Network Working Group
Request for Comments: 2630
Category: Standards Track

R. Housley
SPYRUS

June 1999

Cryptographic Message Syntax

Status of this Memo

Copyright Notice

Abstract

This document describes the Cryptographic Message Syntax. This syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary messages.

The Cryptographic Message Syntax is derived from PKCS #7 version 1.5 as specified in RFC 2315 [PKCS#7]. Wherever possible, backward compatibility is preserved; however, changes were necessary to accommodate attribute certificate transfer and key agreement techniques for key management.

Table of Contents

1 Introduction

This document describes the Cryptographic Message Syntax. This syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary messages.

The Cryptographic Message Syntax describes an encapsulation syntax for data protection. It supports digital signatures, message authentication codes, and encryption. The syntax allows multiple encapsulation, so one encapsulation envelope can be nested inside another. Likewise, one party can digitally sign some previously encapsulated data. It also allows arbitrary attributes, such as signing time, to be signed along with the message content, and provides for other attributes such as countersignatures to be associated with a signature.

The Cryptographic Message Syntax can support a variety of architectures for certificate-based key management, such as the one defined by the PKIX working group.

The Cryptographic Message Syntax values are generated using ASN.1 [X.208-88], using BER-encoding [X.209-88]. Values are typically represented as octet strings. While many systems are capable of transmitting arbitrary octet strings reliably, it is well known that many electronic-mail systems are not. This document does not address mechanisms for encoding octet strings for reliable transmission in such environments.

2 General Overview

The Cryptographic Message Syntax (CMS) is general enough to support many different content types. This document defines one protection content, ContentInfo. ContentInfo encapsulates a single identified content type, and the identified type may provide further encapsulation. This document defines six content types: data, signed-data, enveloped-data, digested-data, encrypted-data, and authenticated-data. Additional content types can be defined outside this document.

An implementation that conforms to this specification must implement the protection content, ContentInfo, and must implement the data, signed-data, and enveloped-data content types. The other content types may be implemented if desired.

As a general design philosophy, each content type permits single pass processing using indefinite-length Basic Encoding Rules (BER) encoding. Single-pass operation is especially helpful if content is large, stored on tapes, or is "piped" from another process. Single-

pass operation has one significant drawback: it is difficult to perform encode operations using the Distinguished Encoding Rules (DER) [X.509-88] encoding in a single pass since the lengths of the various components may not be known in advance. However, signed attributes within the signed-data content type and authenticated attributes within the authenticated-data content type require DER encoding. Signed attributes and authenticated attributes must be transmitted in DER form to ensure that recipients can verify a content that contains one or more unrecognized attributes. Signed attributes and authenticated attributes are the only CMS data types that require DER encoding.

3 General Syntax

The Cryptographic Message Syntax (CMS) associates a content type identifier with a content. The syntax shall have ASN.1 type ContentInfo:

```
ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER
```

The fields of ContentInfo have the following meanings:

contentType indicates the type of the associated content. It is an object identifier; it is a unique string of integers assigned by an authority that defines the content type.

content is the associated content. The type of content can be determined uniquely by contentType. Content types for data, signed-data, enveloped-data, digested-data, encrypted-data, and authenticated-data are defined in this document. If additional content types are defined in other documents, the ASN.1 type defined should not be a CHOICE type.

4 Data Content Type

The following object identifier identifies the data content type:

```
id-data OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 1 }
```

The data content type is intended to refer to arbitrary octet strings, such as ASCII text files; the interpretation is left to the application. Such strings need not have any internal structure

(although they could have their own ASN.1 definition or other structure).

The data content type is generally encapsulated in the signed-data, enveloped-data, digested-data, encrypted-data, or authenticated-data content type.

5 Signed-data Content Type

The signed-data content type consists of a content of any type and zero or more signature values. Any number of signers in parallel can sign any type of content.

The typical application of the signed-data content type represents one signer's digital signature on content of the data content type. Another typical application disseminates certificates and certificate revocation lists (CRLs).

The process by which signed-data is constructed involves the following steps:

1. For each signer, a message digest, or hash value, is computed on the content with a signer-specific message-digest algorithm. If the signer is signing any information other than the content, the message digest of the content and the other information are digested with the signer's message digest algorithm (see Section 5.4), and the result becomes the "message digest."

2. For each signer, the message digest is digitally signed using the signer's private key.

3. For each signer, the signature value and other signer-specific information are collected into a SignerInfo value, as defined in Section 5.3. Certificates and CRLs for each signer, and those not corresponding to any signer, are collected in this step.

4. The message digest algorithms for all the signers and the SignerInfo values for all the signers are collected together with the content into a SignedData value, as defined in Section 5.1.

A recipient independently computes the message digest. This message digest and the signer's public key are used to verify the signature value. The signer's public key is referenced either by an issuer distinguished name along with an issuer-specific serial number or by a subject key identifier that uniquely identifies the certificate containing the public key. The signer's certificate may be included in the SignedData certificates field.

This section is divided into six parts. The first part describes the top-level type SignedData, the second part describes EncapsulatedContentInfo, the third part describes the per-signer information type SignerInfo, and the fourth, fifth, and sixth parts describe the message digest calculation, signature generation, and signature verification processes, respectively.

5.1 SignedData Type

The following object identifier identifies the signed-data content type:

```
id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }
```

The signed-data content type shall have ASN.1 type SignedData:

```
SignedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithms DigestAlgorithmIdentifiers,
  encapContentInfo EncapsulatedContentInfo,
  certificates [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT CertificateRevocationLists OPTIONAL,
  signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

SignerInfos ::= SET OF SignerInfo
```

The fields of type SignedData have the following meanings:

version is the syntax version number. If no attribute certificates are present in the certificates field, the encapsulated content type is id-data, and all of the elements of SignerInfos are version 1, then the value of version shall be 1. Alternatively, if attribute certificates are present, the encapsulated content type is other than id-data, or any of the elements of SignerInfos are version 3, then the value of version shall be 3.

digestAlgorithms is a collection of message digest algorithm identifiers. There may be any number of elements in the collection, including zero. Each element identifies the message digest algorithm, along with any associated parameters, used by one or more signer. The collection is intended to list the message digest algorithms employed by all of the signers, in any order, to facilitate one-pass signature verification. The message digesting process is described in Section 5.4.

encapContentInfo is the signed content, consisting of a content type identifier and the content itself. Details of the EncapsulatedContentInfo type are discussed in section 5.2.

certificates is a collection of certificates. It is intended that the set of certificates be sufficient to contain chains from a recognized "root" or "top-level certification authority" to all of the signers in the signerInfos field. There may be more certificates than necessary, and there may be certificates sufficient to contain chains from two or more independent top-level certification authorities. There may also be fewer certificates than necessary, if it is expected that recipients have an alternate means of obtaining necessary certificates (e.g., from a previous set of certificates). As discussed above, if attribute certificates are present, then the value of version shall be 3.

crls is a collection of certificate revocation lists (CRLs). It is intended that the set contain information sufficient to determine whether or not the certificates in the certificates field are valid, but such correspondence is not necessary. There may be more CRLs than necessary, and there may also be fewer CRLs than necessary.

signerInfos is a collection of per-signer information. There may be any number of elements in the collection, including zero. The details of the SignerInfo type are discussed in section 5.3.

5.2 EncapsulatedContentInfo Type

The content is represented in the type EncapsulatedContentInfo:

```
EncapsulatedContentInfo ::= SEQUENCE {
  eContentType ContentType,
  eContent [0] EXPLICIT OCTET STRING OPTIONAL }

ContentType ::= OBJECT IDENTIFIER
```

The fields of type EncapsulatedContentInfo have the following meanings:

eContentType is an object identifier that uniquely specifies the content type.

eContent is the content itself, carried as an octet string. The eContent need not be DER encoded.

The optional omission of the eContent within the EncapsulatedContentInfo field makes it possible to construct "external signatures." In the case of external signatures, the content being signed is absent from the EncapsulatedContentInfo value included in the signed-data content type. If the eContent value within EncapsulatedContentInfo is absent, then the signatureValue is calculated and the eContentType is assigned as though the eContent value was present.

In the degenerate case where there are no signers, the EncapsulatedContentInfo value being "signed" is irrelevant. In this case, the content type within the EncapsulatedContentInfo value being "signed" should be id-data (as defined in section 4), and the content field of the EncapsulatedContentInfo value should be omitted.

5.3 SignerInfo Type

Per-signer information is represented in the type SignerInfo:

```
SignerInfo ::= SEQUENCE {
  version CMSVersion,
  sid SignerIdentifier,
  digestAlgorithm DigestAlgorithmIdentifier,
  signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature SignatureValue,
  unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }

SignerIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier }

SignedAttributes ::= SET SIZE (1..MAX) OF Attribute

UnsignedAttributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {
  attrType OBJECT IDENTIFIER,
  attrValues SET OF AttributeValue }

AttributeValue ::= ANY

SignatureValue ::= OCTET STRING
```

The fields of type SignerInfo have the following meanings:

version is the syntax version number. If the SignerIdentifier is the CHOICE issuerAndSerialNumber, then the version shall be 1. If

the SignerIdentifier is subjectKeyIdentifier, then the version shall be 3.

sid specifies the signer's certificate (and thereby the signer's public key). The signer's public key is needed by the recipient to verify the signature. SignerIdentifier provides two alternatives for specifying the signer's public key. The issuerAndSerialNumber alternative identifies the signer's certificate by the issuer's distinguished name and the certificate serial number; the subjectKeyIdentifier identifies the signer's certificate by the X.509 subjectKeyIdentifier extension value.

digestAlgorithm identifies the message digest algorithm, and any associated parameters, used by the signer. The message digest is computed on either the content being signed or the content together with the signed attributes using the process described in section 5.4. The message digest algorithm should be among those listed in the digestAlgorithms field of the associated SignerData.

signedAttributes is a collection of attributes that are signed. The field is optional, but it must be present if the content type of the EncapsulatedContentInfo value being signed is not id-data. Each SignedAttribute in the SET must be DER encoded. Useful attribute types, such as signing time, are defined in Section 11. If the field is present, it must contain, at a minimum, the following two attributes:

A content-type attribute having as its value the content type of the EncapsulatedContentInfo value being signed. Section 11.1 defines the content-type attribute. The content-type attribute is not required when used as part of a countersignature unsigned attribute as defined in section 11.4.

A message-digest attribute, having as its value the message digest of the content. Section 11.2 defines the message-digest attribute.

signatureAlgorithm identifies the signature algorithm, and any associated parameters, used by the signer to generate the digital signature.

signature is the result of digital signature generation, using the message digest and the signer's private key.

unsignedAttributes is a collection of attributes that are not signed. The field is optional. Useful attribute types, such as countersignatures, are defined in Section 11.

