SOMMAIRE

1. INTRODUCTION

Ce document décrit la syntaxe cryptographique de message CMS (*Crytographic Message Syntax*). Cette syntaxe est employée numériquement pour signer, compresser, authentifier, ou chiffrer le contenu arbitraire de message.

La CMS décrit une syntaxe d'encapsulation pour la protection de données. Elle prend en compte les signatures numériques et le chiffrement. La syntaxe permet des encapsulations multiples ; une enveloppe d'encapsulation peut être insérée à l'intérieur d'une autre enveloppe. De même, une partie peut numériquement signer quelques données précédemment encapsulées. Elle permet également à ce que des attributs arbitraires, tels que la signature d'horodatage, peuvent être signés avec le contenu de message, et prévoit à ce que d'autres attributs, tels que des `countersignatures` soient associés à une signature.

La CMS peut intégrer différentes architectures pour la gestion des clefs (basée sur les certificats – *certificat-based*), telle que celle définie par le groupe de travail de PKIX [**PROFILE**].

Les valeurs de la CMS sont produites en utilisant la norme ASN.1 (*Abstract Syntax Notation One* - [**X.208-88**]), basée sur le codage BER (*BER-encoding* - [**X.209-88**]). Le codage BER (*Basic Encoding Rules*) est utilisé pour représenter concrètement les données ASN.1 en chaînes d'octets. Alors que beaucoup de systèmes sont capables de transmettre arbitrairement des chaînes d'octets de manière sûre, il est bien connu que la plus part des systèmes de messageries électroniques ne le sont pas. Ce document ne mentionne pas les mécanismes de codage des chaînes d'octets pour la transmission fiable dans de tels environnements.

La CMS est dérivée de la version 1.5 de PKCS #7 spécifiée dans RFC 2315 [**PKCS#7**]. Dans la mesure du possible, la compatibilité avec les spécifications antérieures est préservée ; cependant, des changements ont été nécessaires pour adapter la version 1 concernant le transfert d'attributs des certificats, les techniques d'échange de clefs et de chiffrement symétrique des `key-encryption` pour la gestion des clefs.

1.1. Les changements depuis la RFC 2630

Ce document remplace la RFC 2630 [**OLDCMS**] et la RFC 3211 [**PWRI**]. La gestion des clefs basée sur les mots de passe (*Password-based*) est incluse dans les spécifications de la CMS et un mécanisme supplémentaire est spécifié pour intégrer de nouveaux schémas de gestion de clefs sans changement de la CMS. La compatibilité avec le RFC 2630 et 3211 est préservée ; cependant, le transfert d'attributs de certificat de la version 2 est ajouté et l'utilisation d'attributs de certificats de la version 1 est dévaluée.

Les signatures de S/MIME v2 [**OLDMSG**], basées sur PKCS#7 de la version 1.5, sont compatibles avec les signatures de S/MIME v3 [**MSG**], qui elles, sont basées sur la RFC 2630. Cependant, il y a quelques problèmes subtiles de compatibilité entre des signatures utilisant PKCS#7 de la version 1.5 et la CMS. Ces problèmes sont soulevés dans la section 5.2.1.

Les algorithmes cryptographiques spécifiques ne sont pas explicités dans ce document, mais ils sont détaillés dans RFC 2630. L'étude sur les algorithmes cryptographiques spécifiques a été regroupée dans un document séparé [**CMSALG**]. La séparation du protocole et des caractéristiques des algorithmes permet à l'IETF de mettre à jour chaque document indépendamment. Cette spécification n'exige pas l'implémentation d'algorithme particulier. On s'attend plutôt à ce que des protocoles qui se fondent sur la CMS choisissent des

algorithmes appropriés pour leur environnement. Ces algorithmes peuvent être choisis à partir du [**CMSALG**] ou d'ailleurs.

1.2. Terminologie

Dans ce document, les mots clés DOIT, NE DOIT PAS, NÉCESSAIRE, DEVRAIT, NE DEVRAIT PAS, PEUT, et FACULTATIF doivent être interprétés comme décrits dans la [**STDWORDS**].

2. VUE GENERALE D'ENSEMBLE

La CMS est suffisamment générale pour pouvoir prendre en compte un certain nombre de types de contenus (*content types*) différents. Ce document définit un contenu de protection, (`ContentInfo`). Le `ContentInfo` encapsule un simple type de contenu identifié, et ce type identifié peut fournir plusieurs encapsulations.

Ce document définit six types de contenus :

- les données (`data`) ;
- les données signées (`signed-data`) ;
- les données mises sous enveloppe (`enveloped-data`) ;
- les données compressées (hash) (`digested-data`) ;
- les données chiffrées (`encrypted-data`) ;
- les données d'authentification (`authenticated-data`).

Des types de contenus supplémentaires peuvent être définis en dehors de ce document.

Une implémentation qui se conforme à ces spécifications DOIT mettre en application le contenu de protection, `ContentInfo` et DOIT mettre en application les types de contenus `data`, `signed-data` et `enveloped-data`. Les autres types de contenus PEUVENT être mis en application.

Du point de vue conception générale, à chaque type de contenu correspond un déroulement simple (*single-pass*) d'un processus utilisant les règles de codage de base BER (*Basic Encoding Rules*) pour des longueurs non définies (*indefinite-length*) des composants du contenu. L'opération *single-pass* est particulièrement utile si le contenu est grand, stocké sur bandes, ou issu d'un autre processus (encapsulation). L'opération *single-pass* a un inconvénient significatif : il est difficile d'exécuter des opérations de codage en utilisant les règles de codage distinguées DER (*Distinguished Encoding Rules*) [**X.509-88**] pour une opération *single-pass* puisque les longueurs des divers composants ne peuvent être connues à l'avance. Cependant, les attributs signés dans le type de contenu `signed-data` et les attributs authentifiés dans le type de contenu `authenticated-data` doivent être transmis au format DER pour s'assurer que les destinataires peuvent vérifier un contenu qui contient un ou plusieurs attributs non reconnus. Les attributs signés et les attributs authentifiés sont les seuls types de données utilisés dans la CMS qui exigent le codage DER.

3. SYNTAXE GENERALE

L'identifiant d'objet (`OBJECT IDENTIFIER`) suivant, permet d'identifier le type d'information du contenu :

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

La CMS associe un identifiant de type de contenu à son propre contenu. La syntaxe DOIT être sous la forme ASN.1 pour le type `ContentInfo` :

