBuf	output	The final portion of a BER encoded Signed-Data Type
outputLen	output	The byte length of `outputBuf`

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.
< 0	Failure.

Finalizes the streaming creation of the Signed-Data Content Type.

On success, the `outputBuf` will contain the remainder of the BER encoded Signed-Data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `sdCtx` context parameter must be freed at the conclusion of the streaming creation using `matrixCmsFreeStreamCreatedSignedData`

2.6 matrixFreeStreamCreatedSignedData

```
void matrixFreeStreamCreatedSignedData(cmsSdStream_t *sdCtx);
```

Parameter	Input/Output	Description
sdCtx	input	The context from a previous call to `matrixCmsInitCreateSignedData`

Frees the SD stream creation context.

2.7 Signed Data Parsing

There are two available mechanisms to parse a Signed-Data type. The first is the atomic version that uses the `matrixCmsParseSignedData` function to parse the data type in a single pass. After the parse is complete, the signature confirmation is performed with the `matrixCmsConfirmSignature` API. This process allows the user to examine the data fields after the parse phase to validate the signing certificate that is then used to confirm the signature.

The second mechanism is a stream parsing based flow in which an Init/Update/Final sequence is used to process the SD. The APIs for this method are `matrixCmsInitParseSignedData`, `matrixCmsUpdateParseSignedData`, and `matrixCmsFinalParseSignedData`. The final phase performs the signature confirmation.

2.8 matrixCmsParseSignedData

```
int32 matrixCmsParseSignedData(psPool_t *pool, unsigned char *sdBuf,
            uint32 sdBufLen, cmsSignedData_t **signedData, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
sdBuf	input	ASN.1 formatted signed data to parse
sdBufLen	input	Byte length of sdBuf
signedData	output	Signed data structure
flags	input	Whether the incoming SignedData type includes the ContentInfo header. Set to MCMS_FLAGS_NO_CONTENT_INFO if absent. Set to 0 if the full CMS data type is being parsed.

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_PARSE_FAIL	Failure. SignedData ASN.1 parse failure
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_SUCCESS	Success. The `signedData` can now be validated with `matrixCmsConfirmSignature`
MCMS_PARTIAL	Success. The ASN.1 stream is DER encoded and the passed in `sdBufLen` is not large enough based on the initial encoded size of the Content Type. The caller must retrieve the remainder of the data and call again. It is not possible to return this code with a BER encoded ASN.1 stream that uses indefinite-length encoding.

The atomic parse of a CMS Signed-data Content Type. The function returns the parsed information in a `cmsSignedData_t` structure that will be passed as input to `matrixCmsConfirmSignature`.

The most important fields in the signedData structure will be `cert`, `eContent`, and `eContentLen`. The `cert` is the X.509 certificate whose private key was used to sign the data and should be validated by the user. The `eContent` and `eContentLen` will contain the data from the Encapsulated Content that has been signed. If the Signed-Data type was generated in detached mode, `eContent` will be NULL.

The caller should take this opportunity before calling `matrixCmsConfirmSignature` to locate the signing certificate in the `cmsSignedData_t` structure and confirm a trusted Certificate Authority has issued it.

On success, `signedData` must be freed with `matrixCmsFreeParsedSignedData` when no longer needed.

2.9 matrixCmsConfirmSignature

```
int32 matrixCmsConfirmSignature(psPool_t *pool,
            cmsSignedData_t *signedData, unsigned char *data,
            int32 dataLen, psX509Cert_t *validationCert);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
signedData	input	Populated `signedData` structure from a previous call to `matrixCmsParseSignedData`
data	input	Optional. Plaintext data to confirm signature. Required if not attached in the `signedData` structure
dataLen	input	Byte length of `data`
validationCert	input	Optional. X.509 certificate to perform signature validation. Required if not provided in `signedData` structure

Return Value	Description
PS_ARG_FAIL	Failure. Bad input parameters
MCMS_SIG_FAIL_CONTENT_MISMATCH	Failure. The provided data did not match what was attached in the `signedData` structure
MCMS_SIG_FAIL_NO_CONTENT	Failure. No data was provided and no content was found attached in the `signedData` structure
MCMS_SIG_FAIL_BAD_USER_CERT	Failure. The user provided `validationCert` did not match the certificate found in the `signedData` structure
MCMS_SIG_FAIL_NO_CERT	Failure. No `validationCert` was provided and no certificate was found embedded in the `signedData` structure
MCMS_SIG_FAIL_SIGNATURE_FAIL	Failure. The signature operation failed.
MCMS_SIG_FAIL_SIGNATURE_MISMATCH	Failure. The signature operation succeeded but the `signedData` digest comparison failed.

MCMS_SIG_FAIL_CONTENT_HASH_MISMATCH	Failure. The signature operation succeeded and the `signedData` hash digest matched but the raw digest of the content did not match the value in `signedData`
PS_SUCCESS	The signature was successfully authenticated

This function performs the signature validation of an SD that was parsed with `matrixCmsParseSignedData`.

If used, the `psX509Cert_t *validation` parameter will have been obtained using `psX509ParseCertFile` or `psX509ParseCert`.

`signedData` must be freed with `matrixCmsFreeParsedSignedData` when no longer needed.

2.10 matrixCmsInitParseSignedData

```
int32 matrixCmsInitParseSignedData(psPool_t *pool,
                  unsigned char *sdBuf, uint32 sdBufLen,
                  cmsSignedData_t **sdCtx, unsigned char **remainder,
                  uint32 *remainderLen, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
sdBuf	input	The first portion of an SD type to decrypt.
sdBufLen	input	The byte length of `sdBuf`
sdCtx	output	On success, the context to use as input to the parse routines to follow
remainder	output	The remaining SD data from `sdBuf` that this Init function did not process. The next call to `matrixCmsUpdateParseSignedData` must begin with this remainder data
remainderLen	output	The byte length of any `remainder`
flags	input	Whether the incoming SignedData type includes the ContentInfo header. Set to MCMS_FLAGS_NO_CONTENT_INFO if absent. Set to 0 if the full CMS data type is being parsed.

Return Value	Description
PS_LIMIT_FAIL	Failure. The input buffer did not contain enough of the SD to complete the Init. The buffer must be appended with additional SD data and called again. The original `sdBuf` is NOT saved within this function and must be resubmitted along with the newly appended data.
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_PARSE_FAIL	Failure. The SD type could not be parsed at the ASN.1 level
PS_MEM_FAIL	Failure. An internal memory allocation failed
PS_SUCCESS	Success. The initialization is complete and `matrixCmsUpdateParseSignedData` can now be called.

This is the first call to perform a stream parse of a Signed-Data type. This function requires that all the SD data up to the signed content itself be available in the `sdBuf` parameter. The function will return `PS_LIMIT_FAIL` if this requirement is not met and the user must append additional SD data and call again.

The `sdCtx` output context will become input to the other streaming parse routines for this SD.

The `remainder` output parameter points to the `sdBuf` location where this function stopped processing. The `remainder` must be the start of the data that is passed to the first call to `matrixCmsUpdateParseSignedData` to continue the parse.

sdCtx must be freed with matrixCmsFreeParsedSignedData when the parse is complete.

2.11 matrixCmsUpdateParseSignedData