The fields of type SignedAttribute and UnsignedAttribute have the following meanings:

attrType indicates the type of attribute. It is an object identifier.

attrValues is a set of values that comprise the attribute. The type of each value in the set can be determined uniquely by attrType.

5.4 Message Digest Calculation Process

The message digest calculation process computes a message digest on either the content being signed or the content together with the signed attributes. In either case, the initial input to the message digest calculation process is the "value" of the encapsulated content being signed. Specifically, the initial input is the encapContentInfo eContent OCTET STRING to which the signing process is applied. Only the octets comprising the value of the eContent OCTET STRING are input to the message digest algorithm, not the tag or the length octets.

The result of the message digest calculation process depends on whether the signedAttributes field is present. When the field is absent, the result is just the message digest of the content as described above. When the field is present, however, the result is the message digest of the complete DER encoding of the SignedAttributes value contained in the signedAttributes field. Since the SignedAttributes value, when present, must contain the content type and the content message digest attributes, those values are indirectly included in the result. The content type attribute is not required when used as part of a countersignature unsigned attribute as defined in section 11.4. A separate encoding of the signedAttributes field is performed for message digest calculation. The IMPLICIT [0] tag in the signedAttributes field is not used for the DER encoding, rather an EXPLICIT SET OF tag is used. That is, the DER encoding of the SET OF tag, rather than of the IMPLICIT [0] tag, is to be included in the message digest calculation along with the length and content octets of the SignedAttributes value.

When the signedAttributes field is absent, then only the octets comprising the value of the signedData encapContentInfo eContent OCTET STRING (e.g., the contents of a file) are input to the message digest calculation. This has the advantage that the length of the content being signed need not be known in advance of the signature generation process.

Although the encapContentInfo eContent OCTET STRING tag and length octets are not included in the message digest calculation, they are still protected by other means. The length octets are protected by the nature of the message digest algorithm since it is computationally infeasible to find any two distinct messages of any length that have the same message digest.

5.5 Message Signature Generation Process

The input to the signature generation process includes the result of the message digest calculation process and the signer's private key. The details of the signature generation depend on the signature algorithm employed. The object identifier, along with any parameters, that specifies the signature algorithm employed by the signer is carried in the signatureAlgorithm field. The signature value generated by the signer is encoded as an OCTET STRING and carried in the signature field.

5.6 Message Signature Verification Process

The input to the signature verification process includes the result of the message digest calculation process and the signer's public key. The recipient may obtain the correct public key for the signer by any means, but the preferred method is from a certificate obtained from the SignedData certificates field. The selection and validation of the signer's public key may be based on certification path validation (see [PROFILE]) as well as other external context, but is beyond the scope of this document. The details of the signature verification depend on the signature algorithm employed.

The recipient may not rely on any message digest values computed by the originator. If the signedData signerInfo includes signedAttributes, then the content message digest must be calculated as described in section 5.4. For the signature to be valid, the message digest value calculated by the recipient must be the same as the value of the messageDigest attribute included in the signedAttributes of the signedData signerInfo.

6 Enveloped-data Content Type

The enveloped-data content type consists of an encrypted content of any type and encrypted content-encryption keys for one or more recipients. The combination of the encrypted content and one encrypted content-encryption key for a recipient is a "digital envelope" for that recipient. Any type of content can be enveloped for an arbitrary number of recipients using any of the three key management techniques for each recipient.

The typical application of the enveloped-data content type will represent one or more recipients' digital envelopes on content of the data or signed-data content types.

Enveloped-data is constructed by the following steps:

1. A content-encryption key for a particular content-encryption algorithm is generated at random.

2. The content-encryption key is encrypted for each recipient. The details of this encryption depend on the key management algorithm used, but three general techniques are supported:

   key transport: the content-encryption key is encrypted in the recipient's public key;

   key agreement: the recipient's public key and the sender's private key are used to generate a pairwise symmetric key, then the content-encryption key is encrypted in the pairwise symmetric key; and

   symmetric key-encryption keys: the content-encryption key is encrypted in a previously distributed symmetric key-encryption key.

3. For each recipient, the encrypted content-encryption key and other recipient-specific information are collected into a RecipientInfo value, defined in Section 6.2.

4. The content is encrypted with the content-encryption key. Content encryption may require that the content be padded to a multiple of some block size; see Section 6.3.

5. The RecipientInfo values for all the recipients are collected together with the encrypted content to form an EnvelopedData value as defined in Section 6.1.

A recipient opens the digital envelope by decrypting one of the encrypted content-encryption keys and then decrypting the encrypted content with the recovered content-encryption key.

This section is divided into four parts. The first part describes the top-level type EnvelopedData, the second part describes the per-recipient information type RecipientInfo, and the third and fourth parts describe the content-encryption and key-encryption processes.

6.1 EnvelopedData Type

The following object identifier identifies the enveloped-data content type:

```
id-envelopedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 3 }
```

The enveloped-data content type shall have ASN.1 type EnvelopedData:

```
EnvelopedData ::= SEQUENCE {
  version CMSVersion,
  originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
  recipientInfos RecipientInfos,
  encryptedContentInfo EncryptedContentInfo,
  unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }

OriginatorInfo ::= SEQUENCE {
  certs [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT CertificateRevocationLists OPTIONAL }

RecipientInfos ::= SET OF RecipientInfo

EncryptedContentInfo ::= SEQUENCE {
  contentType ContentType,
  contentEncryptionAlgorithm ContentEncryptionAlgorithmIdentifier,
  encryptedContent [0] IMPLICIT EncryptedContent OPTIONAL }

EncryptedContent ::= OCTET STRING

UnprotectedAttributes ::= SET SIZE (1..MAX) OF Attribute
```

The fields of type EnvelopedData have the following meanings:

version is the syntax version number. If originatorInfo is present, then version shall be 2. If any of the RecipientInfo structures included have a version other than 0, then the version shall be 2. If unprotectedAttrs is present, then version shall be 2. If originatorInfo is absent, all of the RecipientInfo structures are version 0, and unprotectedAttrs is absent, then version shall be 0.

originatorInfo optionally provides information about the originator. It is present only if required by the key management algorithm. It may contain certificates and CRLs:

certs is a collection of certificates. certs may contain originator certificates associated with several different key

management algorithms. certs may also contain attribute certificates associated with the originator. The certificates contained in certs are intended to be sufficient to make chains from a recognized "root" or "top-level certification authority" to all recipients. However, certs may contain more certificates than necessary, and there may be certificates sufficient to make chains from two or more independent top-level certification authorities. Alternatively, certs may contain fewer certificates than necessary, if it is expected that recipients have an alternate means of obtaining necessary certificates (e.g., from a previous set of certificates).

crls is a collection of CRLs. It is intended that the set contain information sufficient to determine whether or not the certificates in the certs field are valid, but such correspondence is not necessary. There may be more CRLs than necessary, and there may also be fewer CRLs than necessary.

recipientInfos is a collection of per-recipient information. There must be at least one element in the collection.

encryptedContentInfo is the encrypted content information.

unprotectedAttrs is a collection of attributes that are not encrypted. The field is optional. Useful attribute types are defined in Section 11.

The fields of type EncryptedContentInfo have the following meanings:

contentType indicates the type of content.

contentEncryptionAlgorithm identifies the content-encryption algorithm, and any associated parameters, used to encrypt the content. The content-encryption process is described in Section 6.3. The same content-encryption algorithm and content-encryption key is used for all recipients.

encryptedContent is the result of encrypting the content. The field is optional, and if the field is not present, its intended value must be supplied by other means.

The recipientInfos field comes before the encryptedContentInfo field so that an EnvelopedData value may be processed in a single pass.

6.2 RecipientInfo Type

Per-recipient information is represented in the type RecipientInfo. RecipientInfo has a different format for the three key management

techniques that are supported: key transport, key agreement, and previously distributed symmetric key-encryption keys. Any of the three key management techniques can be used for each recipient of the same encrypted content. In all cases, the content-encryption key is transferred to one or more recipient in encrypted form.

```
RecipientInfo ::= CHOICE {
  ktri KeyTransRecipientInfo,
  kari [1] KeyAgreeRecipientInfo,
  kekri [2] KEKRecipientInfo }

EncryptedKey ::= OCTET STRING
```

6.2.1 KeyTransRecipientInfo Type

Per-recipient information using key transport is represented in the type KeyTransRecipientInfo. Each instance of KeyTransRecipientInfo transfers the content-encryption key to one recipient.

```
KeyTransRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 0 or 2
  rid RecipientIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }

RecipientIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier }
```

The fields of type KeyTransRecipientInfo have the following meanings:

version is the syntax version number. If the RecipientIdentifier is the CHOICE issuerAndSerialNumber, then the version shall be 0. If the RecipientIdentifier is subjectKeyIdentifier, then the version shall be 2.

rid specifies the recipient's certificate or key that was used by the sender to protect the content-encryption key. The RecipientIdentifier provides two alternatives for specifying the recipient's certificate, and thereby the recipient's public key. The recipient's certificate must contain a key transport public key. The content-encryption key is encrypted with the recipient's public key. The issuerAndSerialNumber alternative identifies the recipient's certificate by the issuer's distinguished name and the certificate serial number; the subjectKeyIdentifier identifies the recipient's certificate by the X.509 subjectKeyIdentifier extension value.

keyEncryptionAlgorithm identifies the key-encryption algorithm, and any associated parameters, used to encrypt the content-encryption key for the recipient. The key-encryption process is described in Section 6.4.

encryptedKey is the result of encrypting the content-encryption key for the recipient.