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

ContentType ::= OBJECT IDENTIFIER
```

Les champs de `ContentInfo` ont les significations suivantes :

le `contentType` indique le type du contenu qui lui est associé. C'est un identifiant d'objet : il est constitué d'une chaîne unique de nombres entiers affectée par une autorité qui définit le type de contenu.

Le contenu devient le contenu associé. Le type de contenu peut être déterminé uniquement par le `contentType`. Les types de contenus pour les données (`data`), les données signées (`signed-data`), les données sous enveloppe (`enveloped-data`), les données compressées (hash) (`digested-data`), les données chiffrées (`encrypted-data`) et les données d'authentification (`authenticated-data`) sont définis dans ce document. Si des types de contenus supplémentaires sont définis dans d'autres documents, le type d'ASN.1 défini NE POURAIT PAS être un type CHOIX.

4. TYPE DE CONTENU : DONNEES (« DATA »)

L'identifiant d'objet suivant, définit le type de contenu `data` :

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

Le type de contenu `data` est prévu pour se rapporter aux chaînes arbitraires d'octets, telles que les fichiers « textes » en ASCII. L'interprétation (du texte) est laissée à l'application. De telles chaînes de caractères n'ont pas besoin d'avoir de structure interne particulière (bien qu'elles pourraient avoir leur propre définition en ASN.1 ou une en toute autre structure).

S/MIME emploie des `id-data` pour identifier le contenu codé au format MIME. L'utilisation de cet identifiant de contenu est spécifié dans la RFC 2311 concernant S/MIME v2 [**OLDMSG**] et dans la RFC 2633 pour S/MIME v3 [**MSG**]. Le type de contenu `data` est généralement encapsulé dans un type de contenu `signed-data`, `enveloped-data`, `digested-data`, `encrypted-data`, ou bien `authenticated-data`.

5. TYPE DE CONTENU : DONNEES SIGNEES (« SIGNED-DATA »)

Le type de contenu de données signées (`signed-data`) se compose d'un contenu de n'importe quel type et de zéro à plusieurs valeurs de signature. Autant de signataires (que nécessaire) en parallèle peuvent signer n'importe quel type de contenu.

La caractéristique du type de contenu de données signées est représentée par une signature numérique du signataire dans le contenu du type de contenu de données « data ». Une autre caractéristique est de contenir les certificats et les listes de révocation de certificats CRLs (*Certificate Revocation Lists*).

Le processus par lequel des données signées sont construites implique les étapes suivantes :

1. pour chaque signataire, des données compressées (hash) ou une valeur du hash, sont calculées à partir du contenu avec un algorithme de compression (hashage) spécifique au signataire (*signer-specific*). Si le signataire signe n'importe quelle information autre que le contenu, la compression (hashage) du contenu et les autres informations sont compressées (hash) avec l'algorithme de compression (hashage) propre au signataire (voir la section 5.4) et le résultat devient le « message compressé » (*message-digest*) ;
2. pour chaque signataire, les données compressées (hash) sont numériquement signées en utilisant la clef privée du signataire ;
3. pour chaque signataire, la valeur de la signature et toutes autres informations spécifiques au signataire sont collectées dans une valeur `SignerInfo`, comme définie dans la section 5.3. Les certificats et les CRLs pour chaque signataire et ceux qui ne correspondent à aucun signataire, sont collectés dans cette valeur `SignerInfo` ;
4. tous les algorithmes de compression (hashage) et toutes les valeurs `SignerInfo` de chaque signataire, intervenant dans le processus, sont collectés ainsi que le contenu dans une valeur `SignedData`, comme définie dans la section 5.1.

Un destinataire effectue indépendamment la compression (hashage) . Ces données compressées (hash) et la clef publique du signataire sont employées pour vérifier la valeur de la signature. La clef publique du signataire est référencée soit par un nom caractéristique (distingué) de l'émetteur avec un numéro de série spécifique (*issuer-specific*), soit par un identifiant d'objet de clef qui identifie uniquement le certificat contenant la clef publique. Le certificat du signataire peut être inclus dans le champ `SignedData → certificates`.

Cette section est divisée en six parties. La première partie décrit le niveau supérieur du type `SignedData`, la deuxième partie décrit le type `EncapsulatedContentInfo`, la troisième partie décrit le type `SignerInfo` issu de l'information de chaque signataire (*per-signers*), la quatrième, cinquième, et sixième parties décrivent, respectivement, le calcul des données compressées (hash), la génération de signature, et les processus de vérification de signature.

5.1. Le type « SignedData »

L'identifiant d'objet suivant décrit le type de contenu de données signées (`signed-data`) :

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

Le type de contenu de données signées (signed-data) devra être sous la forme ASN.1 pour le 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
```

Les champs du type SignedData ont les significations suivantes :

5.1.1. le champ « version »

version est la syntaxe du numéro de version. La valeur correspondante dépend de la présence et des contenus des champs certificates, eContentType et SignerInfo.

La version DOIT être déterminée comme suit :

SI (certificates est présents)
ET
(tous les attributs de certificats de version 2 sont présents)
ALORS la version DOIT être 4
SINON
SI ((certificates est présent)
ET
(tous les attributs de certificats de version 1 sont présents))
OU
(encapContentInfo→eContentType est autre que id-data)
OU
(toutes les structures de SignerInfo sont de la version 3)
ALORS la version DOIT être 3
SINON la version DOIT être 1

5.1.2. le champ « digestAlgorithms »

`digestAlgorithms` est une série d'identifiants d'algorithmes de compression de données (hashage). Il PEUT y avoir tout nombre d'éléments dans la série, y compris zéro. Chaque élément identifie l'algorithme de compression de données (hashage), avec tous les paramètres associés, employés par un ou plusieurs signataires. La série est prévue pour énumérer les algorithmes de compression de données (hashage) utilisés par tous les signataires, dans n'importe quel ordre, pour faciliter la vérification de signature en « un seul passage » (*one-pass*). Les implémentations PEUVENT ne pas valider les signatures utilisant un algorithme de compression (hashage) qui n'est pas inclus dans cette série. Le processus de compression (hashage) des données est décrit dans la section 5.4.

5.1.3. le champ « encapContentInfo »

`encapContentInfo` correspond au contenu signé. Il se compose d'un identifiant de type de contenu et du contenu lui-même. Les détails concernant le type `EncapsulatedContentInfo` sont développés dans la section 5.2.

5.1.4. le champ « certificates »

`certificates` est une collection de certificats. Il est bien entendu que ces certificats doivent être en nombre suffisant pour contenir la chaîne hiérarchique de certification complète depuis une autorité racine (identifiée) de certification ou autorité supérieure de certification ainsi que tous les signataires dans le champ `signerInfos`. Il peut y avoir plus de certificats que nécessaires, et il peut y avoir des certificats en nombre suffisant pour contenir deux ou plusieurs chaînes hiérarchiques de certification. Il peut également y avoir moins de certificats que nécessaire, si on s'attend à ce que les destinataires aient des moyens alternatifs d'obtenir les certificats nécessaires (par exemple, à partir d'un jeu ancien de certificats). Le certificat du signataire PEUT être inclus (encapsulé). L'utilisation des attributs de certificats de la version 1 est fortement déconseillée.

5.1.5. le champ « crls »

`crls` est une collection de listes de révocation de certificats (CRLs). Il est bien entendu que l'ensemble doit contenir un nombre suffisant d'information pour déterminer si les certificats dans le champ `certificates` sont valides ou pas. Mais une telle correspondance n'est pas nécessaire. Il PEUT y avoir plus de CRLs que nécessaire et il PEUT également y avoir moins de CRLs que nécessaire.

5.1.6. le champ « signerInfo »

`signerInfos` est une collection d'information sur les co-signataires. Il PEUT y avoir tout nombre d'éléments dans la collection, y compris zéro. Les détails du type `SignerInfo` sont détaillés dans la section 5.3. Puisque chaque signataire peut utiliser une technique de signature numérique et puisque des spécifications futures pourrait mettre à jour la syntaxe, toutes les implémentations DOIVENT manipuler, avec élégance, des versions non implémentées de `SignerInfo`. De plus, puisque toutes les implémentations ne soutiendront pas tous les algorithmes possibles de signature, elles DOIVENT avec élégance manipuler des algorithmes de signature non implémentés lorsqu'elles sont produites.