```
int32 matrixCmsUpdateParseSignedData(psPool_t *pool,
                        cmsSignedData_t *sdCtx, unsigned char *sdBuf,
                        uint32 sdBufLen, unsigned char **data, uint32 *dataLen,
                        unsigned char **remainder, uint32 *remainderLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
sdCtx	input	The context from a previously successful call to matrixCmsInitParseSignedData
sdBuf	input/output	The next portion of an SD type to process.
sdBufLen	input	The byte length of sdBuf
data	output	If the plaintext signed data is included in the SD, this parameter will hold that data
dataLen	output	The byte length of the data output
remainder	output	The remaining SD data from sdBuf that this Update function did not process. Only used with MCMS_PARTIAL return codes
remainderLen	output	The byte length of any remainder

Return Value	Description
PS_SUCCESS	Success. The end of the contents has been found. matrixCmsFinalSignedData can now be called.
MCMS_PARTIAL	Success. The update successfully completed but there is still more data expected. This function must be called again with more SD. The remainder parameter will indicate where the next SD data should begin
PS_LIMIT_FAIL	Failure. The input buffer did not contain enough data to complete the update. Append more SD data and call again.
PS_PARSE_FAIL	Failure. The SD type could not be parsed at the ASN.1 level

This is the continuation of the SD stream parse. This function will be called with SD data until PS_SUCESS is returned. If the plaintext signed data is included in the SD it will be returned in the data and dataLen output parameters. The data parameter points into sdBuf so the caller should be aware of the sdBuf lifecycle if the data needs to be saved aside.

The MCMS_PARTIAL return code will be returned while parsing the SD if more data is expected. In this case the caller should still test the data and dataLen parameters for data that was successfully parsed. Additionally, the caller must use the remainder and remainderLen parameters as the start of the next sdBuf that is passed to this function.

The PS_LIMIT_FAIL return code can occur while parsing the SD ASN.1 that follows the plaintext content. If this return code is hit, the caller must append additional SD to the sdBuf and call again. The remainder and data parameters will not be used in this return code case.

sdCtx must be freed with matrixCmsFreeParsedSignedData when the parse is complete.

2.12 matrixCmsFinalParseSignedData

```
int32 matrixCmsFinalParseSignedData(psPool_t *pool,
                        cmsSignedData_t *sdCtx, const unsigned char *hash,
                        uint32 hashLen, psX509Cert_t *validationCert);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
sdCtx	input	The context from a previously successful call to `matrixCmsInitParseSignedData`
hash	input	The SHA2 hash of the plaintext data whose signature is being authenticated
hashLen	input	The byte length of `hash`. Must be SHA256_HASH_SIZE, SHA384_HASH_SIZE, or SHA512_HASH_SIZE
validationCert	input	Optional signer certificate.

Return Value	Description
PS_ARG_FAIL	Failure. Bad input parameters
MCMS_SIG_FAIL_BAD_USER_CERT	Failure. The user provided `validationCert` did not match the certificate found in the `sdCtx` structure
MCMS_SIG_FAIL_NO_CERT	Failure. No `validationCert` was provided and no certificate was found embedded in the `signedData` structure
MCMS_SIG_FAIL_SIGNATURE_FAIL	Failure. The signature operation failed.
MCMS_SIG_FAIL_SIGNATURE_MISMATCH	Failure. The signature operation succeeded but the `sdCtx` digest comparison failed.
MCMS_SIG_FAIL_CONTENT_HASH_MISMATCH	Failure. The signature operation succeeded and the `sdCtx` hash digest matched but the raw digest of the content provided by the user did not match the value in `sdCtx`
PS_SUCCESS	The signature was successfully authenticated

This is the final step of a SD stream parse and performs the signature authentication.

The caller should look in the `cmsSignedData_t` structure after `PS_SUCCESS` has been returned from `matrixCmsUpdateParseSignedData` to determine the input parameters to this function. The `hash` and `hashLen` parameters are the user-calculated values of the plaintext data that is being authenticated.

To determine which hash algorithm was used in the creation of the SD the user can examine the `digestId` member of the `sdCtx` structure. Possible supported values are `OID_SHA256_ALG`, `OID_SHA384_ALG`, and `OID_SHA512_ALG` to identify the correct SHA-2 algorithm. If needed, the Matrix SHA-2 functions may be used to calculate the hash value. For example, `psSha256Init`, `psSha256Update`, and `psSha256Final` are the routines for `OID_SHA256_ALG` identities.

A `validationCert` will be required in some cases to provide the public key portion for the signature validation. In MatrixCMS-created SD types, the signer certificate will be embedded in the data type but users may be working with a third party SD type or may simply wish to confirm the certificate by looking at the certificate identification within the `sdCtx`. Certificates may be identified in one of two ways: IssuerAndSerialNumber or SubjectKeyIdentifier. The choice is found in the `version` member of the `cmsSignerId_t *signerId` pointer which itself is referenced through the `cmsSignerInfos *signers` member of the sdCtx. So in C, sdCtx->signers->signerId ->version will get the user to the certificate identification option. A version value of 1 is IssuerAndSerialNumber. A version value of 3 is SubjectKeyIdentifier.

If the version is 1 the signers issuer distinguished name will be found in the dn member in the same cmsSignerId_t *signerId structure where version was found. The serial number will be found in the sn member with a length of snLen. NOTE: It is the ISSUER distinguished name in the X.509 certificate… not the SUBJECT distinguished name.

If the version is 3 the subject key identifier extension of the signer will be found in the sn member and will have a length of snLen.

sdCtx must be freed with matrixCmsFreeParsedSignedData when the parse is complete.

2.13 matrixCmsFreeParsedSignedData

```
void matrixCmsFreeParsedSignedData(cmsSignedData_t *signedData);
```

Parameter	Input/Output	Description
signedData	input	The context from a previous call to matrixCmsParseSignedData or matrixCmsInitParseSignedData

Frees the SD parsing data structure.

3 AUTHENTICATED-ENVELOPED-DATA CONTENT TYPE API

The Authenticated-Enveloped-Data (AED) Content Type is defined in RFC 5083. It defines a standard ASN.1 format for transporting arbitrary content that is both authenticated and encrypted.

MatrixCMS currently supports the "Key Agreement" technique for deriving AES keys that are used to encrypt the data. ECDH is the supported public key algorithm for key agreement.

MatrixCMS currently supports AES_GCM and AES_CBC_CMAC as the authenticated encryption modes.

3.1 AED Creation

There are two available mechanisms to create an Authenticated-Enveloped-Data type.

The first is the atomic version in which the entire data contents are given in a single parameter to the `matrixCmsCreateAuthEnvData` function.

The second is a streaming version that uses an Init/Update/Final flow to create the data type. The APIs for this method are `matrixCmsInitCreateAuthEnvData`, `matrixCmsUpdateCreateAuthEnvData`, and `matrixCmsFinalCreateAuthEnvData`. Each of these streaming APIs will return a portion of the full Authenticated-Enveloped-Data Content Type to the caller who can append them in a single file (or memory buffer) or send them to the receiving entity for them to reconstruct.

3.2 matrixCmsCreateAuthEnvData

