Network Working Group   R. Housley
Request for Comments: 3369   RSA Laboratories
Obsoletes: 2630, 3211   August 2002
Category: Standards Track

Cryptographic Message Syntax (CMS)

Status of this Memo

Copyright Notice

Abstract

This document describes the Cryptographic Message Syntax (CMS). This syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary message content.

Table of Contents

1. Introduction

This document describes the Cryptographic Message Syntax (CMS). This syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary message content.

The CMS describes an encapsulation syntax for data protection. It supports digital signatures and encryption. The syntax allows multiple encapsulations; 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 CMS can support a variety of architectures for certificate-based key management, such as the one defined by the PKIX working group [PROFILE].

The CMS 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.

The CMS 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 version 1 attribute certificate transfer, key agreement and symmetric key-encryption key techniques for key management.

1.1 Changes Since RFC 2630

This document obsoletes RFC 2630 [OLDCMS] and RFC 3211 [PWRI]. Password-based key management is included in the CMS specification, and an extension mechanism to support new key management schemes without further changes to the CMS is specified. Backward compatibility with RFC 2630 and RFC 3211 is preserved; however, version 2 attribute certificate transfer is added. The use of version 1 attribute certificates is deprecated.

S/MIME v2 signatures [OLDMSG], which are based on PKCS#7 version 1.5, are compatible with S/MIME v3 signatures [MSG], which are based on RFC 2630. However, there are some subtle compatibility issues with signatures using PKCS#7 version 1.5 and the CMS. These issues are discussed in section 5.2.1.

Specific cryptographic algorithms are not discussed in this document, but they were discussed in RFC 2630. The discussion of specific cryptographic algorithms has been moved to a separate document [CMSALG]. Separation of the protocol and algorithm specifications allows the IETF to update each document independently. This specification does not require the implementation of any particular algorithms. Rather, protocols that rely on the CMS are expected to choose appropriate algorithms for their environment. The algorithms may be selected from [CMSALG] or elsewhere.

1.2 Terminology

In this document, the key words MUST, MUST NOT, REQUIRED, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL are to be interpreted as described in [STDWORDS].

2 General Overview

The 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.

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 need to 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 data types used in the CMS that require DER encoding.

3 General Syntax

The following object identifier identifies the content information type:

```
id-ct-contentInfo OBJECT IDENTIFIER ::= { iso(1) member-body(2)
   us(840) rsadsi(113549) pkcs(1) pkcs9(9) smime(16) ct(1) 6 }
```

The CMS associates a content type identifier with a content. The syntax MUST 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).

S/MIME uses id-data to identify MIME encoded content. The use of this content identifier is specified in RFC 2311 for S/MIME v2 [OLDMSG] and RFC 2633 for S/MIME v3 [MSG].

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 can 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. The appropriate value depends on certificates, eContentType, and SignerInfo. The version MUST be assigned as follows:

```
IF (certificates is present) AND
    (any version 2 attribute certificates are present)
THEN version MUST be 4
ELSE
    IF ((certificates is present) AND
       (any version 1 attribute certificates are present)) OR
       (encapContentInfo eContentType is other than id-data) OR
       (any SignerInfo structures are version 3)
    THEN version MUST be 3
    ELSE version MUST be 1
```

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. Implementations MAY fail to validate signatures that use a digest algorithm that is not included in this set. 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). The signer’s certificate MAY be included. The use of version 1 attribute certificates is strongly discouraged.

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. Since each signer can employ a digital signature technique and future specifications could update the syntax, all implementations MUST gracefully handle unimplemented versions of SignerInfo. Further, since all implementations will not support every possible signature algorithm, all implementations MUST gracefully handle unimplemented signature algorithms when they are encountered.

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. The object identifier 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" MUST be id-data (as defined in section 4), and the content field of the EncapsulatedContentInfo value MUST be omitted.