6.2.2 KeyAgreeRecipientInfo Type

Recipient information using key agreement is represented in the type KeyAgreeRecipientInfo. Each instance of KeyAgreeRecipientInfo will transfer the content-encryption key to one or more recipient that uses the same key agreement algorithm and domain parameters for that algorithm.

```
KeyAgreeRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 3
  originator [0] EXPLICIT OriginatorIdentifierOrKey,
  ukm [1] EXPLICIT UserKeyingMaterial OPTIONAL,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  recipientEncryptedKeys RecipientEncryptedKeys }

OriginatorIdentifierOrKey ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier,
  originatorKey [1] OriginatorPublicKey }

OriginatorPublicKey ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  publicKey BIT STRING }

RecipientEncryptedKeys ::= SEQUENCE OF RecipientEncryptedKey

RecipientEncryptedKey ::= SEQUENCE {
  rid KeyAgreeRecipientIdentifier,
  encryptedKey EncryptedKey }

KeyAgreeRecipientIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  rKeyId [0] IMPLICIT RecipientKeyIdentifier }

RecipientKeyIdentifier ::= SEQUENCE {
  subjectKeyIdentifier SubjectKeyIdentifier,
  date GeneralizedTime OPTIONAL,
  other OtherKeyAttribute OPTIONAL }

SubjectKeyIdentifier ::= OCTET STRING
```

The fields of type KeyAgreeRecipientInfo have the following meanings:

version is the syntax version number. It shall always be 3.

originator is a CHOICE with three alternatives specifying the sender's key agreement public key. The sender uses the corresponding private key and the recipient's public key to generate a pairwise key. The content-encryption key is encrypted in the pairwise key. The issuerAndSerialNumber alternative identifies the sender's certificate, and thereby the sender's public key, by the issuer's distinguished name and the certificate serial number. The subjectKeyIdentifier alternative identifies the sender's certificate, and thereby the sender's public key, by the X.509 subjectKeyIdentifier extension value. The originatorKey alternative includes the algorithm identifier and sender's key agreement public key. Permitting originator anonymity since the public key is not certified.

ukm is optional. With some key agreement algorithms, the sender provides a User Keying Material (UKM) to ensure that a different key is generated each time the same two parties generate a pairwise key.

keyEncryptionAlgorithm identifies the key-encryption algorithm, and any associated parameters, used to encrypt the content-encryption key in the key-encryption key. The key-encryption process is described in Section 6.4.

recipientEncryptedKeys includes a recipient identifier and encrypted key for one or more recipients. The KeyAgreeRecipientIdentifier is a CHOICE with two alternatives specifying the recipient's certificate, and thereby the recipient's public key, that was used by the sender to generate a pairwise key-encryption key. The recipient's certificate must contain a key agreement public key. The content-encryption key is encrypted in the pairwise key-encryption key. The issuerAndSerialNumber alternative identifies the recipient's certificate by the issuer's distinguished name and the certificate serial number; the RecipientKeyIdentifier is described below. The encryptedKey is the result of encrypting the content-encryption key in the pairwise key-encryption key generated using the key agreement algorithm.

The fields of type RecipientKeyIdentifier have the following meanings:

subjectKeyIdentifier identifies the recipient's certificate by the X.509 subjectKeyIdentifier extension value.

date is optional. When present, the date specifies which of the recipient's previously distributed UKMs was used by the sender.

other is optional. When present, this field contains additional information used by the recipient to locate the public keying material used by the sender.

6.2.3 KEKRecipientInfo Type

Recipient information using previously distributed symmetric keys is represented in the type KEKRecipientInfo. Each instance of KEKRecipientInfo will transfer the content-encryption key to one or more recipients who have the previously distributed key-encryption key.

```
KEKRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 4
  kekid KEKIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }

KEKIdentifier ::= SEQUENCE {
  keyIdentifier OCTET STRING,
  date GeneralizedTime OPTIONAL,
  other OtherKeyAttribute OPTIONAL }
```

The fields of type KEKRecipientInfo have the following meanings:

version is the syntax version number. It shall always be 4.

kekid specifies a symmetric key-encryption key that was previously distributed to the sender and one or more recipients.

keyEncryptionAlgorithm identifies the key-encryption algorithm, and any associated parameters, used to encrypt the content-encryption key with the key-encryption key. The key-encryption process is described in Section 6.4.

encryptedKey is the result of encrypting the content-encryption key in the key-encryption key.

The fields of type KEKIdentifier have the following meanings:

keyIdentifier identifies the key-encryption key that was previously distributed to the sender and one or more recipients.

date is optional. When present, the date specifies a single key-encryption key from a set that was previously distributed.

other is optional. When present, this field contains additional information used by the recipient to determine the key-encryption key used by the sender.

6.3 Content-encryption Process

The content-encryption key for the desired content-encryption algorithm is randomly generated. The data to be protected is padded as described below, then the padded data is encrypted using the content-encryption key. The encryption operation maps an arbitrary string of octets (the data) to another string of octets (the ciphertext) under control of a content-encryption key. The encrypted data is included in the envelopedData encryptedContentInfo encryptedContent OCTET STRING.

The input to the content-encryption process is the "value" of the content being enveloped. Only the value octets of the envelopedData encryptedContentInfo encryptedContent OCTET STRING are encrypted; the OCTET STRING tag and length octets are not encrypted.

Some content-encryption algorithms assume the input length is a multiple of k octets, where k is greater than one. For such algorithms, the input shall be padded at the trailing end with k-(lth mod k) octets all having value k-(lth mod k), where lth is the length of the input. In other words, the input is padded at the trailing end with one of the following strings:

```
                   01 -- if lth mod k = k-1
                02 02 -- if lth mod k = k-2
                    .
                    .
                    .
          k k ... k k -- if lth mod k = 0
```

The padding can be removed unambiguously since all input is padded, including input values that are already a multiple of the block size, and no padding string is a suffix of another. This padding method is well defined if and only if k is less than 256.

6.4 Key-encryption Process

The input to the key-encryption process -- the value supplied to the recipient's key-encryption algorithm -- is just the "value" of the content-encryption key.

Any of the three key management techniques can be used for each recipient of the same encrypted content.

7 Digested-data Content Type

The digested-data content type consists of content of any type and a message digest of the content.

Typically, the digested-data content type is used to provide content integrity, and the result generally becomes an input to the enveloped-data content type.

The following steps construct digested-data:

1. A message digest is computed on the content with a message-digest algorithm.

2. The message-digest algorithm and the message digest are collected together with the content into a DigestedData value.

A recipient verifies the message digest by comparing the message digest to an independently computed message digest.

The following object identifier identifies the digested-data content type:

```
id-digestedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 5 }
```

The digested-data content type shall have ASN.1 type DigestedData:

```
DigestedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithm DigestAlgorithmIdentifier,
  encapContentInfo EncapsulatedContentInfo,
  digest Digest }

Digest ::= OCTET STRING
```

The fields of type DigestedData have the following meanings:

version is the syntax version number. If the encapsulated content type is id-data, then the value of version shall be 0; however, if the encapsulated content type is other than id-data, then the value of version shall be 2.

digestAlgorithm identifies the message digest algorithm, and any associated parameters, under which the content is digested. The message-digesting process is the same as in Section 5.4 in the case when there are no signed attributes.

encapContentInfo is the content that is digested, as defined in section 5.2.

digest is the result of the message-digesting process.

The ordering of the digestAlgorithm field, the encapContentInfo field, and the digest field makes it possible to process a DigestedData value in a single pass.

8 Encrypted-data Content Type

The encrypted-data content type consists of encrypted content of any type. Unlike the enveloped-data content type, the encrypted-data content type has neither recipients nor encrypted content-encryption keys. Keys must be managed by other means.

The typical application of the encrypted-data content type will be to encrypt the content of the data content type for local storage, perhaps where the encryption key is a password.

The following object identifier identifies the encrypted-data content type:

```
id-encryptedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 6 }
```

The encrypted-data content type shall have ASN.1 type EncryptedData:

```
EncryptedData ::= SEQUENCE {
  version CMSVersion,
  encryptedContentInfo EncryptedContentInfo,
  unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }
```

The fields of type EncryptedData have the following meanings:

version is the syntax version number. If unprotectedAttrs is present, then version shall be 2. If unprotectedAttrs is absent, then version shall be 0.

encryptedContentInfo is the encrypted content information, as defined in Section 6.1.

unprotectedAttrs is a collection of attributes that are not encrypted. The field is optional. Useful attribute types are defined in Section 11.

9 Authenticated-data Content Type

The authenticated-data content type consists of content of any type, a message authentication code (MAC), and encrypted authentication keys for one or more recipients. The combination of the MAC and one encrypted authentication key for a recipient is necessary for that recipient to verify the integrity of the content. Any type of content can be integrity protected for an arbitrary number of recipients.

The process by which authenticated-data is constructed involves the following steps:

1. A message-authentication key for a particular message-authentication algorithm is generated at random.

2. The message-authentication key is encrypted for each recipient. The details of this encryption depend on the key management algorithm used.

3. For each recipient, the encrypted message-authentication key and other recipient-specific information are collected into a RecipientInfo value, defined in Section 6.2.

4. Using the message-authentication key, the originator computes a MAC value on the content. If the originator is authenticating any information in addition to the content (see Section 9.2), a message digest is calculated on the content, the message digest of the content and the other information are authenticated using the message-authentication key, and the result becomes the "MAC value."

9.1 AuthenticatedData Type

The following object identifier identifies the authenticated-data content type:

```
id-ct-authData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
    ct(1) 2 }
```

The authenticated-data content type shall have ASN.1 type AuthenticatedData:

```
AuthenticatedData ::= SEQUENCE {
  version CMSVersion,
  originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
  recipientInfos RecipientInfos,
  macAlgorithm MessageAuthenticationCodeAlgorithm,
  digestAlgorithm [1] DigestAlgorithmIdentifier OPTIONAL,
  encapContentInfo EncapsulatedContentInfo,
  authenticatedAttributes [2] IMPLICIT AuthAttributes OPTIONAL,
  mac MessageAuthenticationCode,
  unauthenticatedAttributes [3] IMPLICIT UnauthAttributes OPTIONAL }

AuthAttributes ::= SET SIZE (1..MAX) OF Attribute

UnauthAttributes ::= SET SIZE (1..MAX) OF Attribute

MessageAuthenticationCode ::= OCTET STRING
```

The fields of type AuthenticatedData have the following meanings:

version is the syntax version number. It shall be 0.

originatorInfo optionally provides information about the originator. It is present only if required by the key management algorithm. It may contain certificates, attribute certificates, and CRLs, as defined in Section 6.1.

recipientInfos is a collection of per-recipient information, as defined in Section 6.1. There must be at least one element in the collection.

macAlgorithm is a message authentication code (MAC) algorithm identifier. It identifies the MAC algorithm, along with any associated parameters, used by the originator. Placement of the macAlgorithm field facilitates one-pass processing by the recipient.

digestAlgorithm identifies the message digest algorithm, and any associated parameters, used to compute a message digest on the encapsulated content if authenticated attributes are present. The message digesting process is described in Section 9.2. Placement of the digestAlgorithm field facilitates one-pass processing by the recipient. If the digestAlgorithm field is present, then the authenticatedAttributes field must also be present.

encapContentInfo is the content that is authenticated, as defined in section 5.2.

authenticatedAttributes is a collection of authenticated attributes. The authenticatedAttributes structure is optional, but it must be present if the content type of the EncapsulatedContentInfo value being authenticated is not id-data. If the authenticatedAttributes field is present, then the digestAlgorithm field must also be present. Each AuthenticatedAttribute in the SET must be DER encoded. Useful attribute types are defined in Section 11. If the authenticatedAttributes field is present, it must contain, at a minimum, the following two attributes:

A content-type attribute having as its value the content type of the EncapsulatedContentInfo value being authenticated. Section 11.1 defines the content-type attribute.