```
int32 matrixCmsCreateAuthEnvData(psPool_t *pool,
                                 const psX509Cert_t *myCert,
                                 const psPubKey_t *privKey,
                                 const psX509Cert_t *recipientCert,
                                 const int32 keyMethod,
                                 const int32 encryptMethod,
                                 const int32 wrapMethod,
                                 const int32 keyAgreeScheme,
                                 unsigned char *content,
                                 const int32 contentLen,
                                 const in32  contentType,
                                 unsigned char **outputBuf,
                                 int32 *outputLen,
                                 const int32 flags);
```

Parameter	**Input/Output**	**Description**
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
myCert	input	Optional. The originator certificate. Must be included if embedding certificate in AED. See **Creation Control Flags** section below for more info
privKey	input	Optional. The private key of the originator used the for key agreement algorithm. May be omitted if using ephemeral keys. See **Creation Control Flags** section below for more info
recipientCert	input	Required. The certificate of the receiving entity
keyMethod	input	MCMS_AED_KEY_AGREE_METHOD
encryptMethod	input	The authenticated encryption algorithm. See below
wrapMethod	input	The AES key wrap algorithm. See below
keyAgreeScheme	input	The ECDH key agreement scheme. See below
content	input	The content to be encrypted and tagged
contentLen	input	Byte length of content

Rambus

contentType	input	The OID type of data that is being signed. Must be CMS_PKCS7_DATA, CMS_PKCS7_SIGNED_DATA, CMS_PKCS9_AUTH_ENVELOPED_DATA, or CMS_PKCS9_COMPRESSED_DATA
outputBuf	output	The AED output
outputLen	output	Byte length of the output
flags	input	Creation Control Flags. See **Creation Control Flags** section below for more info

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_ARG_FAIL	Failure. Unsupported input parameters
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_PLATFORM_FAIL	Failure. One of the crypto algorithms failed
PS_SUCCESS	Success.

This is the atomic Authenticated-Enveloped-Data Content Type creation function.

The `psX509Cert_t *myCert` and `*recipientCert` parameters will have been obtained using the Matrix crypto API `psX509ParseCertFile` or `psX509ParseCert`. When parsed with these functions the recipient certificate parse function MUST be called with a flags value of `CERT_STORE_DN_BUFFER` to store the needed encoded portions of the certificate that are required by CMS. The originator certificate (`myCert`) parse MUST be called with a flags value of `CERT_STORE_UNPARSED_BUFFER | CERT_STORE_DN_BUFFER` to keep the needed encoded portions of the certificate that are required by CMS.

The `psPubKey_t *privKey` parameter is the static ECDH key agreement private key associated with `myCert` and will have been obtained using `psEcdsaParsePrivKey`.

The `keyMethod` parameter must be `MCMS_AED_KEY_AGREE_METHOD`.

The `encryptMethod` parameter must be one of: `MCMS_AES128_GCM`, `MCMS_AES192_GCM`, `MCMS_AES256_GCM`, `MCMS_AES128_CBC_CMAC`, `MCMS_AES192_CBC_CMAC`, or `MCMS_AES256_CBC_CMAC`

The `wrapMethod` parameter must be one of: `MCMS_AES128_WRAP`, `MCMS_AES192_WRAP`, or `MCMS_AES256_WRAP`.

The `keyAgreeScheme` parameter must be one of: `MCMS_ECKA_X963KDF_SHA256`, `MCMS_ECKA_X963KDF_SHA384`, or `MCMS_ECKA_X963KDF_SHA512`

The `contentType` should identify the data that is being encrypted and this OID value will be written as the eContentType member of the authEncyprtedContentInfo encoding. If the data being encrypted is a generic blob use the `CMS_PKCS7_DATA` identifier. Otherwise, choose the `CMS_PKCS7_SIGNED_DATA`, `CMS_PKCS9_AUTH_ENVELOPED_DATA`, or `CMS_PKCS9_COMPRESSED_DATA` if the data is itself a CMS data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

Creation Control Flags

The `flags` parameter controls the options on which key agreement method is used, how the originator and recipient are identified within the Authenticated-Enveloped-Data, and whether the outer ContentInfo ASN.1 encoding should be included when writing the data type.

The first configuration option is to determine whether the ECKA key agreement will be static or ephemeral. Static mode is the default and does not have a flag value. If static is chosen, the originator private key will be used when generating the secret key so the `privKey` parameter must be provided. If ephemeral, a random private key is created based on the EC parameters of the supplied recipient certificate. In this case it is not necessary to include `privKey` or `myCert`, however `myCert` is encouraged to be included so there is some information on the originator in the AED. To use ephemeral mode, include the value `MCMS_FLAGS_AED_ORIG_DHE_PUBLIC_KEY` in the flags parameter.

The second configuration option is to determine how the originator and recipient will be identified in the AED. For recipients the options are between using the X.509 issuer Distinguished Name and Serial Number or the X.509 Subject Key Identifier extension. The default is IssuerAndSerialNumber and there is no flags value to identify this choice. Supplying the value `MCMS_FLAGS_AED_RECIP_SUBJECT_KEY_ID` to the flags will create the AED with the SubjectKeyIdentifier instead.

For originator identification, you have already chosen the method if you are using ephemeral key agreement. In addition to the IssuerAndSerialNumber and SubjectKeyIdentifier options a third option for OriginatorPublicKey is available. Ephemeral key agreement must use the OriginatorPublicKey to transfer the public key directly instead of referencing a X.509 certificate. If you are using static mode the two other options are available and IssuerAndSerialNumber is the default and has no flags value. Set `MCMS_FLAGS_AED_ORIG_SUBJECT_KEY_ID` to identify the originator certificate using the Subject Key Id extension instead. Therefore, It is not an allowed combination to supply `MCMS_FLAGS_AED_ORIG_DHE_PUBLIC_KEY | MCMS_FLAGS_AED_ORIG_SUBJECT_KEY_ID` to flags.

The third configuration option is whether or not to include the originator X.509 certificate in the AED originatorInfo ASN.1. This is controlled by the `MCMS_FLAGS_AED_INCLUDE_CERT` flag value and can be used in any combination with other flags.

The final configuration option is to determine whether the outer ContentInfo ASN.1 header is written to the output. If the ContentInfo should be excluded, add the `MCMS_FLAGS_NO_CONTENT_INFO` flag.

The table below shows some viable combinations of flags for creating Authenticated-Enveloped-Data types.

AED flag combinations with relationships to `myCert` and `privKey` parameters

Representative Flag Combinations	Interpretation	myCert or privKey Required
0	Key agreement will be static so private key will be required. OriginatorIdentifierOrKey will be IssuerAndSerialNumber and will require originator certificate. Originator certificate is not included in OriginatorInfo KeyAgreeRecipientIdentifier will be IssuerAndSerialNumber.	myCert privKey
MCMS_FLAGS_AED_ORIG_DHE_PUBLIC_KEY | MCMS_FLAGS_AED_INCLUDE_CERT	Key agreement will be ephemeral so no private key will be required. OriginatorIdentifierOrKey will be OriginatorPublicKey so originator certificate will not be required. Originator certificate is included in OriginatorInfo so will be required KeyAgreeRecipientIdentifier will be IssuerAndSerialNumber.	myCert
MCMS_FLAGS_AED_ORIG_DHE_PUBLIC_KEY | MCMS_FLAGS_AED_RECIP_SUBJECT_KEY_ID	Key agreement will be ephemeral so no private key will be required. OriginatorIdentifierOrKey will be OriginatorPublicKey so originator certificate will not be required. Originator certificate is not included in OriginatorInfo KeyAgreeRecipientIdentifier will be SubjectKeyId.	

MCMS_FLAGS_AED_RECIP_SUBJECT_KEY_ID	Key agreement will be static so private key will be required. OriginatorIdentifierOrKey will be IssuerAndSerialNumber and will require originator certificate. Originator certificate is not included in OriginatorInfo KeyAgreeRecipientIdentifier will be SubjectKeyId.	myCert privKey
MCMS_FLAGS_AED_ORIG_DHE_PUBLIC_KEY (This mode will result in an anonymous originator AED)	Key agreement will be ephemeral so no private key will be required. OriginatorIdentifierOrKey will be OriginatorPublicKey so originator certificate will not be permitted. Originator certificate is not included in OriginatorInfo KeyAgreeRecipientIdentifier will be IssuerAndSerialNumber.	
MCMS_FLAGS_AED_INCLUDE_CERT	Key agreement will be static so private key will be required. OriginatorIdentifierOrKey will be IssuerAndSerialNumber and will require originator certificate. Originator certificate is included in OriginatorInfo KeyAgreeRecipientIdentifier will be IssuerAndSerialNumber.	myCert privKey
MCMS_FLAGS_AED_INCLUDE_CERT | MCMS_FLAGS_AED_ORIG_SUBJECT_KEY_ID | MCMS_FLAGS_AED_RECIP_SUBJECT_KEY_ID	Key agreement will be static so private key will be required. OriginatorIdentifierOrKey will be SubjectKeyId and will require originator certificate. Originator certificate is included in OriginatorInfo KeyAgreeRecipientIdentifier will be SubjectKeyId.	myCert privKey

3.3 matrixCmsInitCreateAuthEnvData