5.2.1 Compatibility with PKCS #7

This section contains a word of warning to implementers that wish to support both the CMS and PKCS #7 [PKCS#7] SignedData content types. Both the CMS and PKCS #7 identify the type of the encapsulated content with an object identifier, but the ASN.1 type of the content itself is variable in PKCS #7 SignedData content type.

PKCS #7 defines content as:

```
content [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL
```

The CMS defines eContent as:

```
eContent [0] EXPLICIT OCTET STRING OPTIONAL
```

The CMS definition is much easier to use in most applications, and it is compatible with both S/MIME v2 and S/MIME v3. S/MIME signed messages using the CMS and PKCS #7 are compatible because identical signed message formats are specified in RFC 2311 for S/MIME v2 [OLDMSG] and RFC 2633 for S/MIME v3 [MSG]. S/MIME v2 encapsulates the MIME content in a Data type (that is, an OCTET STRING) carried in the SignedData contentInfo content ANY field, and S/MIME v3 carries the MIME content in the SignedData encapContentInfo eContent OCTET STRING. Therefore, in both S/MIME v2 and S/MIME v3, the MIME content is placed in an OCTET STRING and the message digest is computed over the identical portions of the content. That is, the message digest is computed over the octets comprising the value of the OCTET STRING, neither the tag nor length octets are included.

There are incompatibilities between the CMS and PKCS #7 signedData types when the encapsulated content is not formatted using the Data type. For example, when an RFC 2634 [ESS] signed receipt is encapsulated in the CMS signedData type, then the Receipt SEQUENCE is encoded in the signedData encapContentInfo eContent OCTET STRING and the message digest is computed using the entire Receipt SEQUENCE encoding (including tag, length and value octets). However, if an RFC 2634 signed receipt is encapsulated in the PKCS #7 signedData type, then the Receipt SEQUENCE is DER encoded [X.509-88] in the SignedData contentInfo content ANY field (a SEQUENCE, not an OCTET STRING). Therefore, the message digest is computed using only the value octets of the Receipt SEQUENCE encoding.

The following strategy can be used to achieve backward compatibility with PKCS #7 when processing SignedData content types. If the implementation is unable to ASN.1 decode the signedData type using the CMS signedData encapContentInfo eContent OCTET STRING syntax, then the implementation MAY attempt to decode the signedData type using the PKCS #7 SignedData contentInfo content ANY syntax and compute the message digest accordingly.

The following strategy can be used to achieve backward compatibility with PKCS #7 when creating a SignedData content type in which the encapsulated content is not formatted using the Data type. Implementations MAY examine the value of the eContentType, and then adjust the expected DER encoding of eContent based on the object identifier value. For example, to support Microsoft AuthentiCode, the following information MAY be included:

eContentType Object Identifier is set to { 1 3 6 1 4 1 311 2 1 4 }

eContent contains DER encoded AuthentiCode signing information

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 MUST be 1. If the SignerIdentifier is subjectKeyIdentifier, then the version MUST 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.

Implementations MUST support the reception of the issuerAndSerialNumber and subjectKeyIdentifier forms of SignerIdentifier. When generating a SignerIdentifier, implementations MAY support one of the forms (either issuerAndSerialNumber or subjectKeyIdentifier) and always use it, or implementations MAY arbitrarily mix the two forms.

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. Implementations MAY fail to validate signatures that use a digest algorithm that is not included in the SignedData digestAlgorithms set.

signedAttrs 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. SignedAttributes MUST be DER encoded, even if the rest of the structure is BER 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. However, the content-type attribute MUST NOT be 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. The details of the signature depend on the signature algorithm employed.

unsignedAttrs 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. The attrType can impose restrictions on the number of items in the set.

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 signedAttrs 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 SignedAttrs value contained in the signedAttrs field. Since the SignedAttrs value, when present, must contain the content-type and the message-digest attributes, those values are indirectly included in the result. The content-type attribute MUST NOT be included in a countersignature unsigned attribute as defined in section 11.4. A separate encoding of the signedAttrs field is performed for message digest calculation. The IMPLICIT [0] tag in the signedAttrs is not used for the DER encoding, rather an EXPLICIT SET OF tag is used. That is, the DER encoding of the EXPLICIT SET OF tag, rather than of the IMPLICIT [0] tag, MUST be included in the message digest calculation along with the length and content octets of the SignedAttributes value.