5.2. Le type « EncapsulatedContentInfo »

Le contenu est représenté dans le type `EncapsulatedContentInfo` :

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

ContentType ::= OBJECT IDENTIFIER
```

Les champs du type `EncapsulatedContentInfo` ont les significations suivantes :

> `eContentType` est un identifiant d'objet. L'identifiant d'objet indique uniquement le type du contenu.
>
> `eContent` est le contenu lui-même, caractérisé par une chaîne d'octets. Le champ `eContent` n'a pas besoin d'être encodé au format DER.

L'omission facultative de `eContent` dans le champ `EncapsulatedContentInfo` permet de construire des signatures « externes ». Dans le cas de signatures externes, le contenu à signer, est absent de la valeur `EncapsulatedContentInfo` incluse dans le type du contenu `signed-data`.

Si la valeur `EncapsulatedContentInfo` → `eContent` est absente, alors la valeur de `signatureValue` est calculée et l'`eContentType` lui est assigné comme si la valeur `eContent` était présente.

Dans le cas dégénéré où il n'y a aucun signataire, la valeur du champ `EncapsulatedContentInfo` à « signer », n'est pas pertinente. Dans ce cas-là, le type de contenu dans la valeur d'`EncapsulatedContentInfo` à « signer », DOIT correspondre avec celui d'`id-data` (comme défini dans la section 4), et le champ du contenu de la valeur d'`EncapsulatedContentInfo` DOIT être omis.

5.2.1. Compatibilité avec PKCS #7

Cette section contient un signal d'avertissement pour les utilisateurs qui souhaitent implémenter à la fois les types de contenus `SignedData` de la CMS et du PKCS #7 [PKCS#7]. La CMS et le PKCS #7 identifient, tous les deux, le type du contenu encapsulé avec un identifiant d'objet, mais le type du contenu au format ASN.1 lui-même est variable selon le type de contenu `SignedData` de PKCS #7.

PKCS #7 définit le champ `content` par :

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

La CMS définit le champ `eContent` par :

```
eContent [0] EXPLICIT OCTET_STRING OPTIONAL
```

La définition de la CMS est plus facile d'utilisation dans la plus part des applications et elle est compatible aussi bien avec la version S/MIME v2 que v3. Les messages signés par

S/MIME utilisant la CMS et PKCS #7 sont compatibles puisque les identifications des formats de messages signés sont spécifiées dans la RFC 2311 pour S/MIME v2 [**OLDMSG**] et dans la RFC 2633 pour S/MIME v3 [**MSG**]. S/MIME v2 encapsule le contenu de MIME dans un type de contenu `Data` (c'est-à-dire, une chaîne d'octets) placé dans le champ `SignedData` → `contentInfo` → `ANY` et S/MIME v3 transfert (encapsule) le contenu de MIME en une chaîne d'octets (`OCTET_STRING`) dans le champ : `SignedData` → `encapContentInfo` → `eContent` → `OCTET_STRING`.

Par conséquent, dans les deux versions de S/MIME, v2 et v3, le contenu de MIME est placé dans une chaîne d'octets et la compression (hashage) des données est calculée à partir des parties identiques du contenu. Ce qui revient à dire que la compression (hashage) des données (de message) n'est calculée qu'à partir des octets comportant la valeur `OCTET_STRING`, ni l'étiquette (*tag* - partie caractérisant les données brutes) ni les longueurs d'octets ne sont incluses dans le calcul de compression (hashage) .

Il y a incompatibilités entre les types `signedData` de la CMS et du PKCS #7 lorsque le contenu encapsulé n'est pas formaté en utilisant le type `Data`. Par exemple, lorsqu'un reçu signé au format ESS (*Enhanced Security Services*) de la RFC 2634 [**ESS**] est encapsulé dans le type de `signedData` de la CMS, alors la `Receipt` → `SEQUENCE` est encodée dans `signedData` → `encapContentInfo` → `eContent` → `OCTET_STRING` et les données compressées (hash) sont calculées en utilisant le encodage complet de `Receipt` → `SEQUENCE` (y compris les étiquettes, les longueurs et les valeurs des octets). Cependant, si un reçu signé au format RFC 2634 est encapsulé dans le type `signedData` de PKCS #7, alors le `Receipt` → `SEQUENCE` est encodé au format DER [X.509-88] dans le champ `SignedData` → `contentInfo` → `content` → `ANY` (une `SEQUENCE` et non pas un `OCTET_STRING`). Par conséquent, les données compressées (hash) sont calculées en utilisant seulement les valeurs en octets de l'encodage de `Receipt` → `SEQUENCE`.

La stratégie suivante peut être employée pour réaliser la compatibilité antérieure avec PKCS #7 lors du processus des types de contenu de `SignedData`. Si l'implémentation n'est pas capable d'interpréter (décoder), par le format ASN.1, le type `signedData` en utilisant la syntaxe de la CMS dans `signedData` → `encapContentInfo` → `eContent` → `OCTET_STRING`, alors l'implémentation PEUT essayer de décoder le type `signedData` en utilisant la syntaxe PKCS #7 dans `SignedData` → `contentInfo` → `content` → `ANY` et de calculer la compression de données (hashage) en conséquence.

La stratégie suivante peut être employée pour réaliser la compatibilité antérieure avec PKCS #7 lors de la création du type de contenu `SignedData` dans lequel le contenu encapsulé n'est pas formaté en utilisant le type `Data`. Les implémentations PEUVENT examiner la valeur de l'`eContentType`, et ajuster ainsi le codage DER prévu du champ `eContent` basé sur la valeur de l'identifiant d'objet. Par exemple, pour supporter `Microsoft AuthentiCode`, l'information suivante PEUT être incluse :

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

`eContent` contient de l'information de signature `AuthentiCode` encodé DER

5.3. Le type « SignerInfo »

L'information de chaque signataire (*per-signers*) est représentée dans le 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
```

Les champs du type `SignerInfo` ont les significations suivantes :

`version` est la syntaxe correspondant au numéro de version. Si le champ `SignerIdentifier` correspond à celui de `CHOICE` → `issuerAndSerialNumber`, alors la version DOIT être 1. Si le champ `SignerIdentifier` est le même que `subjectKeyIdentifier`, alors la version DOIT être 3.