```
int32 matrixCmsInitCreateAuthEnvData(psPool_t *pool,
                              const psX509Cert_t *myCert,
                              const psPubKey_t *privKey,
                              const psX509Cert_t *recipientCert,
                              const int32 keyMethod,
                              const int32 encryptMethod,
                              const int32 wrapMethod,
                              const int32 keyAgreeScheme,
                              const int32 contentType,
                              unsigned char **outputBuf,
                              int32 *outputLen,
                              const int32 flags,
                              cmsAuthEnvelopedData_t **aedCtx);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
myCert	input	Optional. The originator certificate. Must be included if attaching certificate. See the table in `matrixCmsCreateAuthEnvData` above for creation control flag information.
privKey	input	Optional. The private key or the originator used the for key agreement algorithm. May be omitted if using ephemeral keys. See the table in `matrixCmsCreateAuthEnvData` above for creation control flag information.
recipientCert	input	Required. The certificate of the receiving entity
keyMethod	input	MCMS_AED_KEY_AGREE_METHOD
encryptMethod	input	The authenticated encryption algorithm. See the descriptive text in the API for `matrixCmsCreateAuthEnvData` for details.
wrapMethod	input	The AES key wrap algorithm. See the descriptive text in the API for `matrixCmsCreateAuthEnvData` for details.
keyAgreeScheme	input	The ECDH key agreement scheme. See the descriptive text in the API for `matrixCmsCreateAuthEnvData` for details.
contentType	input	The OID type of data that is being signed. Must be CMS_PKCS7_DATA, CMS_PKCS7_SIGNED_DATA, CMS_PKCS9_AUTH_ENVELOPED_DATA, or CMS_PKCS9_COMPRESSED_DATA
outputBuf	output	The AED output
outputLen	output	Byte length of the output
flags	input	Creation flags. See `matrixCmsCreateAuthEnvData` API documentation for **Creation Control Flags** for information.
aedCtx	output	The streaming context for calls to `matrixCmsUpdateCreateAuthEnvData` and `matrixCmsFinalCreateAuthEnvData`

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_ARG_FAIL	Failure. Unsupported input parameters
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_PLATFORM_FAIL	Failure. One of the crypto algorithms failed
PS_SUCCESS	Success.

This is the initialization function for the streaming mode of Authenticated-Enveloped-Data Content Type creation.

This function will return the ASN.1 BER encoded data up to the point of the OCTET STRING for the encrypted data. Each subsequent `matrixCmsUpdateCreateAuthEnvData` function call will output a component OCTET STRING of the encrypted content data. The `matrixCmsFinalCreateAuthEnvData` function call will complete the encryption and authentication process and return the final BER encoding to complete the full data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `aedCtx` streaming context will be input to the Update/Final API calls and must be freed with `matrixCmsFreeStreamCreatedAuthEnvData` when no longer needed.

Please see the information in the API `matrixCmsCreateAuthEnvData` for details on the parameters to this function.

3.4 matrixCmsUpdateCreateAuthEnvData

```
int32 matrixCmsUpdateAuthEnvData(psPool_t *pool,
                                 cmsAuthEnvelopedData_t *aedCtx,
```

```
                                   const unsigned char *dataIn,
                                   const int32 dataInLen,
                                   unsigned char **out,
                                   int32 *outLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
aedCtx	input/output	The context from a previous call to `matrixCmsInitCreateAuthEnvData`
content	input	The next portion of the data that will be encrypted
contentLen	input	The byte length of content
outputBuf	output	The next portion of the output BER encoded data
outputLen	output	The byte length of outputBuf

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_FAILURE	Failure. An internal crypto process failed
PS_SUCCESS	Success.

Continues the streaming creation of an Authenticated-Enveloped-Data Content Type.

On success, the `outputBuf` will contain the BER encoded OCTET STRING of the encrypted content that should be appended to the output of a previous call to `matrixCmsUpdateCreateAuthEnvData` (or `matrixCmsInitCreateAuthEnvData` if this is the first portion). Each subsequent `matrixCmsUpdateCreateAuthEnvData` function call will output a component OCTET STRING of the content data. The `matrixCmsFinalCreateAuthEnvData` function call will complete the encryption and authentication process and return the final BER encoding to complete the full data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `aedCtx` context must be freed with `matrixCmsFreeStreamCreatedAuthEnvData` when no longer needed.

3.5 matrixCmsFinalCreateAuthEnvData

```
int32 matrixCmsFinalCreateAuthEnvData(psPool_t *pool,
                                   cmsAuthEnvelopedData_t *aedCtx,
                                   unsigned char **out,
                                   int32 *outLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
aedCtx	input/output	The context from a previous call to `matrixCmsInitCreateAuthEnvData`
outputBuf	output	The next portion of the BER encoded type
outputLen	output	The byte length of outputBuf

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_FAILURE	Failure. An internal crypto process failed
PS_SUCCESS	Success.

Finishes the streaming creation of an Authenticated-Enveloped-Data Content Type.

On success, the `outputBuf` will contain the remainder of the BER encoded Authenticated-Enveloped-Data type.

The `outputBuf` data must be freed using `psFree` when no longer needed.

The `aedCtx` context must be freed with `matrixCmsFreeStreamCreatedAuthEnvData` when no longer needed.

3.6 matrixCmsFreeStreamCreatedAuthEnvData

```
void matrixCmsFreeStreamCreatedAuthEnvData(cmsAuthEnvelopedData_t *aedCtx);
```

Parameter	Input/Output	Description
aedCtx	input	The context from a previous call to `matrixCmsInitCreateAuthEnvData`

Frees the AED stream creation data structure.

3.7 AED Parsing

There are two available mechanisms to parse an AED type.

The first is the atomic version in which the entire encrypted envelope type is supplied to a single parsing API `matrixCmsParseAuthEnvData`. In this atomic version, the caller must also provide its EC private key and possibly the X.509 originator certificate at the time of the API call. So the use case must involve a known recipient and originator, as there will be no opportunity for the caller to identify the recipient or originator during the parse.

The second is a streaming version that uses an Init/PostInit/Update/Final flow to parse the data type. The APIs for this method are `matrixCmsInitParseAuthEnvData`, `matrixCmsPostInitParseAuthEnvDataBuf`, `matrixCmsUpdateParseAuthEnvData`, and `matrixCmsFinalParseAuthEnvData`. The PostInit phase allows the caller to locate the recipient private key and originator X.509 certificate to allow more flexible use cases.