When the signedAttrs field is absent, 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 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 message contents of any length that have the same message digest.

5.5 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 MUST be encoded as an OCTET STRING and carried in the signature field.

5.6 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 MUST 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.

If the SignedData signerInfo includes signedAttributes, then the content-type attribute value MUST match the SignedData encapContentInfo eContentType value.

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 four 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;

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

   passwords: the content-encryption key is encrypted in a key-encryption key that is derived from a password or other shared secret value.

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 SIZE (1..MAX) 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. The appropriate value depends on originatorInfo, RecipientInfo, and unprotectedAttrs. The version MUST be assigned as follows:

```
IF ((originatorInfo is present) AND
    (any version 2 attribute certificates are present)) OR
   (any RecipientInfo structures include pwri) OR
   (any RecipientInfo structures include ori)
THEN version is 3
ELSE
   IF (originatorInfo is present) OR
      (unprotectedAttrs is present) OR
      (any RecipientInfo structures are a version other than 0)
   THEN version is 2
   ELSE version is 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 for all recipients to build certification paths from a recognized "root" or "top-level certification authority." However, certs may contain more certificates than necessary, and there may be certificates sufficient to make certification paths 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 are 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 each of the supported key management techniques. Any of the key management techniques can be used for each recipient of the same encrypted content. In all cases, the encrypted content-encryption key is transferred to one or more recipients.

Since all implementations will not support every possible key management algorithm, all implementations MUST gracefully handle unimplemented algorithms when they are encountered. For example, if a recipient receives a content-encryption key encrypted in their RSA public key using RSA-OAEP and the implementation only supports RSA PKCS #1 v1.5, then a graceful failure must be implemented.

Implementations MUST support key transport, key agreement, and previously distributed symmetric key-encryption keys, as represented by ktri, kari, and kekri, respectively. Implementations MAY support the password-based key management as represented by pwri. Implementations MAY support any other key management technique as represented by ori. Since each recipient can employ a different key management technique and future specifications could define additional key management techniques, all implementations MUST gracefully handle unimplemented alternatives within the RecipientInfo CHOICE, all implementations MUST gracefully handle unimplemented versions of otherwise supported alternatives within the RecipientInfo CHOICE, and all implementations MUST gracefully handle unimplemented or unknown ori alternatives.

```
RecipientInfo ::= CHOICE {
  ktri KeyTransRecipientInfo,
  kari [1] KeyAgreeRecipientInfo,
  kekri [2] KEKRecipientInfo,
  pwri [3] PasswordRecipientinfo,
  ori [4] OtherRecipientInfo }

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 MUST be 0. If the RecipientIdentifier is subjectKeyIdentifier, then the version MUST 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. Therefore, a recipient X.509 version 3 certificate that contains a key usage extension MUST assert the keyEncipherment bit. 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. For recipient processing, implementations MUST support both of these alternatives for specifying the recipient's certificate; and for sender processing, implementations MUST support at least one of these alternatives.

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 recipients that use 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 MUST 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. This alternative permits originator anonymity since the public key is not certified. Implementations MUST support all three alternatives for specifying the sender's public key.

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. Implementations MUST support recipient processing of a KeyAgreeRecipientInfo SEQUENCE that includes a ukm field. Implementations that do not support key agreement algorithms that make use of UKMs MUST gracefully handle the presence of UKMs.

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.

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. Therefore, a recipient X.509 version 3 certificate that contains a key usage extension MUST assert the keyAgreement bit. 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. Implementations MUST support both alternatives for specifying the recipient's certificate.

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 MUST 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.

6.2.4 PasswordRecipientInfo Type

Recipient information using a password or shared secret value is represented in the type PasswordRecipientInfo. Each instance of PasswordRecipientInfo will transfer the content-encryption key to one or more recipients who possess the password or shared secret value.