A message-digest attribute, having as its value the message digest of the content. Section 11.2 defines the message-digest attribute.

mac is the message authentication code.

unauthenticatedAttributes is a collection of attributes that are not authenticated. The field is optional. To date, no attributes have been defined for use as unauthenticated attributes, but other useful attribute types are defined in Section 11.

9.2 MAC Generation

The MAC calculation process computes a message authentication code (MAC) on either the message being authenticated or a message digest of message being authenticated together with the originator's authenticated attributes.

If authenticatedAttributes field is absent, the input to the MAC calculation process is the value of the encapContentInfo eContent OCTET STRING. Only the octets comprising the value of the eContent OCTET STRING are input to the MAC algorithm; the tag and the length octets are omitted. This has the advantage that the length of the content being authenticated need not be known in advance of the MAC generation process.

If authenticatedAttributes field is present, the content-type attribute (as described in Section 11.1) and the message-digest attribute (as described in section 11.2) must be included, and the input to the MAC calculation process is the DER encoding of

authenticatedAttributes. A separate encoding of the authenticatedAttributes field is performed for message digest calculation. The IMPLICIT [2] tag in the authenticatedAttributes field is not used for the DER encoding, rather an EXPLICIT SET OF tag is used. That is, the DER encoding of the SET OF tag, rather than of the IMPLICIT [2] tag, is to be included in the message digest calculation along with the length and content octets of the authenticatedAttributes value.

The message digest calculation process computes a message digest on the content being authenticated. The initial input to the message digest calculation process is the "value" of the encapsulated content being authenticated. Specifically, the input is the encapContentInfo eContent OCTET STRING to which the authentication process is applied. Only the octets comprising the value of the encapContentInfo eContent OCTET STRING are input to the message digest algorithm, not the tag or the length octets. This has the advantage that the length of the content being authenticated need not be known in advance. Although the encapContentInfo eContent OCTET STRING tag and length octets are not included in the message digest calculation, they are still protected by other means. The length octets are protected by the nature of the message digest algorithm since it is computationally infeasible to find any two distinct messages of any length that have the same message digest.

The input to the MAC calculation process includes the MAC input data, defined above, and an authentication key conveyed in a recipientInfo structure. The details of MAC calculation depend on the MAC algorithm employed (e.g., HMAC). The object identifier, along with any parameters, that specifies the MAC algorithm employed by the originator is carried in the macAlgorithm field. The MAC value generated by the originator is encoded as an OCTET STRING and carried in the mac field.

9.3 MAC Verification

The input to the MAC verification process includes the input data (determined based on the presence or absence of the authenticatedAttributes field, as defined in 9.2), and the authentication key conveyed in recipientInfo. The details of the MAC verification process depend on the MAC algorithm employed.

The recipient may not rely on any MAC values or message digest values computed by the originator. The content is authenticated as described in section 9.2. If the originator includes authenticated attributes, then the content of the authenticatedAttributes is authenticated as described in section 9.2. For authentication to succeed, the message MAC value calculated by the recipient must be

the same as the value of the mac field. Similarly, for authentication to succeed when the authenticatedAttributes field is present, the content message digest value calculated by the recipient must be the same as the message digest value included in the authenticatedAttributes message-digest attribute.

10 Useful Types

This section is divided into two parts. The first part defines algorithm identifiers, and the second part defines other useful types.

10.1 Algorithm Identifier Types

All of the algorithm identifiers have the same type: AlgorithmIdentifier. The definition of AlgorithmIdentifier is imported from X.509 [X.509-88].

There are many alternatives for each type of algorithm listed. For each of these five types, Section 12 lists the algorithms that must be included in a CMS implementation.

10.1.1 DigestAlgorithmIdentifier

The DigestAlgorithmIdentifier type identifies a message-digest algorithm. Examples include SHA-1, MD2, and MD5. A message-digest algorithm maps an octet string (the message) to another octet string (the message digest).

```
DigestAlgorithmIdentifier ::= AlgorithmIdentifier
```

10.1.2 SignatureAlgorithmIdentifier

The SignatureAlgorithmIdentifier type identifies a signature algorithm. Examples include DSS and RSA. A signature algorithm supports signature generation and verification operations. The signature generation operation uses the message digest and the signer's private key to generate a signature value. The signature verification operation uses the message digest and the signer's public key to determine whether or not a signature value is valid. Context determines which operation is intended.

```
SignatureAlgorithmIdentifier ::= AlgorithmIdentifier
```

10.1.3 KeyEncryptionAlgorithmIdentifier

The KeyEncryptionAlgorithmIdentifier type identifies a key-encryption algorithm used to encrypt a content-encryption key. The encryption operation maps an octet string (the key) to another octet string (the encrypted key) under control of a key-encryption key. The decryption operation is the inverse of the encryption operation. Context determines which operation is intended.

The details of encryption and decryption depend on the key management algorithm used. Key transport, key agreement, and previously distributed symmetric key-encrypting keys are supported.

```
KeyEncryptionAlgorithmIdentifier ::= AlgorithmIdentifier
```

10.1.4 ContentEncryptionAlgorithmIdentifier

The ContentEncryptionAlgorithmIdentifier type identifies a content-encryption algorithm. Examples include Triple-DES and RC2. A content-encryption algorithm supports encryption and decryption operations. The encryption operation maps an octet string (the message) to another octet string (the ciphertext) under control of a content-encryption key. The decryption operation is the inverse of the encryption operation. Context determines which operation is intended.

```
ContentEncryptionAlgorithmIdentifier ::= AlgorithmIdentifier
```

10.1.5 MessageAuthenticationCodeAlgorithm

The MessageAuthenticationCodeAlgorithm type identifies a message authentication code (MAC) algorithm. Examples include DES-MAC and HMAC. A MAC algorithm supports generation and verification operations. The MAC generation and verification operations use the same symmetric key. Context determines which operation is intended.

```
MessageAuthenticationCodeAlgorithm ::= AlgorithmIdentifier
```

10.2 Other Useful Types

This section defines types that are used other places in the document. The types are not listed in any particular order.

10.2.1 CertificateRevocationLists

The CertificateRevocationLists type gives a set of certificate revocation lists (CRLs). It is intended that the set contain information sufficient to determine whether the certificates and

attribute certificates with which the set is associated are revoked or not. However, there may be more CRLs than necessary or there may be fewer CRLs than necessary.

The CertificateList may contain a CRL, an Authority Revocation List (ARL), a Delta Revocation List, or an Attribute Certificate Revocation List. All of these lists share a common syntax.

CRLs are specified in X.509 [X.509-97], and they are profiled for use in the Internet in RFC 2459 [PROFILE].

The definition of CertificateList is imported from X.509.

```
CertificateRevocationLists ::= SET OF CertificateList
```

10.2.2 CertificateChoices

The CertificateChoices type gives either a PKCS #6 extended certificate [PKCS#6], an X.509 certificate, or an X.509 attribute certificate [X.509-97]. The PKCS #6 extended certificate is obsolete. PKCS #6 certificates are included for backward compatibility, and their use should be avoided. The Internet profile of X.509 certificates is specified in the "Internet X.509 Public Key Infrastructure: Certificate and CRL Profile" [PROFILE].

The definitions of Certificate and AttributeCertificate are imported from X.509.

```
CertificateChoices ::= CHOICE {
   certificate Certificate,                 -- See X.509
   extendedCertificate [0] IMPLICIT ExtendedCertificate,
                                            -- Obsolete
   attrCert [1] IMPLICIT AttributeCertificate }
                                            -- See X.509 and X9.57
```

10.2.3 CertificateSet

The CertificateSet type provides a set of certificates. It is intended that the set be sufficient to contain chains from a recognized "root" or "top-level certification authority" to all of the sender certificates with which the set is associated. However, there may be more certificates than necessary, or there may be fewer than necessary.

The precise meaning of a "chain" is outside the scope of this document. Some applications may impose upper limits on the length of a chain; others may enforce certain relationships between the subjects and issuers of certificates within a chain.

```
CertificateSet ::= SET OF CertificateChoices
```

10.2.4 IssuerAndSerialNumber

The IssuerAndSerialNumber type identifies a certificate, and thereby an entity and a public key, by the distinguished name of the certificate issuer and an issuer-specific certificate serial number.

The definition of Name is imported from X.501 [X.501-88], and the definition of CertificateSerialNumber is imported from X.509 [X.509-97].

```
IssuerAndSerialNumber ::= SEQUENCE {
  issuer Name,
  serialNumber CertificateSerialNumber }

CertificateSerialNumber ::= INTEGER
```

10.2.5 CMSVersion

The Version type gives a syntax version number, for compatibility with future revisions of this document.

```
CMSVersion ::= INTEGER  { v0(0), v1(1), v2(2), v3(3), v4(4) }
```

10.2.6 UserKeyingMaterial

The UserKeyingMaterial type gives a syntax for user keying material (UKM). Some key agreement algorithms require UKMs to ensure that a different key is generated each time the same two parties generate a pairwise key. The sender provides a UKM for use with a specific key agreement algorithm.

```
UserKeyingMaterial ::= OCTET STRING
```

10.2.7 OtherKeyAttribute

The OtherKeyAttribute type gives a syntax for the inclusion of other key attributes that permit the recipient to select the key used by the sender. The attribute object identifier must be registered along with the syntax of the attribute itself. Use of this structure should be avoided since it may impede interoperability.

```
OtherKeyAttribute ::= SEQUENCE {
  keyAttrId OBJECT IDENTIFIER,
  keyAttr ANY DEFINED BY keyAttrId OPTIONAL }
```

11 Useful Attributes

This section defines attributes that may be used with signed-data, enveloped-data, encrypted-data, or authenticated-data. The syntax of Attribute is compatible with X.501 [X.501-88] and RFC 2459 [PROFILE]. Some of the attributes defined in this section were originally defined in PKCS #9 [PKCS#9], others were not previously defined. The attributes are not listed in any particular order.

Additional attributes are defined in many places, notably the S/MIME Version 3 Message Specification [MSG] and the Enhanced Security Services for S/MIME [ESS], which also include recommendations on the placement of these attributes.

11.1 Content Type

The content-type attribute type specifies the content type of the ContentInfo value being signed in signed-data. The content-type attribute type is required if there are any authenticated attributes present.

The content-type attribute must be a signed attribute or an authenticated attribute; it cannot be an unsigned attribute, an unauthenticated attribute, or an unprotectedAttribute.

The following object identifier identifies the content-type attribute:

```
id-contentType OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 3 }
```

Content-type attribute values have ASN.1 type ContentType:

```
ContentType ::= OBJECT IDENTIFIER
```

A content-type attribute must have a single attribute value, even though the syntax is defined as a SET OF AttributeValue. There must not be zero or multiple instances of AttributeValue present.