`sid` spécifie le certificat du signataire (et de ce fait la clef publique du signataire). La clef publique du signataire est nécessaire pour vérifier la signature par le destinataire. `SignerIdentifier` fournit deux solutions pour spécifier la clef publique du signataire. La solution `issuerAndSerialNumber` identifie le certificat du signataire par le nom caractéristique (distingué) de l'émetteur et par le numéro de série de son certificat ; la solution `subjectKeyIdentifier` identifie le certificat du signataire par la valeur

subjectKeyIdentifier de la prolongation X.509. Les implémentations DOIVENT supporter la réception des formes issuerAndSerialNumber et subjectKeyIdentifier de SignerIdentifier. Lors de la génération d'un SignerIdentifier, des implémentations PEUVENT supporter l'une des deux formes (issuerAndSerialNumber ou subjectKeyIdentifier) et n'employer qu'à chaque fois celle qui a été choisie, ou alors, des implémentations PEUVENT arbitrairement mélanger les deux formes.

digestAlgorithm identifie l'algorithme de compression de données (hashage) (de message), et tous les paramètres associés, employés par le signataire. Les données compressées (hash) sont calculées soit à partir du contenu à signer soit à partir du contenu ainsi que de ses attributs signés en utilisant le processus décrit dans la section 5.4. L'algorithme de compression de données (hashage) DEVRAIT être un de ceux énumérés dans le champ digestAlgorithms de SignerData qui lui est associé. Les implémentations PEUVENT échouer à la validation de signatures lorsqu'elles utilisent un algorithme de compression (hashage) qui ne fait pas partie de ceux établis dans SignedData → digestAlgorithms.

signedAttrs est une collection d'attributs qui sont signés. Le champ est facultatif, mais il DOIT être présent si le type de contenu de la valeur d'EncapsulatedContentInfo à signer, ne correspond pas à id-data. Le champ SignedAttributes DOIT être encodé au format DER, même si le reste de la structure est codé au format BER. Des types utiles d'attribut, tels que la signature d'horodatage, sont définis dans la section 11. Si le champ est présent, il DOIT contenir, au minimum, les deux attributs suivants :

> Un attribut content-type a comme valeur le type de contenu de la valeur EncapsulatedContentInfo à signer. La section 11.1 définit l'attribut content-type. Cependant, l'attribut content-type NE DOIT PAS être employé en tant qu'attribut non signé de countersignature comme il est défini dans la section 11.4.
>
> Un attribut message-digest, ayant comme valeur les données compressées (hash) du contenu. La section 11.2 définit l'attribut message-digest.

signatureAlgorithm identifie l'algorithme de signature, et tous ses paramètres associés. Il est utilisé par le signataire pour générer la signature numérique.

signature est le résultat de la génération de la signature numérique, en utilisant les données compressées (hash) et la clef privée du signataire. Les détails de la signature dépendent de l'algorithme de signature utilisé.

unsignedAttrs est une collection d'attributs qui ne sont pas signés. Le champ est facultatif. Des types utiles d'attributs, tels que les countersignatures, sont définis dans la section 11.

Les champs du type SignedAttribute et UnsignedAttribute ont les significations suivantes :

> attrType indique le type d'attribut. C'est un identifiant d'objet.
>
> attrValues est un ensemble de valeurs qui composent l'attribut. Le type de chaque valeur dans cet ensemble peut être déterminé uniquement par attrType. Ce champ attrType peut imposer des restrictions sur le nombre de valeurs de cet ensemble.

5.4. Le processus de calcul de données compressées (hash)

Le processus informatique de compression de données (hashage) calcule les données compressées (hash) soit à partir du contenu à signer, soit à partir de l'ensemble : contenu et attributs signés. Dans les autres cas, les données initiales pour le processus de calcul de compression (hashage) correspondent à la « valeur » du contenu encapsulé à signer. Spécifiquement, les données initiales correspondent aux valeurs du champ : `encapContentInfo` → `eContent` → `OCTET_STRING` auxquelles s'applique le processus de signature. Seuls les octets contenus dans la valeur `eContent` → `OCTET_STRING` sont pris en compte par l'algorithme de compression de données (hashage), mais pas l'étiquette (*tag*) ni la taille des octets.

Le résultat du processus de calcul de compression de données (hashage) dépend de la présence ou non du champ `signedAttrs`. Si ce champ est absent, le résultat est simplement la compression (hashage) du contenu, comme décrit ci-dessus. Si ce champ est présent, quoi qu'il en soit, le résultat correspond à la compression de données (hashage) de l'encodage DER complet, de la valeur `SignedAttrs` contenue dans le champ `signedAttrs`. Puisque la valeur `SignedAttrs`, lorsqu'elle est présente, doit contenir le `content-type` et les attributs des données compressées (hash) (`message-digest` attributes), ces valeurs sont indirectement incluses dans le résultat du calcul. Les attributs de `content-type` NE DOIVENT PAS être inclus dans un attribut non signé de `countersignature` comme défini dans la section 11.4. Un encodage, à part, du champ `signedAttrs` est réalisé pour le calcul de compression de données (hashage). L'étiquette (*tag*) `IMPLICIT [0]` dans `signedAttrs` n'est pas utilisée pour l'encodage DER, mais plutôt l'étiquette `EXPLICIT SET OF`. Ce qui signifie que l'encodage DER de l'étiquette `EXPLICIT SET OF`, plutôt que `IMPLICIT [0]`, DOIT être incluse dans le calcul de compression de données (hashage) avec la longueur et le contenu d'octets de la valeur `SignedAttributes`.

Si le champ `signedAttrs` est absent, seuls les octets comprenant la valeur de `signedData` → `encapContentInfo` → `eContent` →`OCTET_STRING` (c'est-à-dire le contenu d'un fichier) correspondent aux données calculées pour la compression (hashage) . Ceci a pour avantage le fait que la longueur du contenu à signer doit être connu au préalable pour le processus de génération de signature.

Toute fois, l'étiquette `encapContentInfo` → `eContent` → `OCTET_STRING` ainsi que la longueur des octets ne sont pas prises en compte pour le calcul de compression (hashage) , elles sont protégées par d'autres moyens. La longueur des octets est protégée par la nature de l'algorithme de compression (hashage) puisqu'il est techniquement (informatiquement) impossible de trouver deux contenus de messages, de tailles différentes, qui ont les mêmes données compressées (hash).

5.5. Le processus de génération de signature

Les données issues du processus de génération de signature incluent le résultat du processus de calcul de compression (hashage) ainsi que la clef privée du signataire. Les détails concernant la génération de signature dépend de l'algorithme de signature utilisé. L'identifiant d'objet, avec tous les paramètres associés, qui spécifie l'algorithme de signature utilisé, se trouve dans le champ `signatureAlgorithm`. La valeur de la signature générée par le signataire DOIT être encodée en tant que `OCTET_STRING` et DOIT se trouver dans le champ `signature`.

5.6. Le processus de vérification de signature

Les données issues du processus de vérification de signature incluent le résultat du processus de calcul de compression (hashage) ainsi que la clef publique du signataire. Le destinataire PEUT obtenir la bonne clef publique du signataire par tous moyens, mais la méthode préférentielle est de récupérer un certificat obtenu à partir du champ `SignedData → certificates`. La sélection et la validation de la clef publique du signataire PEUVENT aussi bien être basées sur la validation du chemin de certification (voir [**PROFILE**]) que sur un autre contexte externe, mais ceci sort du cadre de ce document. Le détail de la vérification de signature dépend de l'algorithme de signature utilisé.

Le destinataire NE DOIT PAS faire de liens entre toutes les valeurs de données compressées (hash) calculées par l'expéditeur. Si le champ `SignedData → signerInfo` inclus `signedAttributes`, alors les données compressées (hash) DOIVENT être calculées, comme il est décrit dans la section 5.4. Pour que la signature soit valide, la valeur des données compressées (hash), calculée par le destinataire DOIT être la même que la valeur de l'attribut `messageDigest` inclus dans `signedAttributes` de `SignedData → signerInfo`.

Si SignedData signerInfo inclus signedAttributes, alors la valeur de l'attribut `content-type` DOIT correspondre avec la valeur `SignedData → encapContentInfo → eContentType`.