The PasswordRecipientInfo Type is specified in RFC 3211 [PWRI]. The PasswordRecipientInfo structure is repeated here for completeness.

```
PasswordRecipientInfo ::= SEQUENCE {
  version CMSVersion,   -- Always set to 0
  keyDerivationAlgorithm [0] KeyDerivationAlgorithmIdentifier
                               OPTIONAL,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }
```

The fields of type PasswordRecipientInfo have the following meanings:

version is the syntax version number. It MUST always be 0.

keyDerivationAlgorithm identifies the key-derivation algorithm, and any associated parameters, used to derive the key-encryption key from the password or shared secret value. If this field is absent, the key-encryption key is supplied from an external source, for example a hardware crypto token such as a smart card.

keyEncryptionAlgorithm identifies the encryption algorithm, and any associated parameters, used to encrypt the content-encryption key with the key-encryption key.

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

6.2.5 OtherRecipientInfo Type

Recipient information for additional key management techniques are represented in the type OtherRecipientInfo. The OtherRecipientInfo type allows key management techniques beyond key transport, key agreement, previously distributed symmetric key-encryption keys, and password-based key management to be specified in future documents. An object identifier uniquely identifies such key management techniques.

```
OtherRecipientInfo ::= SEQUENCE {
  oriType OBJECT IDENTIFIER,
  oriValue ANY DEFINED BY oriType }
```

The fields of type OtherRecipientInfo have the following meanings:

oriType identifies the key management technique.

oriValue contains the protocol data elements needed by a recipient using the identified key management technique.

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.

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 aforementioned 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 MUST be 0; however, if the encapsulated content type is other than id-data, then the value of version MUST 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 derived from 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 MUST be 2. If unprotectedAttrs is absent, then version MUST 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,
  authAttrs [2] IMPLICIT AuthAttributes OPTIONAL,
  mac MessageAuthenticationCode,
  unauthAttrs [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. The version MUST be assigned as follows:

```
IF ((originatorInfo is present) AND
    (any version 2 attribute certificates are present))
THEN version is 1
ELSE version is 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 authAttrs field MUST also be present.

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

authAttrs is a collection of authenticated attributes. The authAttrs structure is optional, but it MUST be present if the content type of the EncapsulatedContentInfo value being authenticated is not id-data. If the authAttrs field is present, then the digestAlgorithm field MUST also be present. The AuthAttributes structure MUST be DER encoded, even if the rest of the structure is BER encoded. Useful attribute types are defined in Section 11. If the authAttrs 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.

unauthAttrs 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 content being authenticated or a message digest of content being authenticated together with the originator's authenticated attributes.

If authAttrs 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 authAttrs 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 authAttrs. A separate encoding of the authAttrs field is performed for message digest calculation. The IMPLICIT [2] tag in the authAttrs 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 authAttrs 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 contents 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 authAttrs 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 MUST 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 authAttrs is authenticated as described in section 9.2. For authentication to succeed, the MAC value calculated by the recipient MUST be the same as the value of the mac field. Similarly, for authentication to succeed when the authAttrs field is present, the content message digest value calculated by the recipient MUST be the same as the message digest value included in the authAttrs message-digest attribute.

If the AuthenticatedData includes authAttrs, then the content-type attribute value MUST match the AuthenticatedData encapContentInfo eContentType value.

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 taken from X.509 [X.509-88].

There are many alternatives for each algorithm type.

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 content) to another octet string (the message digest).

```
DigestAlgorithmIdentifier ::= AlgorithmIdentifier
```

10.1.2 SignatureAlgorithmIdentifier

The SignatureAlgorithmIdentifier type identifies a signature algorithm. Examples include RSA, DSA, and ECDSA. 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, previously distributed symmetric key-encrypting keys, and symmetric key-encrypting keys derived from passwords 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 plaintext) 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-SHA-1. 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
```

10.1.6 KeyDerivationAlgorithmIdentifier

The KeyDerivationAlgorithmIdentifier type is specified in RFC 3211 [PWRI]. The KeyDerivationAlgorithmIdentifier definition is repeated here for completeness.