The SignedAttributes and AuthAttributes syntaxes are each defined as a SET OF Attributes. The SignedAttributes in a signerInfo must not include multiple instances of the content-type attribute. Similarly, the AuthAttributes in an AuthenticatedData must not include multiple instances of the content-type attribute.

11.2 Message Digest

The message-digest attribute type specifies the message digest of the encapContentInfo eContent OCTET STRING being signed in signed-data (see section 5.4) or authenticated in authenticated-data (see section 9.2). For signed-data, the message digest is computed using the signer's message digest algorithm. For authenticated-data, the message digest is computed using the originator's message digest algorithm.

Within signed-data, the message-digest signed attribute type is required if there are any attributes present. Within authenticated-data, the message-digest authenticated attribute type is required if there are any attributes present.

The message-digest attribute must be a signed attribute or an authenticated attribute; it cannot be an unsigned attribute or an unauthenticated attribute.

The following object identifier identifies the message-digest attribute:

```
id-messageDigest OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 4 }
```

Message-digest attribute values have ASN.1 type MessageDigest:

```
MessageDigest ::= OCTET STRING
```

A message-digest attribute must have a single attribute value, even though the syntax is defined as a SET OF AttributeValue. There must not be zero or multiple instances of AttributeValue present.

The SignedAttributes syntax is defined as a SET OF Attributes. The SignedAttributes in a signerInfo must not include multiple instances of the message-digest attribute.

11.3 Signing Time

The signing-time attribute type specifies the time at which the signer (purportedly) performed the signing process. The signing-time attribute type is intended for use in signed-data.

The signing-time attribute may be a signed attribute; it cannot be an unsigned attribute, an authenticated attribute, or an unauthenticated attribute.

The following object identifier identifies the signing-time attribute:

```
id-signingTime OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 5 }
```

Signing-time attribute values have ASN.1 type SigningTime:

```
SigningTime ::= Time

Time ::= CHOICE {
  utcTime          UTCTime,
  generalizedTime  GeneralizedTime }
```

Note: The definition of Time matches the one specified in the 1997 version of X.509 [X.509-97].

Dates between 1 January 1950 and 31 December 2049 (inclusive) must be encoded as UTCTime. Any dates with year values before 1950 or after 2049 must be encoded as GeneralizedTime.

UTCTime values must be expressed in Greenwich Mean Time (Zulu) and must include seconds (i.e., times are YYMMDDHHMMSSZ), even where the number of seconds is zero. Midnight (GMT) must be represented as "YYMMDD000000Z". Century information is implicit, and the century must be determined as follows:

Where YY is greater than or equal to 50, the year shall be interpreted as 19YY; and

Where YY is less than 50, the year shall be interpreted as 20YY.

GeneralizedTime values shall be expressed in Greenwich Mean Time (Zulu) and must include seconds (i.e., times are YYYYMMDDHHMMSSZ), even where the number of seconds is zero. GeneralizedTime values must not include fractional seconds.

A signing-time attribute must have a single attribute value, even though the syntax is defined as a SET OF AttributeValue. There must not be zero or multiple instances of AttributeValue present.

The SignedAttributes syntax is defined as a SET OF Attributes. The SignedAttributes in a signerInfo must not include multiple instances of the signing-time attribute.

No requirement is imposed concerning the correctness of the signing time, and acceptance of a purported signing time is a matter of a recipient's discretion. It is expected, however, that some signers,

such as time-stamp servers, will be trusted implicitly.

11.4 Countersignature

The countersignature attribute type specifies one or more signatures on the contents octets of the DER encoding of the signatureValue field of a SignerInfo value in signed-data. Thus, the countersignature attribute type countersigns (signs in serial) another signature.

The countersignature attribute must be an unsigned attribute; it cannot be a signed attribute, an authenticated attribute, or an unauthenticated attribute.

The following object identifier identifies the countersignature attribute:

```
id-countersignature OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 6 }
```

Countersignature attribute values have ASN.1 type Countersignature:

```
Countersignature ::= SignerInfo
```

Countersignature values have the same meaning as SignerInfo values for ordinary signatures, except that:

1. The signedAttributes field must contain a message-digest attribute if it contains any other attributes, but need not contain a content-type attribute, as there is no content type for countersignatures.

2. The input to the message-digesting process is the contents octets of the DER encoding of the signatureValue field of the SignerInfo value with which the attribute is associated.

A countersignature attribute can have multiple attribute values. The syntax is defined as a SET OF AttributeValue, and there must be one or more instances of AttributeValue present.

The UnsignedAttributes syntax is defined as a SET OF Attributes. The UnsignedAttributes in a signerInfo may include multiple instances of the countersignature attribute.

A countersignature, since it has type SignerInfo, can itself contain a countersignature attribute. Thus it is possible to construct arbitrarily long series of countersignatures.

12 Supported Algorithms

This section lists the algorithms that must be implemented. Additional algorithms that should be implemented are also included.

12.1 Digest Algorithms

CMS implementations must include SHA-1. CMS implementations should include MD5.

Digest algorithm identifiers are located in the SignedData digestAlgorithms field, the SignerInfo digestAlgorithm field, the DigestedData digestAlgorithm field, and the AuthenticatedData digestAlgorithm field.

Digest values are located in the DigestedData digest field, and digest values are located in the Message Digest authenticated attribute. In addition, digest values are input to signature algorithms.

12.1.1 SHA-1

The SHA-1 digest algorithm is defined in FIPS Pub 180-1 [SHA1]. The algorithm identifier for SHA-1 is:

```
sha-1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
    oiw(14) secsig(3) algorithm(2) 26 }
```

The AlgorithmIdentifier parameters field is optional. If present, the parameters field must contain an ASN.1 NULL. Implementations should accept SHA-1 AlgorithmIdentifiers with absent parameters as well as NULL parameters. Implementations should generate SHA-1 AlgorithmIdentifiers with NULL parameters.

12.1.2 MD5

The MD5 digest algorithm is defined in RFC 1321 [MD5]. The algorithm identifier for MD5 is:

```
md5 OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) digestAlgorithm(2) 5 }
```

The AlgorithmIdentifier parameters field must be present, and the parameters field must contain NULL. Implementations may accept the MD5 AlgorithmIdentifiers with absent parameters as well as NULL parameters.

12.2 Signature Algorithms

CMS implementations must include DSA. CMS implementations may include RSA.

Signature algorithm identifiers are located in the SignerInfo signatureAlgorithm field. Also, signature algorithm identifiers are located in the SignerInfo signatureAlgorithm field of countersignature attributes.

Signature values are located in the SignerInfo signature field. Also, signature values are located in the SignerInfo signature field of countersignature attributes.

12.2.1 DSA

The DSA signature algorithm is defined in FIPS Pub 186 [DSS]. DSA is always used with the SHA-1 message digest algorithm. The algorithm identifier for DSA is:

```
id-dsa-with-sha1 OBJECT IDENTIFIER ::=  { iso(1) member-body(2)
    us(840) x9-57 (10040) x9cm(4) 3 }
```

The AlgorithmIdentifier parameters field must not be present.

12.2.2 RSA

The RSA signature algorithm is defined in RFC 2347 [NEWPKCS#1]. RFC 2347 specifies the use of the RSA signature algorithm with the SHA-1 and MD5 message digest algorithms. The algorithm identifier for RSA is:

```
rsaEncryption OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-1(1) 1 }
```

12.3 Key Management Algorithms

CMS accommodates three general key management techniques: key agreement, key transport, and previously distributed symmetric key-encryption keys.

12.3.1 Key Agreement Algorithms

CMS implementations must include key agreement using X9.42 Ephemeral-Static Diffie-Hellman.

Any symmetric encryption algorithm that a CMS implementation includes as a content-encryption algorithm must also be included as a key-

encryption algorithm. CMS implementations must include key agreement of Triple-DES pairwise key-encryption keys and Triple-DES wrapping of Triple-DES content-encryption keys. CMS implementations should include key agreement of RC2 pairwise key-encryption keys and RC2 wrapping of RC2 content-encryption keys. The key wrap algorithm for Triple-DES and RC2 is described in section 12.3.3.

A CMS implementation may support mixed key-encryption and content-encryption algorithms. For example, a 128-bit RC2 content-encryption key may be wrapped with 168-bit Triple-DES key-encryption key. Similarly, a 40-bit RC2 content-encryption key may be wrapped with 128-bit RC2 key-encryption key.

For key agreement of RC2 key-encryption keys, 128 bits must be generated as input to the key expansion process used to compute the RC2 effective key [RC2].

Key agreement algorithm identifiers are located in the EnvelopedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm and AuthenticatedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm fields.

Key wrap algorithm identifiers are located in the KeyWrapAlgorithm parameters within the EnvelopedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm and AuthenticatedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm fields.

Wrapped content-encryption keys are located in the EnvelopedData RecipientInfos KeyAgreeRecipientInfo RecipientEncryptedKeys encryptedKey field. Wrapped message-authentication keys are located in the AuthenticatedData RecipientInfos KeyAgreeRecipientInfo RecipientEncryptedKeys encryptedKey field.

12.3.1.1 X9.42 Ephemeral-Static Diffie-Hellman

Ephemeral-Static Diffie-Hellman key agreement is defined in RFC 2631 [DH-X9.42]. When using Ephemeral-Static Diffie-Hellman, the EnvelopedData RecipientInfos KeyAgreeRecipientInfo and AuthenticatedData RecipientInfos KeyAgreeRecipientInfo fields are used as follows:

version must be 3.

originator must be the originatorKey alternative. The originatorKey algorithm fields must contain the dh-public-number object identifier with absent parameters. The originatorKey publicKey field must contain the sender's ephemeral public key. The dh-public-number object identifier is:

```
dh-public-number OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) ansi-x942(10046) number-type(2) 1 }
```

ukm may be absent. When present, the ukm is used to ensure that a different key-encryption key is generated when the ephemeral private key might be used more than once.

keyEncryptionAlgorithm must be the id-alg-ESDH algorithm identifier. The algorithm identifier parameter field for id-alg-ESDH is KeyWrapAlgorihtm, and this parameter must be present. The KeyWrapAlgorithm denotes the symmetric encryption algorithm used to encrypt the content-encryption key with the pairwise key-encryption key generated using the Ephemeral-Static Diffie-Hellman key agreement algorithm. Triple-DES and RC2 key wrap algorithms are discussed in section 12.3.3. The id-alg-ESDH algorithm identifier and parameter syntax is:

```
id-alg-ESDH OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 5 }

KeyWrapAlgorithm ::= AlgorithmIdentifier
```

recipientEncryptedKeys contains an identifier and an encrypted key for each recipient. The RecipientEncryptedKey KeyAgreeRecipientIdentifier must contain either the issuerAndSerialNumber identifying the recipient's certificate or the RecipientKeyIdentifier containing the subject key identifier from the recipient's certificate. In both cases, the recipient's certificate contains the recipient's static public key. RecipientEncryptedKey EncryptedKey must contain the content-encryption key encrypted with the Ephemeral-Static Diffie-Hellman generated pairwise key-encryption key using the algorithm specified by the KeyWrapAlgortihm.