6. TYPE DE CONTENU : DONNEES MISES SOUS ENVELOPPES (« ENVELOPED-DATA »)

Le type de contenu `enveloped-data` (données sous enveloppes) se compose d'un contenu chiffré de n'importe quel type et de clefs chiffrées de type `content-encryption` pour un ou plusieurs destinataires. La combinaison du contenu chiffré et d'une clef chiffrée `content-encryption` pour un destinataire correspond à une « enveloppe numérique » (*digital envelope*) pour ce destinataire. N'importe quel type de contenu peut être enveloppé pour un nombre arbitraire de destinataires en utilisant n'importe laquelle des trois techniques de gestion des clefs pour chaque destinataire.

L'application typique de ce type de contenu `enveloped-data` permet de représenter les enveloppes numériques d'un ou plusieurs destinataires dans les types `content` de `data` ou `signed-data → content`.

`Enveloped-data` est construit à partir des étapes suivantes :

1. Une clef `content-encryption` est générée aléatoirement dans un algorithme particulier `content-encryption`.
2. Cette clef `content-encryption` est chiffrée pour chaque destinataire. Les détails de ce chiffrement dépendent de l'algorithme utilisé pour la gestion des clefs, mais quatre techniques principales sont retenues :
   - la clef de transport (*key transport*) : la clef `content-encryption` est chiffrée grâce à la clef publique du destinataire ;
   - la clef d'approbation (*key agreement*) : la clef publique du destinataire et la clef privée de l'expéditeur sont utilisées pour générer une clef symétrique confidentielle (*pairwise symmetric key*), ainsi la clef `content-encryption`

est chiffrée par la *pairwise symmetric key* ;

- les clefs symétriques de chiffrement de clef (*symetric key-encryption keys*) : la clef `content-encryption` est chiffrée par une clef symétrique de chiffrement de clef (*symetric key-encryption key*) précédemment distribuée ;
- les mots de passe (*passwords*) : la clef `content-encryption` est chiffrée dans une clef de chiffrement de chiffrement de clef (*key-encryption key*) qui est dérivée d'un mot de passe ou de toute autre valeur secrète partagée.

3. Pour chaque destinataire, la clef chiffrée `content-encryption` et toute autre information spécifique `recipient-specific` se retrouvent dans la valeur `RecipientInfo`, définie dans la section 6.2.
4. Le contenu est chiffré avec la clef `content-encryption`. `content-encryption` peut exiger que le contenu soit fractionné (notion de *padding*) en plusieurs blocks d'une certaine longueur ; voir la section 6.3.
5. Les valeurs de `RecipientInfo` pour tous les destinataires sont rassemblées ainsi que le contenu chiffré pour former une valeur `EnvelopedData` comme définie dans la section 6.1.

Un destinataire ouvre l'enveloppe numérique en déchiffrant une des clefs chiffrées `content-encryption` et déchiffre ensuite le contenu chiffré avec la clef récupérée `content-encryption`.

Cette section est divisée en quatre parties. La première partie décrit le type supérieur `EnvelopedData`, la deuxième partie décrit le type d'information (par destinataire) `RecipientInfo`, et les troisième et quatrièmes parties décrivent les processus `content-encryption` et `key-encryption`.

6.1. Le type « EnvelopedData »

L'identifiant d'objet suivant définit le type de contenu `enveloped-data` :

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

Le type de contenu `enveloped-data` devra être sous la forme ASN.1 pour le 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
```

Les champs du type `EnvelopedData` ont les significations suivantes :

`version` est la syntaxe du numéro de version. La valeur correspondante dépend des champs `originatorInfo`, `RecipientInfo` et `unprotectedAttrs`.

La version DOIT être déterminée comme suit :

SI ((`originatorInfo` est présents)
ET
(tous les attributs de certificats de version 2 sont présents))
OU
(toutes les structures `RecipientInfo` incluent pwri)
OU
(toutes les structures `RecipientInfo` incluent ori)
ALORS la version DOIT être 3
SINON
SI (`originatorInfo` est présent)
OU
(`unprotectedAttrs` est présent)
OU
(toutes les structures `RecipientInfo` ont une version autre que 0)
ALORS la version DOIT être 2
SINON la version DOIT être 0

originatorInfo fournit sur option des informations concernant l'expéditeur (créateur du message). Ce champ est présent seulement s'il est demandé par l'algorithme de gestion des clefs. Il peut contenir des certificats (certs) et des CRLs (crls) :

> certs est une collection de certificats. certs peut contenir plusieurs certificats de l'expéditeur associés aux différents algorithmes de gestion des clefs, qu'il utilise. certs peut également contenir des certificats d'attributs liés à l'expéditeur. Les certificats contenus dans certs sont prévus pour être en nombre suffisant pour que tous les destinataires puissent remonter (et vérifier) les chemins de certification, jusqu'à l'autorité « racine » identifiée ou autorité « supérieure » de certification. Cependant, certs peut contenir plus de certificats que nécessaires, et il peut y avoir un nombre suffisant de certificats pour remonter deux ou plusieurs chemins indépendant de certification jusqu'aux autorités supérieures de certification. Alternativement, certs peut contenir moins de certificats que nécessaires, si on s'attend à ce que les destinataires aient des moyens alternatifs d'obtenir les certificats nécessaires (par exemple, à partir d'un ensemble d'anciens certificats).
>
> crls est une collection de CRLs. Il est prévu que cette collection contienne suffisamment d'informations pour déterminer si les certificats dans le champ certs sont valides ou pas, mais une telle correspondance n'est pas nécessaire. Il PEUT y avoir plus de CRLs que nécessaire et il PEUT également y avoir moins de CRLs que nécessaire.

recipientInfos est une collection d'information de chaque destinataire (*per-recipient*). Il DOIT y avoir au moins un élément dans cette collection.

encryptedContentInfo est l'information du contenu chiffré.

unprotectedAttrs est une collection d'attributs qui ne sont pas chiffrés. Ce champ est facultatif. Des types utiles d'attributs sont définis dans la section 11.

Les champs du type EncryptedContentInfo ont les significations suivantes :

> contentType indique le type de contenu.
>
> contentEncryptionAlgorithm identifie l'algorithme de content-encryption, et tous ses paramètres associés. Il est utilisé pour chiffrer le contenu. Le processus content-encryption est décrit dans la section 6.3. Le même algorithme et la même clef de content-encryption sont utilisés pour tous les destinataires.
>
> encryptedContent est le résultat du chiffrement du contenu. Ce champ est facultatif et si le champ n'est pas présent, sa valeur prévue doit être assurée par d'autres moyens.

Le champ recipientInfos vient avant le champ encryptedContentInfo de sorte qu'une valeur d'EnvelopedData puisse être traitée en un seul passage (*single pass*).

6.2. Le type « RecipientInfo »

L'information de chaque destinataire (*Per-recipient*) est représentée dans le type RecipientInfo. RecipientInfo a un format différent pour chacune des techniques de gestion des clefs soutenues. N'importe laquelle des techniques de gestion de clefs peut être utilisée pour chaque destinataire sur le même contenu chiffré. Dans tous les cas, la clef chiffrée de chiffrement du contenu est transférée à un ou plusieurs destinataires.

Puisque toutes les implémentations ne soutiendront pas chaque algorithme de gestion de clefs possible, toutes les implémentations DOIVENT avec élégance manipuler des algorithmes non implémentés lorsqu'elles sont générées. Par exemple, si un destinataire reçoit une clef de chiffrement de contenu chiffrée par l'algorithme de clef publique RSA en utilisant le RSA-OAEP et si l'exécution supporte seulement le RSA PKCS #1 v1.5, alors le lancement d'un échec « élégant » doit être implémenté.

Les implémentations DOIVENT supporter les clefs de transport (*key transport*), les clefs d'approbation (*key agreement*) et les clefs symétriques de chiffrement de clef (*symmetric key-encryption keys*) préalablement distribuées, comme représentées respectivement par `ktri`, `kari`, et `kekri`. Les implémentations PEUVENT supporter les clefs de gestion de mot de passe (*password-based key management*), représentées par `pwri`. Les implémentations PEUVENT supporter n'importe quelle autre technique de gestion de clefs représentée par `ori`. Puisque chaque destinataire peut utiliser une technique de gestion de clefs différente et puisque des spécifications futures pourrait définir des techniques de gestion de clefs supplémentaires, toutes les implémentations DOIVENT avec élégance manipuler des solutions de rechange non implémentées dans le champ `RecipientInfo` → `CHOICE`, toutes ces implémentations DOIVENT avec élégance manipuler des versions non implémentées de solutions de rechange autrement supportées dans le champ `RecipientInfo` → `CHOICE`, et toutes les implémentations DOIVENT avec élégance manipuler des solutions de rechange non implémentées ou inconnues du champ `ori`.

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

EncryptedKey ::= OCTET STRING
```