3.7.1 Stream parsing AED and AuthAttributes

Authenticated attributes are plaintext components that are included in the CMAC or GCM algorithms when calculating the MAC of an AED type. These AuthAttributes are defined in RFC 5083 to follow the EncryptedContentInfo in the ASN.1 format. This poses a significant stream-parsing problem for AES_GCM because AES_GCM requires the plaintext additional authenticated data to be an input to the initialization of the algorithm.

Unless you are willing to tolerate a `MCMS_AED_KEY_AGREED_BUT_AUTH_FAILED` return code from `matrixCmsFinalParseAuthEnvData`, **it is not possible to get a successful return code when stream-parsing AES_GCM based AED if AuthAttributes are included**. The data will still decrypt correctly.

The AuthAttributes are OPTIONAL in the ASN.1 definition, however, and they are disabled by default in that Matrix CMS implementation. If you wish to include the default AuthAttributes in the AED for CMAC usage you may disable the `MCMS_EMPTY_AUTH_ATTRIBS` in *matrixCmsConfig.h*.

3.8 matrixCmsParseAuthEnvData

```
int32 matrixCmsParseAuthEnvData(psPool_t *pool,
                                unsigned char *buf,
                                const uint32 bufLen,
```

```
const psX509Cert_t *originatorCert,
const psPubKey_t *privKey,
const int32 flags,
unsigned char **data,
int32 *dataLen,
cmsEncryptedEnvelope_t **authData);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
buf	input	The AED type to decrypt. Not a const type because optional insitu decryption will overwrite
bufLen	input	The byte length of buf
originatorCert	input	Optional. The expected originator certificate that was used to create the data type if it has not been provided in the AED itself
privKey	input	Required. The recipient ECDSA private key
flags	input	Supply MCMS_FLAGS_EE_OVERWRITE_CT to perform an insitu decryption that destroys the cipher text. Omit to decrypt to a dedicated buffer and preserve the cipher text. Supply MCMS_FLAGS_NO_CONTENT_INFO if the incoming AED type does not includes the ContentInfo header. Set to 0 if the full CMS data type is being parsed.
data	output	The decrypted contents output. Must be freed with psFree if flags parameter is 0
dataLen	output	The byte length of the output data
authData	output	The context structure that was created during parse. Must be freed with `matrixCmsFreeParsedAuthEnvData`

Return Value	Description
MCMS_AED_FAIL_NO_CERT	Failure. The originator cert was not provided or did not match the identity specified in the data type
MCMS_AED_FAIL_KEY_AGREE	Failure. The internal ECKA algorithm failed
MCMS_AED_FAIL_KEY_UNWRAP	Failure. The key unwrap algorithm failed.
MCMS_AED_KEY_AGREED_BUT_AUTH_FAILED	Failure. The key extraction worked but the CMAC or GCM tag did not authenticate. The decrypted content will be available in content in this return case.
MCMS_PARTIAL	Failure. The input buffer was not as large as the initial ASN.1 length identifier
PS_LIMIT_FAIL	Failure. The input buffer ran out of data before parsing could complete
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_PARSE_FAIL	Failure. The AED type could not be parsed at the ASN.1 level
PS_MEM_FAIL	Failure. An internal memory allocation failed
PS_SUCCESS	Success. The decrypted and authenticated data is available in the `data` parameter

This is the atomic parse API for an AED type. This parser does require that the caller know in advance the originator certificate so is only suitable for some use cases. In some uses cases the originator may have included its certificate in the AED itself so no `originatorCert` would be needed. If an originator cert is provided and the AED also contains one, the two will be compared for a match.

If used, the `psX509Cert_t *originatorCert` parameter will have been obtained using `psX509ParseCertFile` or `psX509ParseCert`.

The `psPubKey_t *privKey` parameter is the private key of this recipient and will have been obtained using `psEcdsaParsePrivKey`.

The output `eedData` structure allows the caller to see some of the details of the AED if desired. It must be freed with `matrixCmsFreeParsedAuthEnvData` when no longer needed. However, the decrypted contents might be held within the structure so it is important not to free the context until the `data` has been processed. The following section describes the memory usage.

Memory Profile

Using the `MCMS_FLAGS_EE_OVERWRITE_CT` as the flags parameter will perform an insitu decryption and will always require less overall memory than if the value is set to `0`. However, there are some differences in implementation based on whether the AED type had encoded its contents as one large OCTET_STRING or a constructed OCTET_STRING made up of several components. In the MatrixSSL library, an atomic creation will result in the single large OCTET_STRING format and a stream creation will result in a constructed OCTET_STRING format. When the constructed OCTET_STRING format is parsed, there will ALWAYS be a dedicated memory location allocated within the `cmsEncryptedEnvelope_t` structure to store the packed component parts. This means that `matrixCmsFreeParsedAuthEnvData` must not be called until the `data` has been used.

If the `flags` value is 0 the caller must use `psFree` to free `data` directly when no longer needed. The `buf` parameter may be freed at any time after this call and `matrixCmsFreeParsedAuthEnvData` may be called any time after this call.

The following table summarizes the relationship between the AED type and `flags` parameter:

	MCMS_FLAGS_EE_OVERWRITE_CT	**0**
Single OCTET_STRING (Atomic creation)	The decryption will happen directly in the `buf` parameter so `buf` must not be freed or invalidated until data has been used. The `authData` context may theoretically be freed immediately after this parse call but the rule should still be to call `matrixCmsFreeParsedAuthEnvData` after `data` is used because caller probably doesn't know the AED type.	The decryption is made into a dedicated output memory buffer so `buf` may be freed immediately and `matrixCmsFreeParsedAuthEnvData` may be called immediately. The output data must be freed with `psFree` when done being used.
Constructed OCTET_STRING (Streaming creation)	The decryption will be written to an allocated memory buffer inside the `authData` structure. The `buf` parameter may be freed immediately after the parse call. `matrixCmsFreeParsedAuthEnvData` must not be called until after data is used.	The decryption is made into a dedicated output memory buffer so `buf` may be freed immediately and `matrixCmsFreeParsedAuthEnvData` may be called immediately. The output data must be freed with `psFree` when done being used. This option results in three memory buffers to manage the decryption of `buf` and therefore this is the most memory inefficient option.

3.9 matrixCmsInitParseAuthEnvData

```
int32 matrixCmsInitParseAuthEnvData(psPool_t *pool,
                                   const unsigned char *buf,
                                   const uint32 bufLen,
                                   cmsEncryptedEnvelope_t **eeCtx,
                                   unsigned char **remainder,
                                   int32 *remainderLen,
                                   int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
buf	input	The first portion of an AED type to decrypt.
bufLen	input	The byte length of `buf`
eeCtx	output	On success, the context to use as input to the parse routines to follow
remainder	output	The remaining AED data from `buf` that this Init function did not process. The next call to `matrixCmsUpdateParseAuthEnvData` must begin with this remainder data
remainderLen	output	The byte length of any `remainder`
flags	input	Whether the incoming AuthEnv type includes the ContentInfo header. Set to MCMS_FLAGS_NO_CONTENT_INFO if absent. Set to 0 if the full CMS data type is being parsed.