12.3.2 Key Transport Algorithms

CMS implementations should include key transport using RSA. RSA implementations must include key transport of Triple-DES content-encryption keys. RSA implementations should include key transport of RC2 content-encryption keys.

Key transport algorithm identifiers are located in the EnvelopedData RecipientInfos KeyTransRecipientInfo keyEncryptionAlgorithm and AuthenticatedData RecipientInfos KeyTransRecipientInfo keyEncryptionAlgorithm fields.

Key transport encrypted content-encryption keys are located in the EnvelopedData RecipientInfos KeyTransRecipientInfo encryptedKey

field. Key transport encrypted message-authentication keys are located in the AuthenticatedData RecipientInfos KeyTransRecipientInfo encryptedKey field.

12.3.2.1 RSA

The RSA key transport algorithm is the RSA encryption scheme defined in RFC 2313 [PKCS#1], block type is 02, where the message to be encrypted is the content-encryption key. The algorithm identifier for RSA is:

```
rsaEncryption OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-1(1) 1 }
```

The AlgorithmIdentifier parameters field must be present, and the parameters field must contain NULL.

When using a Triple-DES content-encryption key, adjust the parity bits for each DES key comprising the Triple-DES key prior to RSA encryption.

The use of RSA encryption, as defined in RFC 2313 [PKCS#1], to provide confidentiality has a known vulnerability concerns. The vulnerability is primarily relevant to usage in interactive applications rather than to store-and-forward environments. Further information and proposed countermeasures are discussed in the Security Considerations section of this document.

Note that the same encryption scheme is also defined in RFC 2437 [NEWPKCS#1]. Within RFC 2437, this scheme is called RSAES-PKCS1-v1_5.

12.3.3 Symmetric Key-Encryption Key Algorithms

CMS implementations may include symmetric key-encryption key management. Such CMS implementations must include Triple-DES key-encryption keys wrapping Triple-DES content-encryption keys, and such CMS implementations should include RC2 key-encryption keys wrapping RC2 content-encryption keys. Only 128-bit RC2 keys may be used as key-encryption keys, and they must be used with the RC2ParameterVersion parameter set to 58. A CMS implementation may support mixed key-encryption and content-encryption algorithms. For example, a 40-bit RC2 content-encryption key may be wrapped with 168-bit Triple-DES key-encryption key or with a 128-bit RC2 key-encryption key.

Key wrap algorithm identifiers are located in the EnvelopedData RecipientInfos KEKRecipientInfo keyEncryptionAlgorithm and AuthenticatedData RecipientInfos KEKRecipientInfo keyEncryptionAlgorithm fields.

Wrapped content-encryption keys are located in the EnvelopedData RecipientInfos KEKRecipientInfo encryptedKey field. Wrapped message-authentication keys are located in the AuthenticatedData RecipientInfos KEKRecipientInfo encryptedKey field.

The output of a key agreement algorithm is a key-encryption key, and this key-encryption key is used to encrypt the content-encryption key. In conjunction with key agreement algorithms, CMS implementations must include encryption of content-encryption keys with the pairwise key-encryption key generated using a key agreement algorithm. To support key agreement, key wrap algorithm identifiers are located in the KeyWrapAlgorithm parameter of the EnvelopedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm and AuthenticatedData RecipientInfos KeyAgreeRecipientInfo keyEncryptionAlgorithm fields. Wrapped content-encryption keys are located in the EnvelopedData RecipientInfos KeyAgreeRecipientInfo RecipientEncryptedKeys encryptedKey field, wrapped message-authentication keys are located in the AuthenticatedData RecipientInfos KeyAgreeRecipientInfo RecipientEncryptedKeys encryptedKey field.

12.3.3.1 Triple-DES Key Wrap

Triple-DES key encryption has the algorithm identifier:

```
id-alg-CMS3DESwrap OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 6 }
```

The AlgorithmIdentifier parameter field must be NULL.

The key wrap algorithm used to encrypt a Triple-DES content-encryption key with a Triple-DES key-encryption key is specified in section 12.6.

Out-of-band distribution of the Triple-DES key-encryption key used to encrypt the Triple-DES content-encryption key is beyond of the scope of this document.

12.3.3.2 RC2 Key Wrap

RC2 key encryption has the algorithm identifier:

```
id-alg-CMSRC2wrap OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 7 }
```

The AlgorithmIdentifier parameter field must be RC2wrapParameter:

```
RC2wrapParameter ::= RC2ParameterVersion

RC2ParameterVersion ::= INTEGER
```

The RC2 effective-key-bits (key size) greater than 32 and less than 256 is encoded in the RC2ParameterVersion. For the effective-key-bits of 40, 64, and 128, the rc2ParameterVersion values are 160, 120, and 58 respectively. These values are not simply the RC2 key length. Note that the value 160 must be encoded as two octets (00 A0), because the one octet (A0) encoding represents a negative number.

Only 128-bit RC2 keys may be used as key-encryption keys, and they must be used with the RC2ParameterVersion parameter set to 58.

The key wrap algorithm used to encrypt a RC2 content-encryption key with a RC2 key-encryption key is specified in section 12.6.

Out-of-band distribution of the RC2 key-encryption key used to encrypt the RC2 content-encryption key is beyond of the scope of this document.

12.4 Content Encryption Algorithms

CMS implementations must include Triple-DES in CBC mode. CMS implementations should include RC2 in CBC mode.

Content encryption algorithms identifiers are located in the EnvelopedData EncryptedContentInfo contentEncryptionAlgorithm and the EncryptedData EncryptedContentInfo contentEncryptionAlgorithm fields.

Content encryption algorithms are used to encipher the content located in the EnvelopedData EncryptedContentInfo encryptedContent field and the EncryptedData EncryptedContentInfo encryptedContent field.

12.4.1 Triple-DES CBC

The Triple-DES algorithm is described in ANSI X9.52 [3DES]. The Triple-DES is composed from three sequential DES [DES] operations: encrypt, decrypt, and encrypt. Three-Key Triple-DES uses a different key for each DES operation. Two-Key Triple-DES uses one key for the two encrypt operations and different key for the decrypt operation. The same algorithm identifiers are used for Three-Key Triple-DES and Two-Key Triple-DES. The algorithm identifier for Triple-DES in Cipher Block Chaining (CBC) mode is:

```
des-ede3-cbc OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) encryptionAlgorithm(3) 7 }
```

The AlgorithmIdentifier parameters field must be present, and the parameters field must contain a CBCParameter:

```
CBCParameter ::= IV

IV ::= OCTET STRING  -- exactly 8 octets
```

12.4.2 RC2 CBC

The RC2 algorithm is described in RFC 2268 [RC2]. The algorithm identifier for RC2 in CBC mode is:

```
rc2-cbc OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) encryptionAlgorithm(3) 2 }
```

The AlgorithmIdentifier parameters field must be present, and the parameters field must contain a RC2CBCParameter:

```
RC2CBCParameter ::= SEQUENCE {
  rc2ParameterVersion INTEGER,
  iv OCTET STRING  }  -- exactly 8 octets
```

The RC2 effective-key-bits (key size) greater than 32 and less than 256 is encoded in the rc2ParameterVersion. For the effective-key-bits of 40, 64, and 128, the rc2ParameterVersion values are 160, 120, and 58 respectively. These values are not simply the RC2 key length. Note that the value 160 must be encoded as two octets (00 A0), since the one octet (A0) encoding represents a negative number.

12.5 Message Authentication Code Algorithms

CMS implementations that support authenticatedData must include HMAC with SHA-1.

MAC algorithm identifiers are located in the AuthenticatedData macAlgorithm field.

MAC values are located in the AuthenticatedData mac field.

12.5.1 HMAC with SHA-1

The HMAC with SHA-1 algorithm is described in RFC 2104 [HMAC]. The algorithm identifier for HMAC with SHA-1 is:

```
hMAC-SHA1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
    dod(6) internet(1) security(5) mechanisms(5) 8 1 2 }
```

The AlgorithmIdentifier parameters field must be absent.

12.6 Triple-DES and RC2 Key Wrap Algorithms

CMS implementations must include encryption of a Triple-DES content-encryption key with a Triple-DES key-encryption key using the algorithm specified in Sections 12.6.2 and 12.6.3. CMS implementations should include encryption of a RC2 content-encryption key with a RC2 key-encryption key using the algorithm specified in Sections 12.6.4 and 12.6.5. Triple-DES and RC2 content-encryption keys are encrypted in Cipher Block Chaining (CBC) mode [MODES].

Key Transport algorithms allow for the content-encryption key to be directly encrypted; however, key agreement and symmetric key-encryption key algorithms encrypt the content-encryption key with a second symmetric encryption algorithm. This section describes how the Triple-DES or RC2 content-encryption key is formatted and encrypted.

Key agreement algorithms generate a pairwise key-encryption key, and a key wrap algorithm is used to encrypt the content-encryption key with the pairwise key-encryption key. Similarly, a key wrap algorithm is used to encrypt the content-encryption key in a previously distributed key-encryption key.

The key-encryption key is generated by the key agreement algorithm or distributed out of band. For key agreement of RC2 key-encryption keys, 128 bits must be generated as input to the key expansion process used to compute the RC2 effective key [RC2].

The same algorithm identifier is used for both 2-key and 3-key Triple-DES. When the length of the content-encryption key to be wrapped is a 2-key Triple-DES key, a third key with the same value as the first key is created. Thus, all Triple-DES content-encryption keys are wrapped like 3-key Triple-DES keys.

12.6.1 Key Checksum

The CMS Checksum Algorithm is used to provide a content-encryption key integrity check value. The algorithm is:

1. Compute a 20 octet SHA-1 [SHA1] message digest on the content-encryption key.
2. Use the most significant (first) eight octets of the message digest value as the checksum value.

12.6.2 Triple-DES Key Wrap

The Triple-DES key wrap algorithm encrypts a Triple-DES content-encryption key with a Triple-DES key-encryption key. The Triple-DES key wrap algorithm is:

1. Set odd parity for each of the DES key octets comprising the content-encryption key, call the result CEK.
2. Compute an 8 octet key checksum value on CEK as described above in Section 12.6.1, call the result ICV.
3. Let CEKICV = CEK || ICV.
4. Generate 8 octets at random, call the result IV.
5. Encrypt CEKICV in CBC mode using the key-encryption key. Use the random value generated in the previous step as the initialization vector (IV). Call the ciphertext TEMP1.
6. Let TEMP2 = IV || TEMP1.
7. Reverse the order of the octets in TEMP2. That is, the most significant (first) octet is swapped with the least significant (last) octet, and so on. Call the result TEMP3.
8. Encrypt TEMP3 in CBC mode using the key-encryption key. Use an initialization vector (IV) of 0x4adda22c79e82105. The ciphertext is 40 octets long.