Key derivation algorithms convert a password or shared secret value into a key-encryption key.

```
KeyDerivationAlgorithmIdentifier ::= 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. 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 CRL, 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 3280 [PROFILE].

The definition of CertificateList is taken 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, a version 1 X.509 attribute certificate (ACv1) [X.509-97], or a version 2 X.509 attribute certificate (ACv2) [X.509-00]. The PKCS #6 extended certificate is obsolete. The PKCS #6 certificate is included for backward compatibility, and PKCS #6 certificates SHOULD NOT be used. The ACv1 is also obsolete. ACv1 is included for backward compatibility, and ACv1 SHOULD NOT be used. The Internet profile of X.509 certificates is specified in the "Internet X.509 Public Key Infrastructure: Certificate and CRL Profile" [PROFILE]. The Internet profile of ACv2 is specified in the "An Internet Attribute Certificate Profile for Authorization" [ACPROFILE].

The definition of Certificate is taken from X.509.

The definitions of AttributeCertificate are taken from X.509-1997 and X.509-2000. The definition from X.509-1997 is assigned to AttributeCertificateV1 (see section 12.2), and the definition from X.509-2000 is assigned to AttributeCertificateV2.

```
CertificateChoices ::= CHOICE {
certificate Certificate,
extendedCertificate [0] IMPLICIT ExtendedCertificate,  -- Obsolete
v1AttrCert [1] IMPLICIT AttributeCertificateV1,        -- Obsolete
v2AttrCert [2] IMPLICIT AttributeCertificateV2 }
```

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

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

CertificateSerialNumber ::= INTEGER
```

10.2.5 CMSVersion

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

```
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 might 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 3280 [PROFILE]. Some of the attributes defined in this section were originally defined in PKCS #9 [PKCS#9]; others were originally defined in a previous version of this specification [OLDCMS]. 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 within signed-data or authenticated-data. The content-type attribute type MUST be present whenever signed attributes are present in signed-data or authenticated attributes present in authenticated-data. The content-type attribute value MUST match the encapContentInfo eContentType value in the signed-data or authenticated-data.

The content-type attribute MUST be a signed attribute or an authenticated attribute; it MUST NOT be an unsigned attribute, unauthenticated attribute, or unprotected attribute.

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
```

Even though the syntax is defined as a SET OF AttributeValue, a content-type attribute MUST have a single attribute value; zero or multiple instances of AttributeValue are not permitted.

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 MUST be present when there are any signed attributes present. Within authenticated-data, the message-digest authenticated attribute type MUST be present when there are any authenticated attributes present.

The message-digest attribute MUST be a signed attribute or an authenticated attribute; it MUST NOT be an unsigned attribute, unauthenticated attribute, or unprotected 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 and AuthAttributes syntax are each defined as a SET OF Attributes. The SignedAttributes in a signerInfo MUST include only one instance of the message-digest attribute. Similarly, the AuthAttributes in an AuthenticatedData MUST include only one instance 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 MUST be a signed attribute or an authenticated attribute; it MUST NOT be an unsigned attribute, unauthenticated attribute, or unprotected 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 MUST be interpreted as 19YY; and

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

GeneralizedTime values MUST 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 and the AuthAttributes syntax are each defined as a SET OF Attributes. The SignedAttributes in a signerInfo MUST NOT include multiple instances of the signing-time attribute. Similarly, the AuthAttributes in an AuthenticatedData 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 MUST NOT be a signed attribute, an authenticated attribute, an unauthenticated attribute, or an unprotected 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 NOT contain a content-type attribute; there is no content type for countersignatures.

2. The signedAttributes field MUST contain a message-digest attribute if it contains any other attributes.

3. 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 an arbitrarily long series of countersignatures.

12. ASN.1 Modules

Section 12.1 contains the ASN.1 module for the CMS, and section 12.2 contains the ASN.1 module for the Version 1 Attribute Certificate.