Return Value	Description
PS_LIMIT_FAIL	Failure. The input buffer did not contain enough of the AED to complete the Init. The buffer must be appended with additional AED data and called again. The original `buf` is NOT saved within this function and must be resubmitted along with the newly appended data.
PS_UNSUPPORTED_FAIL	Failure. An unsupported algorithm was encountered
PS_PARSE_FAIL	Failure. The AED type could not be parsed at the ASN.1 level
PS_MEM_FAIL	Failure. An internal memory allocation failed
PS_SUCCESS	Success. The initialization is complete and `matrixCmsPostInitParseAuthEnvData` can now be called.

This is the initialization routine for stream parsing an AED type. This function requires that all the AED data up to the encrypted content itself be available in the `buf` parameter. The function will return `PS_LIMIT_FAIL` if this requirement is not met and the user must append additional AED data and call again.

The `eeCtx` output context will become input to the other streaming parse routines for this AED.

The `remainder` output parameter points to the `buf` location where this function stopped processing. The `remainder` must be the start of the data that is passed to the first call to `matrixCmsUpdateParseAuthEnvData` to continue the parse.

However, before Update the next step in the streaming parse is to call `matrixCmsPostInitParseAuthEnvData` to register the originator certificate and the recipient private key that will be used to perform the ECDH key agreement.

3.10 matrixCmsPostInitParseAuthEnvData

```
int32 matrixCmsPostInitParseAuthEnvData(psPool_t *pool,
                                        cmsEncryptedEnvelope_t *eeCtx,
                                        const psX509Cert_t *originatorCert,
                                        const psPubKey_t *privKey);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
eeCtx	input/output	Context from previous successful call to `matrixCmsInitParseAuthEnvData`
originatorCert	input	The X.509 certificate of the originator of the AED
privKey	input	The EC private key of this local recipient

Return Value	Description
MCMS_AED_FAIL_NO_CERT	Failure. The `originatorCert` parameter was `NULL` and there was no certificate embedded within the AED itself.
MCMS_AED_FAIL_KEY_AGREE	Failure. The ECDH key agreement function failed.
MCMS_AED_FAIL_KEY_UNWRAP	Failure. The AES unwrap function failed.
PS_UNSUPPORTED_FAIL	Failure. An unsupported crypto algorithm was encountered in the `privKey` or in the AED.
PS_SUCCESS	Success. Parsing should move to the `matrixCmsUpdateParseAuthEnvData` phase.

This PostInit phase of stream parsing allows the user to locate and load the proper key material for decrypting the AED. It is required to be called after a successful return from `matrixCmsInitParseAuthEnvData`.

The user may know the `originatorCert` and recipient `privKey` based on the use case or maybe the parser needs to locate that information from the AED itself. The `matrixCmsInitParseAuthEnvData` routine has parsed the OriginatorInfo and RecipientInfos ASN.1 of the data type, which the caller may examine to locate the key material as described in the following sections.

Locating the Originator certificate

If the AED type was created with the originator certificate embedded directly in the ASN.1 it would be found in the `originator` member of the `eeCtx`. That data type is a `psX509Cert_t` (*crypto/keyformat/x509.h*), which is a fully parsed X.509 certificate including the serial number and distinguished name that should enable the user to verify it is the expected originator. In this case where the originator cert was embedded, it is NOT NECESSARY to pass an `originatorCert` to this `matrixCmsPostInitParseAuthEnvData` API because the public key material is already available. The `originatorCert` may be set to `NULL` in this case.

If the certificate was not embedded in the AED type the `originator` member will be `NULL`. In that case, the user can look inside the `recipients` member of `eeCtx`. There are two ways an AED may identify its originator certificate; IssuerAndSerialNumber or SubjectKeyIdentifier. The choice is found in the `originatorId` where it will be one of either `MCMS_ORIGIN_ID_ISSUERDN` or `MCMS_ORIGIN_ID_KEYID`.

If `MCMS_ORIGIN_ID_ISSUERDN` the issuer distinguished name of the originator will be found in the `originatorDn` member and the serial number will be found in the `originatorSn` member with a length of `originatorSnLen`. NOTE: It is the ISSUER distinguished name in the X.509 certificate… not the SUBJECT distinguished name.

If the `originatorId` is MCMS_ORIGIN_ID_KEYID the subject key identifier of the originator will be found in the `originatorSn` member and will have a length of `originatorSnLen`.

In either case, if the certificate was not embedded in the AED type the originatorCert parameter to this `matrixCmsPostInitParseAuthEnvData` must be provided. That `psX509Cert_t *` parameter will have been obtained using `psX509ParseCertFile` or `psX509ParseCert`.

Confirming the Recipient private key

The recipient should ideally only hold one private identity key but if verification of that key to the X.509 certificate is required, the user can look in the `recipients` member of `eeCtx`. The `recipientDn` and `recipientSn` sub-members will hold the distinguished name and serial number of the intended recipient.

The `psPubKey_t*` structure for the `privKey` parameter will have been obtained from a call to `psEcdsaParsePrivFile`.

3.11 matrixCmsUpdateParseAuthEnvData

```
int32 matrixCmsUpdateParseAuthEnvData(psPool_t *pool,
                                    unsigned char *buf,
                                    uint32 bufLen,
                                    cmsEncryptedEnvelope_t *eeCtx,
                                    unsigned char **data,
                                    int32 *dataLen,
                                    uint32 dataSize,
                                    unsigned char **remainder,
                                    int32 *remainderLen);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
buf	input/output	The next portion of an AED type to decrypt.
bufLen	input	The byte length of `buf`
eeCtx	input	The context from a previously successful call to `matrixCmsInitParseAuthEnvData`
data	input/output	Decrypted content data from the AED.
dataLen	output	The byte length of the `data` output
dataSize	input	The byte length of the available memory of `data` that can be written to. Must be at least same size as `bufLen`
remainder	output	The remaining AED data from `buf` that this Update function did not process.
remainderLen	output	The byte length of any `remainder`

Return Value	Description
PS_SUCCESS	Success. The end of the contents has been found and decrypted. `matrixCmsFinalParseAuthEnvData` can now be called.
MCMS_PARTIAL	Success. The update successfully completed but there is still more data expected. This function must be called again with more AED.
MCMS_UNKNOWN	Success. A corner case for AED using a block cipher mode with constructed OCTET_STRING where the parse fell on a component boundary and it can't be determined if this is the final component or there is more to follow. See discussion below for more information.
PS_LIMIT_FAIL	Success. A rare corner case for AED using constructed OCTET_STRING contents where 3 or less bytes are passed in `bufLen` and they fall right on the ASN.1 parse of the OCTET_STRING component. Append more data and recall.
PS_PARSE_FAIL	Failure. The AED type could not be parsed at the ASN.1 level
PS_ARG_FAIL	Failure. The `dataSize` parameter must be at least as large as `bufLen`. Or this function has been called after PS_SUCCESS has already been returned.

This Update phase of stream parsing is used to decrypt the actual encrypted content of the AED. The decrypted data is returned in the `data` and `dataLen` parameters.

SECURITY NOTE: The outgoing decrypted data has not been authenticated. The MAC authentication occurs during the `matrixCmsFinalParseAuthEnvData` call. In theory, the decrypted data should not be used until authenticated.

The output `data` location MAY point to the same location as in the incoming encrypted `buf` if an in-situ decryption is desired. In other words, the decrypted data will overwrite the encrypted data to save on memory usage if the original encrypted content does not need to be saved.

If the output `data` is to be written to a different dedicated buffer, the caller is responsible for allocating (and freeing) that memory.