Note: When the same content-encryption key is wrapped in different key-encryption keys, a fresh initialization vector (IV) must be generated for each invocation of the key wrap algorithm.

12.6.3 Triple-DES Key Unwrap

The Triple-DES key unwrap algorithm decrypts a Triple-DES content-encryption key using a Triple-DES key-encryption key. The Triple-DES key unwrap algorithm is:

1. If the wrapped content-encryption key is not 40 octets, then error.
2. Decrypt the wrapped content-encryption key in CBC mode using the key-encryption key. Use an initialization vector (IV) of 0x4adda22c79e82105. Call the output TEMP3.

3. Reverse the order of the octets in TEMP3. That is, the most significant (first) octet is swapped with the least significant (last) octet, and so on. Call the result TEMP2.
4. Decompose the TEMP2 into IV and TEMP1. IV is the most significant (first) 8 octets, and TEMP1 is the least significant (last) 32 octets.
5. Decrypt TEMP1 in CBC mode using the key-encryption key. Use the IV value from the previous step as the initialization vector. Call the ciphertext CEKICV.
6. Decompose the CEKICV into CEK and ICV. CEK is the most significant (first) 24 octets, and ICV is the least significant (last) 8 octets.
7. Compute an 8 octet key checksum value on CEK as described above in Section 12.6.1. If the computed key checksum value does not match the decrypted key checksum value, ICV, then error.
8. Check for odd parity each of the DES key octets comprising CEK. If parity is incorrect, then there is an error.
9. Use CEK as the content-encryption key.

12.6.4 RC2 Key Wrap

The RC2 key wrap algorithm encrypts a RC2 content-encryption key with a RC2 key-encryption key. The RC2 key wrap algorithm is:

1. Let the content-encryption key be called CEK, and let the length of the content-encryption key in octets be called LENGTH. LENGTH is a single octet.
2. Let LCEK = LENGTH || CEK.
3. Let LCEKPAD = LCEK || PAD. If the length of LCEK is a multiple of 8, the PAD has a length of zero. If the length of LCEK is not a multiple of 8, then PAD contains the fewest number of random octets to make the length of LCEKPAD a multiple of 8.
4. Compute an 8 octet key checksum value on LCEKPAD as described above in Section 12.6.1, call the result ICV.
5. Let LCEKPADICV = LCEKPAD || ICV.
6. Generate 8 octets at random, call the result IV.
7. Encrypt LCEKPADICV in CBC mode using the key-encryption key. Use the random value generated in the previous step as the initialization vector (IV). Call the ciphertext TEMP1.
8. Let TEMP2 = IV || TEMP1.
9. Reverse the order of the octets in TEMP2. That is, the most significant (first) octet is swapped with the least significant (last) octet, and so on. Call the result TEMP3.
10. Encrypt TEMP3 in CBC mode using the key-encryption key. Use an initialization vector (IV) of 0x4adda22c79e82105.

Note: When the same content-encryption key is wrapped in different key-encryption keys, a fresh initialization vector (IV) must be generated for each invocation of the key wrap algorithm.

12.6.5 RC2 Key Unwrap

The RC2 key unwrap algorithm decrypts a RC2 content-encryption key using a RC2 key-encryption key. The RC2 key unwrap algorithm is:

1. If the wrapped content-encryption key is not a multiple of 8 octets, then error.
2. Decrypt the wrapped content-encryption key in CBC mode using the key-encryption key. Use an initialization vector (IV) of 0x4adda22c79e82105. Call the output TEMP3.
3. Reverse the order of the octets in TEMP3. That is, the most significant (first) octet is swapped with the least significant (last) octet, and so on. Call the result TEMP2.
4. Decompose the TEMP2 into IV and TEMP1. IV is the most significant (first) 8 octets, and TEMP1 is the remaining octets.
5. Decrypt TEMP1 in CBC mode using the key-encryption key. Use the IV value from the previous step as the initialization vector. Call the plaintext LCEKPADICV.
6. Decompose the LCEKPADICV into LCEKPAD, and ICV. ICV is the least significant (last) octet 8 octets. LCEKPAD is the remaining octets.
7. Compute an 8 octet key checksum value on LCEKPAD as described above in Section 12.6.1. If the computed key checksum value does not match the decrypted key checksum value, ICV, then error.
8. Decompose the LCEKPAD into LENGTH, CEK, and PAD. LENGTH is the most significant (first) octet. CEK is the following LENGTH octets. PAD is the remaining octets, if any.
9. If the length of PAD is more than 7 octets, then error.
10. Use CEK as the content-encryption key.

Appendix A: ASN.1 Module

```
CryptographicMessageSyntax
    { iso(1) member-body(2) us(840) rsadsi(113549)
      pkcs(1) pkcs-9(9) smime(16) modules(0) cms(1) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- EXPORTS All
-- The types and values defined in this module are exported for use in
-- the other ASN.1 modules.  Other applications may use them for their
-- own purposes.

IMPORTS

  -- Directory Information Framework (X.501)
        Name
           FROM InformationFramework { joint-iso-itu-t ds(5) modules(1)
                informationFramework(1) 3 }

  -- Directory Authentication Framework (X.509)
        AlgorithmIdentifier, AttributeCertificate, Certificate,
        CertificateList, CertificateSerialNumber
           FROM AuthenticationFramework { joint-iso-itu-t ds(5)
                module(1) authenticationFramework(7) 3 } ;


-- Cryptographic Message Syntax

ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER

SignedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithms DigestAlgorithmIdentifiers,
  encapContentInfo EncapsulatedContentInfo,
  certificates [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT CertificateRevocationLists OPTIONAL,
  signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

SignerInfos ::= SET OF SignerInfo
```

```
EncapsulatedContentInfo ::= SEQUENCE {
  eContentType ContentType,
  eContent [0] EXPLICIT OCTET STRING OPTIONAL }

SignerInfo ::= SEQUENCE {
  version CMSVersion,
  sid SignerIdentifier,
  digestAlgorithm DigestAlgorithmIdentifier,
  signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature SignatureValue,
  unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }

SignerIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier }

SignedAttributes ::= SET SIZE (1..MAX) OF Attribute

UnsignedAttributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {
  attrType OBJECT IDENTIFIER,
  attrValues SET OF AttributeValue }

AttributeValue ::= ANY

SignatureValue ::= OCTET STRING

EnvelopedData ::= SEQUENCE {
  version CMSVersion,
  originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
  recipientInfos RecipientInfos,
  encryptedContentInfo EncryptedContentInfo,
  unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }

OriginatorInfo ::= SEQUENCE {
  certs [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT CertificateRevocationLists OPTIONAL }

RecipientInfos ::= SET OF RecipientInfo

EncryptedContentInfo ::= SEQUENCE {
  contentType ContentType,
  contentEncryptionAlgorithm ContentEncryptionAlgorithmIdentifier,
  encryptedContent [0] IMPLICIT EncryptedContent OPTIONAL }

EncryptedContent ::= OCTET STRING
```

```
UnprotectedAttributes ::= SET SIZE (1..MAX) OF Attribute

RecipientInfo ::= CHOICE {
  ktri KeyTransRecipientInfo,
  kari [1] KeyAgreeRecipientInfo,
  kekri [2] KEKRecipientInfo }

EncryptedKey ::= OCTET STRING

KeyTransRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 0 or 2
  rid RecipientIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }

RecipientIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier }

KeyAgreeRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 3
  originator [0] EXPLICIT OriginatorIdentifierOrKey,
  ukm [1] EXPLICIT UserKeyingMaterial OPTIONAL,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  recipientEncryptedKeys RecipientEncryptedKeys }

OriginatorIdentifierOrKey ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier,
  originatorKey [1] OriginatorPublicKey }

OriginatorPublicKey ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  publicKey BIT STRING }

RecipientEncryptedKeys ::= SEQUENCE OF RecipientEncryptedKey

RecipientEncryptedKey ::= SEQUENCE {
  rid KeyAgreeRecipientIdentifier,
  encryptedKey EncryptedKey }

KeyAgreeRecipientIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  rKeyId [0] IMPLICIT RecipientKeyIdentifier }
```

```
RecipientKeyIdentifier ::= SEQUENCE {
  subjectKeyIdentifier SubjectKeyIdentifier,
  date GeneralizedTime OPTIONAL,
  other OtherKeyAttribute OPTIONAL }

SubjectKeyIdentifier ::= OCTET STRING

KEKRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 4
  kekid KEKIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }

KEKIdentifier ::= SEQUENCE {
  keyIdentifier OCTET STRING,
  date GeneralizedTime OPTIONAL,
  other OtherKeyAttribute OPTIONAL }

DigestedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithm DigestAlgorithmIdentifier,
  encapContentInfo EncapsulatedContentInfo,
  digest Digest }

Digest ::= OCTET STRING

EncryptedData ::= SEQUENCE {
  version CMSVersion,
  encryptedContentInfo EncryptedContentInfo,
  unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }

AuthenticatedData ::= SEQUENCE {
  version CMSVersion,
  originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
  recipientInfos RecipientInfos,
  macAlgorithm MessageAuthenticationCodeAlgorithm,
  digestAlgorithm [1] DigestAlgorithmIdentifier OPTIONAL,
  encapContentInfo EncapsulatedContentInfo,
  authenticatedAttributes [2] IMPLICIT AuthAttributes OPTIONAL,
  mac MessageAuthenticationCode,
  unauthenticatedAttributes [3] IMPLICIT UnauthAttributes OPTIONAL }

AuthAttributes ::= SET SIZE (1..MAX) OF Attribute

UnauthAttributes ::= SET SIZE (1..MAX) OF Attribute

MessageAuthenticationCode ::= OCTET STRING
```

```
DigestAlgorithmIdentifier ::= AlgorithmIdentifier

SignatureAlgorithmIdentifier ::= AlgorithmIdentifier

KeyEncryptionAlgorithmIdentifier ::= AlgorithmIdentifier

ContentEncryptionAlgorithmIdentifier ::= AlgorithmIdentifier

MessageAuthenticationCodeAlgorithm ::= AlgorithmIdentifier

CertificateRevocationLists ::= SET OF CertificateList

CertificateChoices ::= CHOICE {
  certificate Certificate,  -- See X.509
  extendedCertificate [0] IMPLICIT ExtendedCertificate,  -- Obsolete
  attrCert [1] IMPLICIT AttributeCertificate }  -- See X.509 & X9.57

CertificateSet ::= SET OF CertificateChoices

IssuerAndSerialNumber ::= SEQUENCE {
  issuer Name,
  serialNumber CertificateSerialNumber }

CMSVersion ::= INTEGER  { v0(0), v1(1), v2(2), v3(3), v4(4) }

UserKeyingMaterial ::= OCTET STRING

OtherKeyAttribute ::= SEQUENCE {
  keyAttrId OBJECT IDENTIFIER,
  keyAttr ANY DEFINED BY keyAttrId OPTIONAL }


-- CMS Attributes

MessageDigest ::= OCTET STRING

SigningTime  ::= Time

Time ::= CHOICE {
  utcTime UTCTime,
  generalTime GeneralizedTime }

Countersignature ::= SignerInfo
```

```
-- Algorithm Identifiers

sha-1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
    oiw(14) secsig(3) algorithm(2) 26 }

md5 OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) digestAlgorithm(2) 5 }

id-dsa-with-sha1 OBJECT IDENTIFIER ::=  { iso(1) member-body(2)
    us(840) x9-57 (10040) x9cm(4) 3 }

rsaEncryption OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-1(1) 1 }

dh-public-number OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) ansi-x942(10046) number-type(2) 1 }

id-alg-ESDH OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 5 }

id-alg-CMS3DESwrap OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 6 }

id-alg-CMSRC2wrap OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) alg(3) 7 }

des-ede3-cbc OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) encryptionAlgorithm(3) 7 }

rc2-cbc OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) encryptionAlgorithm(3) 2 }

hMAC-SHA1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
    dod(6) internet(1) security(5) mechanisms(5) 8 1 2 }


-- Algorithm Parameters

KeyWrapAlgorithm ::= AlgorithmIdentifier

RC2wrapParameter ::= RC2ParameterVersion

RC2ParameterVersion ::= INTEGER

CBCParameter ::= IV

IV ::= OCTET STRING  -- exactly 8 octets
```

```
RC2CBCParameter ::= SEQUENCE {
  rc2ParameterVersion INTEGER,
  iv OCTET STRING  }  -- exactly 8 octets


-- Content Type Object Identifiers

id-ct-contentInfo OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
    ct(1) 6 }

id-data OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 1 }

id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }

id-envelopedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 3 }

id-digestedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 5 }

id-encryptedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 6 }

id-ct-authData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
    ct(1) 2 }


-- Attribute Object Identifiers

id-contentType OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 3 }

id-messageDigest OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 4 }

id-signingTime OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 5 }

id-countersignature OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs9(9) 6 }
```

```
-- Obsolete Extended Certificate syntax from PKCS#6

ExtendedCertificate ::= SEQUENCE {
  extendedCertificateInfo ExtendedCertificateInfo,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature Signature }

ExtendedCertificateInfo ::= SEQUENCE {
  version CMSVersion,
  certificate Certificate,
  attributes UnauthAttributes }

Signature ::= BIT STRING


END -- of CryptographicMessageSyntax
```