12.1 CMS ASN.1 Module

```
CryptographicMessageSyntax
    { iso(1) member-body(2) us(840) rsadsi(113549)
      pkcs(1) pkcs-9(9) smime(16) modules(0) cms-2001(14) }

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

  -- Imports from RFC 3280 [PROFILE], Appendix A.1
        AlgorithmIdentifier, Certificate, CertificateList,
        CertificateSerialNumber, Name
           FROM PKIX1Explicit88 { iso(1)
                identified-organization(3) dod(6) internet(1)
                security(5) mechanisms(5) pkix(7) mod(0)
                pkix1-explicit(18) }

  -- Imports from RFC 3281 [ACPROFILE], Appendix B
        AttributeCertificate
           FROM PKIXAttributeCertificate { iso(1)
                identified-organization(3) dod(6) internet(1)
                security(5) mechanisms(5) pkix(7) mod(0)
                attribute-cert(12) }
```

```
   -- Imports from Appendix B of this document
        AttributeCertificateV1
           FROM AttributeCertificateVersion1 { iso(1) member-body(2)
                 us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
                 modules(0) v1AttrCert(15) } ;

 -- 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 SIZE (1..MAX) 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,
     pwri [3] PasswordRecipientInfo,
     ori [4] OtherRecipientInfo }

   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 }
```

```
PasswordRecipientInfo ::= SEQUENCE {
  version CMSVersion,   -- always set to 0
  keyDerivationAlgorithm [0] KeyDerivationAlgorithmIdentifier
                             OPTIONAL,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  encryptedKey EncryptedKey }

OtherRecipientInfo ::= SEQUENCE {
  oriType OBJECT IDENTIFIER,
  oriValue ANY DEFINED BY oriType }

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,
  authAttrs [2] IMPLICIT AuthAttributes OPTIONAL,
  mac MessageAuthenticationCode,
  unauthAttrs [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

KeyDerivationAlgorithmIdentifier ::= AlgorithmIdentifier

CertificateRevocationLists ::= SET OF CertificateList

CertificateChoices ::= CHOICE {
  certificate Certificate,
  extendedCertificate [0] IMPLICIT ExtendedCertificate,  -- Obsolete
  v1AttrCert [1] IMPLICIT AttributeCertificateV1,        -- Obsolete
  v2AttrCert [2] IMPLICIT AttributeCertificateV2 }

AttributeCertificateV2 ::= AttributeCertificate

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 }

-- The CMS Attributes

MessageDigest ::= OCTET STRING

SigningTime  ::= Time

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

Countersignature ::= SignerInfo

-- 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

   ExtendedCertificateOrCertificate ::= CHOICE {
     certificate Certificate,
     extendedCertificate [0] IMPLICIT ExtendedCertificate }
   ExtendedCertificate ::= SEQUENCE {
     extendedCertificateInfo ExtendedCertificateInfo,
     signatureAlgorithm SignatureAlgorithmIdentifier,
     signature Signature }

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

   Signature ::= BIT STRING

   END -- of CryptographicMessageSyntax
```