6.2.1. Le type « KeyTransRecipientInfo »

L'information de chaque destinataire (*per-recipient*) utilisant une clef de transport (*key transport*) est représentée dans le type `KeyTransRecipientInfo`. Chaque instance de `KeyTransRecipientInfo` transfert la clef de chiffrement du contenu (`content-encryption key`) vers un destinataire.

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

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

Les champs du type `KeyTransRecipientInfo` ont les significations suivantes :

`version` est la syntaxe du numéro de version. Si le champ `RecipientIdentifier` correspond à celui de `CHOICE` → `issuerAndSerialNumber`, alors la version DOIT être 0. Si le champ de `RecipientIdentifier` correspond à celui de `subjectKeyIdentifier`, alors la version DOIT être 2.

`rid` indique le certificat ou la clef du destinataire qui a été utilisé par l'expéditeur pour protéger la clef de chiffrement du contenu. Le champ `RecipientIdentifier` fournit deux possibilités pour spécifier le certificat du destinataire, et de ce fait la clef publique du destinataire. La première possibilité concerne le certificat du destinataire qui doit contenir une clef publique de clef de transport. Par conséquent, un certificat X.509, version 3, du destinataire qui contient une prolongation d'utilisation de clef DOIT spécifier le bit de `keyEncipherment`. La clef de chiffrement du contenu est chiffrée avec la clef publique du destinataire. La deuxième possibilité concerne le champ `issuerAndSerialNumber` qui permet d'identifier le certificat du destinataire par le nom distingué de l'émetteur et le numéro de série du certificat ; le champ `subjectKeyIdentifier` identifie le certificat du destinataire par la valeur `subjectKeyIdentifier` de l'extension X.509. Concernant le processus à réaliser par le destinataire, les implémentations DOIVENT supporter les deux possibilités (décrites ci-dessus) pour spécifier le certificat du destinataire. Concernant le processus à réaliser par l'expéditeur, les implémentations DOIVENT soutenir au moins une de ces deux solutions.

`keyEncryptionAlgorithm` spécifie l'algorithme de chiffrement de clef, et tous ses paramètres associés, utilisé pour chiffrer la clef de chiffrement du contenu pour le destinataire. Le procédé de chiffrement de clef est décrit dans la section 6.4.

`encryptedKey` est le résultat du chiffrement de la clef de chiffrement du contenu pour le destinataire.

6.2.2. Le type « KeyAgreeRecipientInfo »

L'information des destinataires utilisant une clef d'approbation est représentée dans le type `KeyAgreeRecipientInfo`. Chaque instance de `KeyAgreeRecipientInfo` doit transmettre la clef de chiffrement du contenu vers un ou plusieurs destinataires qui utilisent le même algorithme de génération de clef d'approbation et les mêmes paramètres de domaine pour cet algorithme.

```
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
```

Les champs du type `KeyAgreeRecipientInfo` ont les significations suivantes :

`version` est la syntaxe du numéro de version. Elle DOIT toujours être 3.

`originator` est lié au champ `CHOICE` qui détermine trois possibilités pour spécifier la clef publique de clef d'approbation de l'expéditeur. L'expéditeur utilise la clef privée correspondante et la clef publique du destinataire pour générer une clef symétrique confidentielle (*pairwise key*). La clef de chiffrement du contenu est chiffrée par cette clef confidentielle (*pairwise key*).

La première possibilité réside dans le champ `issuerAndSerialNumber` qui spécifie le certificat de l'expéditeur, et de ce fait, la clef publique de l'expéditeur, par le biais du nom distingué de l'émetteur et du numéro de série du certificat.

La deuxième possibilité réside dans le champ `subjectKeyIdentifier` qui spécifie le certificat de l'expéditeur, et de ce fait, la clef publique de l'expéditeur, grâce à la valeur `subjectKeyIdentifier` de l'extension X.509.

La troisième possibilité réside dans le champ `originatorKey` qui inclut l'identifiant de l'algorithme la clef publique de clef d'approbation de l'expéditeur. Cette solution permet l'anonymat de créateur puisque la clef publique n'est pas certifiée.

Les implémentations DOIVENT supporter chacune des trois possibilités pour indiquer la clef publique de l'expéditeur.

`ukm` est un champ optionnel. Avec des algorithmes de génération de clefs d'approbation, l'expéditeur fournit un UKM (*User Keying Material*) pour s'assurer qu'une clef différente est générée à chaque fois lors de la génération d'une clef confidentielle commune des deux parties (*pairwise key*). Les implémentations DOIVENT supporter le processus des destinataires sur le champ `KeyAgreeRecipientInfo` → `SEQUENCE` qui contient un champ `ukm`. Les implémentations qui ne supportent pas les algorithmes de génération de clef d'approbation qui se servent d'UKMs DOIVENT avec élégance manipuler la présence d'UKMs.

`keyEncryptionAlgorithm` identifie l'algorithme de chiffrement de clefs (*key-encryption algorithm*), et tous ses paramètres associés, utilisé pour chiffrer la clef de chiffrement du contenu (*content-encryption key*) avec la clef de chiffrement de clef (*key-encryption key*). Le processus de chiffrement de clef (*key-encryption process*) est décrit dans la section 6.4.

Le champ `recipientEncryptedKeys` inclut un identifiant du destinataire et une clef chiffrée pour un ou plusieurs destinataires. Le champ `KeyAgreeRecipientIdentifier` est lié au champ `CHOICE` qui détermine deux possibilités pour spécifier le certificat du destinataire, et de ce fait la clef publique du destinataire, qui a été utilisée par l'expéditeur pour générer une clef confidentielle commune des deux parties (*pairwise key-encryption key*).

La première possibilité réside dans le fait que le certificat du destinataire doit contenir une clef publique de clef d'approbation. Par conséquent, un certificat X.509, version 3, du destinataire qui contient une extension de clef d'utilisation DOIT spécifier le bit de `keyAgreement`. La clef de chiffrement du contenu est chiffrée par la clef de chiffrement de clef commune (*pairwise key-encryption key*).