References

3DES American National Standards Institute. ANSI X9.52-1998, Triple Data Encryption Algorithm Modes of Operation. 1998.

DES American National Standards Institute. ANSI X3.106, "American National Standard for Information Systems - Data Link Encryption". 1983.

DH-X9.42 Rescorla, E., "Diffie-Hellman Key Agreement Method", RFC 2631, June 1999.

DSS National Institute of Standards and Technology. FIPS Pub 186: Digital Signature Standard. 19 May 1994.

ESS Hoffman, P., Editor, "Enhanced Security Services for S/MIME", RFC 2634, June 1999.

HMAC Krawczyk, H., "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

MD5 Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

MODES National Institute of Standards and Technology. FIPS Pub 81: DES Modes of Operation. 2 December 1980.

MSG Ramsdell, B., Editor, "S/MIME Version 3 Message Specification", RFC 2633, June 1999.

NEWPKCS#1 Kaliski, B., "PKCS #1: RSA Encryption, Version 2.0", RFC 2347, October 1998.

PROFILE Housley, R., Ford, W., Polk, W. and D. Solo, "Internet X.509 Public Key Infrastructure: Certificate and CRL Profile", RFC 2459, January 1999.

PKCS#1 Kaliski, B., "PKCS #1: RSA Encryption, Version 1.5.", RFC 2313, March 1998.

PKCS#6 RSA Laboratories. PKCS #6: Extended-Certificate Syntax Standard, Version 1.5. November 1993.

PKCS#7 Kaliski, B., "PKCS #7: Cryptographic Message Syntax, Version 1.5.", RFC 2315, March 1998.

PKCS#9 RSA Laboratories. PKCS #9: Selected Attribute Types, Version 1.1. November 1993.

RANDOM Eastlake, D., Crocker, S. and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994.

RC2 Rivest, R., "A Description of the RC2 (r) Encryption Algorithm", RFC 2268, March 1998.

SHA1 National Institute of Standards and Technology. FIPS Pub 180-1: Secure Hash Standard. 17 April 1995.

X.208-88 CCITT. Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1). 1988.

X.209-88 CCITT. Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). 1988.

X.501-88 CCITT. Recommendation X.501: The Directory - Models. 1988.

X.509-88 CCITT. Recommendation X.509: The Directory - Authentication Framework. 1988.

X.509-97 ITU-T. Recommendation X.509: The Directory - Authentication Framework. 1997.

Security Considerations

The Cryptographic Message Syntax provides a method for digitally signing data, digesting data, encrypting data, and authenticating data.

Implementations must protect the signer's private key. Compromise of the signer's private key permits masquerade.

Implementations must protect the key management private key, the key-encryption key, and the content-encryption key. Compromise of the key management private key or the key-encryption key may result in the disclosure of all messages protected with that key. Similarly, compromise of the content-encryption key may result in disclosure of the associated encrypted content.

Implementations must protect the key management private key and the message-authentication key. Compromise of the key management private key permits masquerade of authenticated data. Similarly, compromise of the message-authentication key may result in undetectable modification of the authenticated content.

Implementations must randomly generate content-encryption keys, message-authentication keys, initialization vectors (IVs), and padding. Also, the generation of public/private key pairs relies on a random numbers. The use of inadequate pseudo-random number generators (PRNGs) to generate cryptographic keys can result in little or no security. An attacker may find it much easier to reproduce the PRNG environment that produced the keys, searching the resulting small set of possibilities, rather than brute force searching the whole key space. The generation of quality random numbers is difficult. RFC 1750 [RANDOM] offers important guidance in this area, and Appendix 3 of FIPS Pub 186 [DSS] provides one quality PRNG technique.

When using key agreement algorithms or previously distributed symmetric key-encryption keys, a key-encryption key is used to encrypt the content-encryption key. If the key-encryption and content-encryption algorithms are different, the effective security is determined by the weaker of the two algorithms. If, for example, a message content is encrypted with 168-bit Triple-DES and the Triple-DES content-encryption key is wrapped with a 40-bit RC2 key, then at most 40 bits of protection is provided. A trivial search to determine the value of the 40-bit RC2 key can recover Triple-DES key, and then the Triple-DES key can be used to decrypt the content. Therefore, implementers must ensure that key-encryption algorithms are as strong or stronger than content-encryption algorithms.

Section 12.6 specifies key wrap algorithms used to encrypt a Triple-DES [3DES] content-encryption key with a Triple-DES key-encryption key or to encrypt a RC2 [RC2] content-encryption key with a RC2 key-encryption key. The key wrap algorithms make use of CBC mode [MODES]. These key wrap algorithms have been reviewed for use with Triple and RC2. They have not been reviewed for use with other cryptographic modes or other encryption algorithms. Therefore, if a CMS implementation wishes to support ciphers in addition to Triple-DES or RC2, then additional key wrap algorithms need to be defined to support the additional ciphers.

Implementers should be aware that cryptographic algorithms become weaker with time. As new cryptoanalysis techniques are developed and computing performance improves, the work factor to break a particular cryptographic algorithm will reduce. Therefore, cryptographic algorithm implementations should be modular allowing new algorithms to be readily inserted. That is, implementers should be prepared for the set of mandatory to implement algorithms to change over time.

The countersignature unauthenticated attribute includes a digital signature that is computed on the content signature value, thus the countersigning process need not know the original signed content.

This structure permits implementation efficiency advantages; however, this structure may also permit the countersigning of an inappropriate signature value. Therefore, implementations that perform countersignatures should either verify the original signature value prior to countersigning it (this verification requires processing of the original content), or implementations should perform countersigning in a context that ensures that only appropriate signature values are countersigned.

Users of CMS, particularly those employing CMS to support interactive applications, should be aware that PKCS #1 Version 1.5 as specified in RFC 2313 [PKCS#1] is vulnerable to adaptive chosen ciphertext attacks when applied for encryption purposes. Exploitation of this identified vulnerability, revealing the result of a particular RSA decryption, requires access to an oracle which will respond to a large number of ciphertexts (based on currently available results, hundreds of thousands or more), which are constructed adaptively in response to previously-received replies providing information on the successes or failures of attempted decryption operations. As a result, the attack appears significantly less feasible to perpetrate for store-and-forward S/MIME environments than for directly interactive protocols. Where CMS constructs are applied as an intermediate encryption layer within an interactive request-response communications environment, exploitation could be more feasible.

An updated version of PKCS #1 has been published, PKCS #1 Version 2.0 [NEWPKCS#1]. This new document will supersede RFC 2313. PKCS #1 Version 2.0 preserves support for the encryption padding format defined in PKCS #1 Version 1.5 [PKCS#1], and it also defines a new alternative. To resolve the adaptive chosen ciphertext vulnerability, the PKCS #1 Version 2.0 specifies and recommends use of Optimal Asymmetric Encryption Padding (OAEP) when RSA encryption is used to provide confidentiality. Designers of protocols and systems employing CMS for interactive environments should either consider usage of OAEP, or should ensure that information which could reveal the success or failure of attempted PKCS #1 Version 1.5 decryption operations is not provided. Support for OAEP will likely be added to a future version of the CMS specification.

Acknowledgments

This document is the result of contributions from many professionals. I appreciate the hard work of all members of the IETF S/MIME Working Group. I extend a special thanks to Rich Ankney, Tim Dean, Steve Dusse, Carl Ellison, Peter Gutmann, Bob Jueneman, Stephen Henson, Paul Hoffman, Scott Hollenbeck, Don Johnson, Burt Kaliski, John Linn, John Pawling, Blake Ramsdell, Francois Rousseau, Jim Schaad, and Dave Solo for their efforts and support.

Author's Address

Russell Housley
SPYRUS
381 Elden Street
Suite 1120
Herndon, VA 20170
USA

EMail: housley@spyrus.com

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.