The dataSize parameter is used to explicitly remind the caller that the destination data buffer must be at least as large as in the incoming buf. If insitu decryption is desired it is fine to assign dataSize to be the same value as bufLen.

When MCMS_PARTIAL is returned to indicate there is more data expected for decrypting, the remainder and remainderLen parameters will identify any bytes from buf that were not processed. The reason these bytes could not be processed is because there was not enough to feed to the AES block cipher. So the caller should expect that remainderLen will always be less than the AES blocksize of 16. The next call to matrixCmsUpdateParseAuthEnvData must begin with these remainder bytes.

Constructed OCTET_STRING AED considerations

AED that was generated using stream creation uses indefinite length ASN.1 encoding which creates a couple corner cases in stream parsing. The MCMS_UNKNOWN return code is one such case. Because the overall length of the encrypted content is not known, the only way for the parser to know if more data is expected is by looking at the bytes following each component decrypt.

If the buf ends exactly on one of these component boundaries **and the symmetric cipher is block based** (currently only AES_CBC_CMAC) there are no further bytes to determine if this is the final block that must be unpadded. In this case the MCMS_UNKNOWN return code is used. When this return code is encountered the decrypted data may be the final unpadded component of the contents OR it may be a full decrypted component with more data to follow. In either case, the caller must gather more AED data and call matrixCmsUpdateParseAuthEnvData again to see what the next result is. If that next call results in more decrypted data, the previous unknown is not the final block and can be used exactly as returned. If the next call does not result in more decrypted data and the return code is PS_SUCESS, the previous unknown data was the final block and it will include the pad bytes. **The padding bytes can be removed by looking at the final byte of the data, taking the decimal value of that byte, and subtracting that number of bytes from the end.**

The PS_LIMIT_FAIL is also a potential return code if indefinite length encoding was used at AED creation. This return code was chosen to match the meaning of matrixCmsInitFinalParseAuthEnvData that indicates there was not enough data to act on. The caller must append more AED to the existing buf and call again.

3.12 matrixCmsFinalParseAuthEnvData

```
int32 matrixCmsFinalParseAuthEnvData(psPool_t *pool,
                              unsigned char *buf,
                              uint32 bufLen,
                              cmsEncryptedEnvelope_t *eeCtx);
```

Parameter	Input/Output	Description
pool	input	Optional Matrix Deterministic memory pool for allocations. NULL if unused
buf	input	The remainder of AED data after matrixCmsUpdateParseAuthEnvData returns PS_SUCESS
bufLen	input	The byte length of buf
eeCtx	input	The context from a previously successful call to matrixCmsInitParseAuthEnvData

Return Value	Description
PS_SUCCESS	Success. The AED is fully decrypted and authenticated
PS_MEM_FAIL	Failure. An internal memory allocation failed
MCMS_AED_KEY_AGREED_BUT_AUTH_FAILED	Failure. The final CMAC or GCM tag validation failed. The previously decrypted data from the matrixCmsUpdateParseAuthEnvData calls did not ultimately authenticate correctly.

PS_LIMIT_FAIL	Failure. There was not enough data to complete the AED parse. Append more AED data and call again
PS_PARSE_FAIL	Failure. The AED type could not be parsed at the ASN.1 level
PS_UNSUPPORTED_FAIL	Failure. An unknown crypto algorithm was encountered

This is the final step for stream parsing an AED.

After `PS_SUCCESS` is returned from `matrixCmsUpdateParseAuthEnvData` this function should be called with the remainder of the AED.

When finished with the AED processing, the `eeCtx` parameter must be freed with a call to `matrixCmsFreeParsedAuthEnvData`.

NOTE: If you are receiving the `MCMS_AED_KEY_AGREED_BUT_AUTH_FAILED` return code and the decrypted data looks correct please see section 4.2.1 for a discussion on AES_GCM stream parsing and AuthAttributes as a possible explanation.

3.13 matrixCmsFreeParsedAuthEnvData

```
void matrixCmsFreeParsedAuthEnvData(cmsEncryptedEnvelope_t *ee);
```

Parameter	Input/Output	Description
ee	input	The context from a previous call to `matrixCmsInitParseAuthEnvData` or `matrixCmsParseAuthEnvData`

Frees the data structure. To guarantee memory safety, call this routine after the final decrypted data has been processed.

4 COMPRESSED-DATA CONTENT TYPE API

The Compressed-Data Content Type is defined in RFC 3274. It defines a standard ASN.1 format for transporting compressed data.

MatrixCMS does not support the compression and decompression of data. The API only provides the ASN.1 wrapping and unwrapping functionality. However, zlib compression is assumed and zlib OID values will be used in the encoding.

4.1 Compressed Data Creation

There are two available mechanisms to create a Compressed-Data type. The first is the atomic version in which the entire data contents are given in a single parameter to the `matrixCmsCreateCompressedData` function.

The second is a streaming version that uses an Init/Update/Final API. The APIs for this method are `matrixCmsInitCreateCompressedData`, `matrixCmsUpdateCreateCompressedData`, and `matrixCmsFinalCreateCompressedData`. Each of these three APIs will return a portion of the full Compressed-Data Content Type to the caller who can append them in a single file (or memory buffer) or send them to the receiving entity for them to reconstruct.

4.2 matrixCmsCreateCompressedData

```
int32 matrixCmsCreateCompressedData(psPool_t *pool,
            unsigned char *compressedData, int32 compressedDataLen,
            unsigned char **outputBuf, int32 *outputLen, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
compressedData	input	The compressed content to be wrapped
compressedDataLen	input	Byte length of content
outputBuf	output	The compressed data type output
outputLen	output	Byte length of the output
flags	input	Supply MCMS_FLAGS_NO_CONTENT_INFO if the outer ContentInfo header should be excluded from the output. Use 0 to create the full CMS data type.

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.

This is the atomic Compressed-Data Content Type creation function.

The `outputBuf` data must be freed using `psFree` when no longer needed.

4.3 matrixCmsInitCreateCompressedData

```
int32 matrixCmsInitCreateCompressedData(psPool_t *pool,
            unsigned char **outputBuf, int32 *outputLen, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
outputBuf	output	The Compressed-Data output
outputLen	output	Byte length of the output
flags	input	Supply MCMS_FLAGS_NO_CONTENT_INFO if the outer ContentInfo header should be excluded from the output. Use 0 to create the full CMS data type.

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.

This is the initialization function for the streaming Compressed-Data Content Type creation. The internall allocated output buffer will be the ASN.1 data right up to the point of expected compressed data. That compressed data will be sent in subsequent calls to `matrixCmsUpdateCreateCompressedData`.

Note that there is no context to associate the streaming creation with the update and final calls. It is the caller's responsibility to give any necessary context to the creation if required.

The `outputBuf` data must be freed using `psFree` when no longer needed.