La deuxième possibilité réside dans le champ `issuerAndSerialNumber` qui spécifie le certificat du destinataire grâce au nom distingué de l'émetteur et du numéro de série du certificat ; le champ `RecipientKeyIdentifier` est décrit ci-dessous. `encryptedKey` est le résultat du chiffrement de la clef de chiffrement du contenu par la clef de chiffrement de la clef commune (*pairwise key-encryption key*) générée en utilisant l'algorithme de génération de clef d'approbation.

Les implémentations DOIVENT supporter les deux possibilités pour spécifiuer le certificat du destinataire.

Les champs du type `RecipientKeyIdentifier` ont les significations suivantes :

`subjectKeyIdentifier` identifie le certificat du destinataire par la valeur d'extension X.509 de `subjectKeyIdentifier`.

`date` est un champ optionnel. Lorsqu'il est présent, `date` spécifie quel est l'UKMs préalalement distribué au destinataire a été utilisé par l'expéditeur.

`other` est un champ optionnel. Lorsqu'il est présent, ce champ contient l'information supplémentaire utilisée par le destinataire pour localiser les éléments publics issus de clef de chiffrement (*public keying material*) utilisé par l'expéditeur.

6.2.3. Le type « KEKRecipientInfo »

L'information du destinataire utilisant les clefs symétriques préalablement distribuées, est représentée dans le type `KEKRecipientInfo`. Chaque instance de `KEKRecipientInfo` doit permettre de transférer la clef de chiffrement du contenu à un ou plusieurs destinataires qui possèdent déjà les clefs de chiffrement de clefs préalablement distribuées.

```
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 }
```

Le schamps du type `KEKRecipientInfo` ont les significations suivantes:

`version` est la syntaxe du numéro de version. Elle DOIT être 4.

`kekid` spécifie une clef symétrique de chiffrement de clef qui a été préalablement distribuée à l'expéditeur et à un ou plusieurs destinataires.

`keyEncryptionAlgorithm` identifie l'algorithme générant le chiffrement de clef, et ses paramètres associés, utilises pour chiffrer la clef de chiffrement de contenu avec la clef de chiffrement de clef. Le processus de chiffrement de clef est décrit dans la section 6.4.

`encryptedKey` est le résultat du chiffrement de la clef de chiffrement de contenu par la clef de chiffrement de clef.

Les champs du type `KEKIdentifier` ont les significations suivantes:

`keyIdentifier` identifie la clef de chiffrement de clef qui a été distribuée préalablement à l'expéditeur et à un ou plusieurs destinataires.

`date` est un champ optionnel. Lorsqu'il est présent, le champ `date` spécifie une clef unique de chiffrement de clef à partir d'un jeu de clefs qui a été préalablement distribué.

`other` est un champ optionnel. Lorsqu'il est présent, ce champ contient des informations supplémentaires utilisées par le destinataire pour déterminer la clef de chiffrement de clef utilisée par l'expéditeur.

6.2.4. Le type « PasswordRecipientInfo »

Les information du destinataire utilisant un mot de passe ou la valeur d'un secret partagé, sont représentées dans le champ `PasswordRecipientInfo`. Chaque instance de `PasswordRecipientInfo` doit transférer la clef de chiffrement de contenu à un ou plusieurs destinataires qui possèdent un mot de passe ou la valeur d'un secret partagé.

Le type PasswordRecipientInfo est spécifié dans la RFC 3211 [PWRI]. La structure de `PasswordRecipientInfo` est répétée ici pour plus de clarté :

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

Les champs du type `PasswordRecipientInfo` ont les significations suivantes :

`version` est la syntaxe du numéro de version. Elle DOIT être 0.

`keyDerivationAlgorithm` identifie l'algorithme de derivation de clef (*key-derivation algorithm*), et ses paramètres associés, utilisés pour deriver la clef de chiffrement de clef à partir d'un mot de passe ou d'une valeur d'un secret partagé. Si ce champ est absent, la clef de chiffrement de clef est supplée à partir d'une source externe, par exemple un logiciel, une carte à puce, une carte PCcard, un boîtier externe (lecteur de carte à puce sécurisé, boîtier chiffrant)

`keyEncryptionAlgorithm` identifie l'algorithme de chiffrement, et tous ses paramètres associés, utilisés pour chiffrer la clef de chiffrement du contenu avec la clef de chiffrement de clef.

`encryptedKey` est le résultat du chiffrement de la clef de chiffrement du contenu.

6.2.5. Le type « OtherRecipientInfo »

L'information du destinataire pour des techniques supplémentaires de gestion de clefs est représentée dans le type `OtherRecipientInfo`. Le type `OtherRecipientInfo` permet de réaliser des techniques de gestion de clefs à travers les clefs de transport (*key transport*), les clefs d'approbation (*key agreement*), les clefs symétriques de chiffrement de clef (*symmetric key-encryption keys*) préalablement distribuées et les clefs de gestion basées sur des mots de passe (*password-based key*) pour être spécifiées dans de futurs documents.

Un identifiant d'objet identifie uniquement de telles techniques de gestion des clefs.

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

Les champs du type `OtherRecipientInfo` ont les significations suivantes :

`oriType` identifie la technique de gestion de clefs.

`oriValue` contient les elements de données du protocole nécessaire à un destinataire utilisant une technique identifiée de gestion de clefs.

6.3. Le processus de chiffrement de données « Content-encryption »

La clef de chiffrement de contenu pour l'algorithme désiré de chiffrement de contenu, est générée de manière aléatoire. Les données à protéger sont « paddées » (complétées par d'autres données). Le processus de « padding » est décrit ci dessous, ainsi, les données « complétées » sont chiffrées en utilisant la clef de chiffrement de contenu. L'opération de chiffrement complète (« notion de mappage ») une chaîne arbitraire d'octets (les données) à une autre chaîne d'octets (texte chiffré) sous le contrôle d'une clef de chiffrement de contenu. Les données chiffrées sont incluses dans le champ `envelopedData` → `encryptedContentInfo` → `encryptedContent` →`OCTET_STRING`.

Des algorithmes de chiffrement de contenu prennent en compte le fait que la taille des données à chiffrer doit être un multiple de `k` octets, où `k` doit être plus grand que un. Pour de tels algorithmes, les données initiales doivent être complétées (*padded*) en suivant (à partir de la fin), par `k-(lth mod k)` octets, où `lth` est la taille (*length*) des données initiales et `mod` le modulo. En d'autres termes, les données sont complétées, à la fin, par une des chaînes suivantes :

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

Le « padding » peut être enlevé sans ambiguité puisque toutes les données sont complétées, incluant les valeurs de données qui ont déjà une taille multiple d'une taille de blocks. Aucune chaîne de « padding » n'a de suffixe identique. Cette méthode de « padding » est bien définie si et seulement si `k` est plus petit que 256.

6.4. Le processus de chiffrement des clefs « Key-encryption »

Les données du processus générant le chiffrement de clefs – la valeur supplée par l'algorithme de chiffrement de clef du destinataire – correspondent juste à la « valeur » de la clef de chiffrement du contenu.

N'importe quelles techniques de gestion de clefs susmentionnées peuvent être utilisées pour chaque destinataire du même contenu chiffré.