12.2 Version 1 Attribute Certificate ASN.1 Module

```
   AttributeCertificateVersion1
       { iso(1) member-body(2) us(840) rsadsi(113549)
         pkcs(1) pkcs-9(9) smime(16) modules(0) v1AttrCert(15) }

   DEFINITIONS IMPLICIT TAGS ::=
   BEGIN

   -- EXPORTS All

   IMPORTS

     -- Imports from RFC 3280 [PROFILE], Appendix A.1
           AlgorithmIdentifier, Attribute, CertificateSerialNumber,
           Extensions, UniqueIdentifier
              FROM PKIX1Explicit88 { iso(1)
                   identified-organization(3) dod(6) internet(1)
                   security(5) mechanisms(5) pkix(7) mod(0)
                   pkix1-explicit(18) }
```

```
     -- Imports from RFC 3280 [PROFILE], Appendix A.2
           GeneralNames
              FROM PKIX1Implicit88 { iso(1)
                   identified-organization(3) dod(6) internet(1)
                   security(5) mechanisms(5) pkix(7) mod(0)
                   pkix1-implicit(19) }

     -- Imports from RFC 3281 [ACPROFILE], Appendix B
           AttCertValidityPeriod, IssuerSerial
              FROM PKIXAttributeCertificate { iso(1)
                   identified-organization(3) dod(6) internet(1)
                   security(5) mechanisms(5) pkix(7) mod(0)
                   attribute-cert(12) } ;


   -- Definition extracted from X.509-1997 [X.509-97], but
   -- different type names are used to avoid collisions.

   AttributeCertificateV1 ::= SEQUENCE {
     acInfo AttributeCertificateInfoV1,
     signatureAlgorithm AlgorithmIdentifier,
     signature BIT STRING }

   AttributeCertificateInfoV1 ::= SEQUENCE {
     version AttCertVersionV1 DEFAULT v1,
     subject CHOICE {
       baseCertificateID [0] IssuerSerial,
         -- associated with a Public Key Certificate
       subjectName [1] GeneralNames },
         -- associated with a name
     issuer GeneralNames,
     signature AlgorithmIdentifier,
     serialNumber CertificateSerialNumber,
     attCertValidityPeriod AttCertValidityPeriod,
     attributes SEQUENCE OF Attribute,
     issuerUniqueID UniqueIdentifier OPTIONAL,
     extensions Extensions OPTIONAL }

   AttCertVersionV1 ::= INTEGER { v1(0) }

   END -- of AttributeCertificateVersion1
```

13. References

[ACPROFILE] Farrell, S. and R. Housley, "An Internet Attribute Certificate Profile for Authorization", RFC 3281, April 2002.

[CMSALG] Housley, R., "Cryptographic Message Syntax (CMS) Algorithms", RFC 3269, August 2002.

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

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

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

[OLDCMS] Housley, R., "Cryptographic Message Syntax", RFC 2630, June 1999.

[OLDMSG] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L. and L. Repka, "S/MIME Version 2 Message Specification", RFC 2311, March 1998.

[PROFILE] Housley, R., Polk, W., Ford, W. and D. Solo, "Internet X.509 Public Key Infrastructure: Certificate and CRL Profile", RFC 3280, April 2002.

[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.

[PWRI] Gutmann, P., "Password-based Encryption for S/MIME", RFC 3211, December 2001.

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

[STDWORDS] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[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.

[X.509-00] ITU-T. Recommendation X.509: The Directory - Authentication Framework. 2000.

14. 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 contents 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.

The key management technique employed to distribute message-authentication keys must itself provide data origin authentication, otherwise the contents are delivered with integrity from an unknown source. Neither RSA [PKCS#1, NEWPKCS#1] nor Ephemeral-Static Diffie-Hellman [DH-X9.42] provide the necessary data origin authentication. Static-Static Diffie-Hellman [DH-X9.42] does provide the necessary data origin authentication when both the originator and recipient public keys are bound to appropriate identities in X.509 certificates.

When more than two parties share the same message-authentication key, data origin authentication is not provided. Any party that knows the message-authentication key can compute a valid MAC, therefore the contents could originate from any one of the parties.

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, content is encrypted with Triple-DES using a 168-bit Triple-DES content-encryption key, and the content-encryption key is wrapped with RC2 using a 40-bit RC2 key-encryption 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 the 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.

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 be reduced. Therefore, cryptographic algorithm implementations should be modular, allowing new algorithms to be readily inserted. That is, implementors should be prepared for the set of algorithms that must be supported to change over time.

The countersignature unsigned 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.

15. 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, Simon Blake-Wilson, 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.

16. Authors' Address

Russell Housley
RSA Laboratories
918 Spring Knoll Drive
Herndon, VA 20170
USA
EMail: rhousley@rsasecurity.com

17. Full Copyright Statement

Acknowledgement

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