4.4 matrixCmsUpdateCreateCompressedData

```
int32 matrixCmsUpdateCreateCompressedData(psPool_t *pool,
            unsigned char *compressedData, int32 compressedDataLen,
            unsigned char **outputBuf, int32 *outputLen);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
compressedData	input	The compressed content to be wrapped
compressedDataLen	input	Byte length of content
outputBuf	output	The compressed data type output
outputLen	output	Byte length of the output

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.

This is the continuation function for the streaming Compressed-Data Content Type creation. The input `compressedData` must be zlib compressed before calling this function.

Note that there is no context to associate the streaming update with the init and final calls. It is the caller's responsibility to give any necessary context to the creation.

The internally allocated `outputBuf` data should be appended on to the output from the previous call to `matrixCmsInitCreateCompressedData`. Additionally, the `outputBuf` must be freed using `psFree` when no longer needed.

4.5 matrixCmsFinalCreateCompressedData

```
int32 matrixCmsFinalCreateCompressedData(psPool_t *pool,
```

```
        unsigned char **outputBuf, int32 *outputLen, int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
outputBuf	output	The Compressed-Data output
outputLen	output	Byte length of the output
flags	input	Supply MCMS_FLAGS_NO_CONTENT_INFO if the outer ContentInfo header should be excluded from the output. Use 0 to create the full CMS data type. Must match whatever was passed as the flags value to `matrixCmsInitCreateCompressedData`

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_SUCCESS	Success.

This is the finalize function for the streaming Compressed-Data Content Type creation. The internally allocated output buffer should be appended to the output from the previous calls to `matrixCmsUpdateCreateCompressedData`.

Note that there is no context to associate the streaming finalize with the Init and Update calls. It is the caller's responsibility to give any necessary context to the creation. Therefore, this Final creation must also provide the same `flags` parameter that was passed to the `matrixCmsInitCreateCompressedData` function.

The `outputBuf` data must be freed using `psFree` when no longer needed.

4.6 Compressed Data Parsing

There are two available mechanisms to parse a CD type. The first is the atomic version in which the entire data contents are given in a single parameter to `matrixCmsParseCompressedData`.

The second is a streaming version that uses an Init/Update API. The APIs for this method are `matrixCmsInitParseCompressedData` and `matrixCmsUpdateParseCompressedData`. Note there is no Final API for stream parsing a CD type because there is no ASN.1 data that follows the data in that particular CMS data type.

4.7 matrixCmsParseCompressedData

```
int32 matrixCmsParseCompressedData(psPool_t *pool,
        unsigned char *cdBuf, uint32 cdBufLen, cmsCompressedData_t **cd,
        int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
cdBuf	input	The Compressed-Data input
cdBufLen	input	Byte length of the input
cd	output	Data structure containing the parsed information
flags	input	Supply MCMS_FLAGS_NO_CONTENT_INFO if the outer ContentInfo header is not included in the CD being parsed. Use 0 to parse the full CMS data type.

Return Value	Description
PS_MEM_FAIL	Failure. Internal memory allocation failure
PS_PARSE_FAIL	Failure. ASN.1 parse failure
PS_SUCCESS	Success. The compressed data is available in the cd structure
MCMS_PARTIAL	Success. If the ASN.1 stream is DER encoded and the passed in cdBufLen is not large enough for the initial encoded size of the Content Type, this return code will be passed back. The caller must retrieve the remainder of the data and call again. It is not possible to return this code with a BER encoded ASN.1 stream that uses indefinite-length encoding.

This is the atomic parse of a Compressed Data type. The compressed data information can be found in the `cmsCompressedData_t` structure. The compressed data itself is contained in the `compressedData` member with a length of `compressedDataLen`. The caller must inflate with zlib.

`cd` must be freed with `matrixCmsFreeCompressedData` when no longer needed.

4.8 matrixCmsInitParseCompressedData

```
int32 matrixCmsInitParseCompressedData(psPool_t *pool, unsigned char *cdBuf,
            uint32 cdBufLen, cmsCompressedData_t **cdCtx,
            unsigned char **compressedOut, int32 *compressedOutLen,
            int32 flags);
```

Parameter	Input/Output	Description
pool	input	Optional. Matrix Deterministic memory pool for allocations. NULL if unused
cdBuf	input	The initial bytes of a Compressed-Data type
cdBufLen	input	Byte length of the input
cdCtx	output	Context data structure that will be passed to `matrixCmsUpdateParseCompressedData`
compressedOut	output	Pointer to start of compressed data within the CD. NULL if data not reached
compressedOutLen	output	Byte length of `compressedOut` if not NULL
flags	input	Supply MCMS_FLAGS_NO_CONTENT_INFO if the outer ContentInfo header is not included in the CD being parsed. Use 0 to parse the full CMS data type.

Return Value	Description
PS_LIMIT_FAIL	Failure. The input buffer did not contain enough of the CD to complete the Init. The buffer must be appended with additional CD data and called again. The original `cdBuf` is NOT saved within this function and must be resubmitted along with the newly appended data.
PS_PARSE_FAIL	Failure. The CD type could not be parsed at the ASN.1 level
PS_MEM_FAIL	Failure. An internal memory allocation failed
PS_SUCCESS	Success. The initialization is complete and `matrixCmsUpdateParseCompressedData` can now be called.

This is the stream parse initialization function for Compressed Data types. If there are not enough initial bytes to reach the compressed data, this function will return PS_LIMIT_FAIL and the user must append additional CD data and call again.

If non-NULL the `compressedOut` output parameter will be a pointer directly into the supplied `cdBuf` memory. If the application requires the compressed data to remain available for other uses, it may copy the output elsewhere before inflating.

4.9 matrixCmsUpdateParseCompressedData

```
int32 matrixCmsUpdateParseCompressedData(cmsCompressedData_t *cdCtx,
            unsigned char *cdBuf, uint32 cdBufLen,
            unsigned char **compressedOut, int32 *compressedOutLen,
            unsigned char **remainder, int32 *remainderLen);
```

Parameter	Input/Output	Description
cdCtx	input	Context structure from a previous call to `matrixCmsInitParseCompressedData`
cdBuf	input	The next bytes of a Compressed-Data type
cdBufLen	input	Byte length of the input
compressedOut	output	Pointer to start of the next portion of compressed data within the CD. NULL if data not found
compressedOutLen	output	Byte length of `compressedOut` if not NULL
remainder	output	Pointer to CD data from `cdBuf` that was not parsed. Must be the start of the `cdBuf` to the next call to `matrixCmsUpdateParseCompressedData`
remainderLen	output	Byte length of `remainder` if not NULL

Return Value	Description
MCMS_PARTIAL	Success. The parse was successful and more CD bytes are expected. Gather more (or check for data in remainder) and call this function again.
PS_PARSE_FAIL	Failure. The CD type could not be parsed at the ASN.1 level
PS_SUCCESS	Success. The CD has fully completed the parse.

This is the continuation of a stream parsed CD type.

If non-NULL the `compressedOut` output parameter will be a pointer directly into the supplied `cdBuf` memory. If the application requires the compressed data to remain available for other uses, it may copy the output elsewhere before inflating.

If parsing a CD that was generated with indefinite length encoding the `remainder` and `remainderLen` output parameters may be populated to point to the next OCTET_STRING component to be parsed. If non-NULL the `remainder` output will be pointing to a location within `cdBuf`. The caller MUST make the remainder pointer the start of data passed to the next `matrixCmsUpdateParseCompressedData` call.

Note there is no Final API for CD stream parsing. The compressed data is the final ASN.1 component in a CMS Compressed Data type so there will be no further data to parse to a final routine. The return code of `PS_SUCCESS` is the indication that all compressed data has been returned to the caller.

The `cdCtx` must be freed with a call to `matrixCmdFreeCompressedData` when parsing is complete.

4.10 matrixCmsFreeCompressedData

```
void matrixCmsFreeCompressedData(cmsCompressedData_t *compressedData);
```

Parameter	Input/Output	Description
compressedData	input	Data structure created from a previous call to `matrixCmsParseCompressedData` or `matrixCmsInitParseCompressedData`

Frees the data structure.