7. TYPE DE CONTENU : DONNEES COMPRESSEES (HASH) (« DIGESTED-DATA »)

Le type de contenu de données compressées (hash) (`digested-data`) se compose du contenu de n'importe quel type et des données compressées (hash) du contenu.

Typiquement, le type de contenu de données compressées (hash) est employé pour fournir l'intégrité du contenu, et le résultat devient généralement une entrée au type de contenu de données sous enveloppe (`enveloped-data`).

Les étapes suivantes construisent les données compressées (hash) :

1. des données compressées (hash) sont calculées à partir du contenu avec un algorithme de compression de données (hashage).
2. l'algorithme de compression de données (hashage) et les données compressées (hash) sont rassemblés ainsi que le contenu dans une valeur de `DigestedData`.

Un destinataire vérifie les données compressées (hash) en les comparant à des données compressées calculées indépendamment.

L'identifiant d'objet suivant spécifie le type de contenu `digested-data` :

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

Le type de contenu `digested-data` doit être au format ASN.1 pour le type `DigestedData` :

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

Digest ::= OCTET STRING
```

Les champs du type `DigestedData` ont les significations suivantes :

`version` est la syntaxe du numéro de version. Si le type du contenu encapsulé correspond à `id-data`, alors la valeur de la version DOIT être 0 ; cependant, si le type du contenu encapsulé est autre que `id-data`, alors la valeur de la version DOIT être 2.

`digestAlgorithm` identifie l'algorithme de compression de données (hashage) et

tous ses paramètres associés, pour lesquels le contenu est compressé. Le processus de compression de données est le même que celui de la section 5.4 dans le cas où il n'y a aucun attribut signé.

`encapContentInfo` est le contenu qui est compressé, comme défini dans la section 5.2.

`digest` est le résultat du processus de compression de données.

L'ordre des champ `digestAlgorithm`, `encapContentInfo` et `digest` rendent possible le traitementd'une valeur de `DigestedData` en une seule passe (*single pass*).

8. TYPE DE CONTENU : DONNEES CHIFFREES (« ENCRYPTED-DATA »)

Le type de contenu `encrypted-data` se compose du contenu chiffré de n'importe quel type. À la différence du type de contenu `enveloped-data`, le type de contenu `encrypted-data` n'a ni destinataires ni clefs chiffrées `content-encryption`. Les clefs DOIVENT être gérées par d'autres moyens.

L'application typique du type de contenu `encrypted-data` doit être de chiffrer le type de contenu `data` pour le stockage local, peut-être où la clef de chiffrement est dérivée d'un mot de passe.

L'identifiant d'objet suivant spécifie le type de contenu de `encrypted-data` :

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

Le type de contenu `encrypted-data` doit être au format ASN.1 pour le type `Encrypted-Data` :

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

Les champs du type `EncryptedData` ont les significations suivantes :

`version` est la syntaxe du numéro de version. Si `unprotectedAttrs` est présent, alors la version DOIT être 2. Si `unprotectedAttrs` est absent, alors la version DOIT être 0.

`encryptedContentInfo` est l'information du contenu chiffré, comme défini dans la section 6.1.

`unprotectedAttrs` est une collection d'attributs qui ne sont pas chiffrés. Le champ est facultatif. Des types utiles d'attribut sont définis dans la section 11.

9. TYPE DE CONTENU : DONNEES D'AUTHENTIFICATION (« AUTHENTICATED-DATA »)

Le type de contenu `Authenticated-data` se compose du contenu de n'importe quel type, d'un code d'authentification de message MAC (*Message Authentication Code*), et des clefs chiffrées d'authentification pour un ou plusieurs destinataires. La combinaison du MAC et d'une clef chiffrée d'authentification est nécessaire pour qu'un destinataire puisse vérifier l'intégrité du contenu. N'importe quel type de contenu peut être protégée en intégrité pour un nombre arbitraire de destinataires.

Le processus par lequel les `authenticated-data` sont construites implique les étapes suivantes :

1. Une clef `message-authentication` est produite aléatoirement par un algorithme particulier `message-authentication`.
2. La clef `message-authentication` est chiffrée pour chaque destinataire. Les détails de ce chiffrement dépendent de l'algorithme de gestion des clefs utilisé.
3. Pour chaque destinataire, la clef chiffrée `message-authentication` et toute autre information spécifique de chaque destinataire (*recipient-specific*) sont rassemblées dans la valeur `RecipientInfo`, définie dans la section 6.2.
4. En utilisant la clef `message-authentication`, l'expéditeur calcule une valeur du MAC à partir du contenu. Si l'expéditeur authentifie n'importe quelle information en plus du contenu (voir la section 9.2), les données compressées (hash) sont calculées à partir du contenu, ces données compressées du contenu ainsi que d'autres informations sont authentifiées en utilisant la clef `message-authentication`, et le résultat devient la « valeur MAC ».

9.1. Le type « authenticated-data »

L'identifiant d'objet suivant spécifie le type de contenu `authenticated-data` :

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

Le type de contenu `authenticated-data` doit être au format ASN.1 pour le 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
```

Les champs du type `AuthenticatedData` ont les significations suivantes :

`version` est la syntaxe du numéro de version. La version DOIT être déterminée selon les situations suivantes :

SI ((`originatorInfo` est présents)

ET

(tous les attributs de certificats de version 2 sont présents))

ALORS la version DOIT être 1

SINON la version DOIT être 0

`originatorInfo` fournit sur option des informations au sujet de l'expéditeur. Ce champ est présent seulement si l'algorithme de gestion des clefs en a besoin. Il PEUT contenir des certificats, des attributs de certificats, et des CRLs, comme défini dans la section 6.1.

`recipientInfos` est une collection d'information sur chaque destinataire (*per-recipient*), comme définie dans la section 6.1. Il DOIT y avoir au moins un élément dans la collection.

`macAlgorithm` est un identifiant de l'algorithme du code d'authentification de message (MAC). Il identifie l'algorithme du MAC, avec tous ses paramètres associés, utilisés par l'expéditeur. La mise en place du champ `macAlgorithm` facilite le traitement en un passage simple (*one-pass*) par le destinataire.

`digestAlgorithm` identifie l'algorithme de compression de données (hashage), et tous ses paramètres associés, utilisés pour calculer les données compressées sur le contenu encapsulé dans le cas où les attributs authentifiés sont présents. Le processus de compression de données est décrit dans la section 9.2. La mise en place du champ `digestAlgorithm` facilite le traitement en un passage simple (*one-pass*) par le destinataire. Si le champ `digestAlgorithm` est présent, alors le champ `authAttrs` DOIT l'être également.

`encapContentInfo` est le contenu qui est authentifié, comme défini dans la section 5.2.

`authAttrs` est une collection d'attributs authentifiés. La structure `authAttrs` est facultative, mais elle DOIT être présente si le type de contenu de la valeur `EncapsulatedContentInfo` à authentifier n'est pas la même que celle d'`id-`

data. Si le champ authAttrs est présent, alors le champ digestAlgorithm DOIT l'être également. La structure AuthAttributes DOIT être codée au format DER, même si le reste de la structure est codé au format BER. Des types d'attributs utiles sont définis dans la section 11. Si le champ authAttrs est présent, il DOIT contenir, au minimum, les deux attributs suivants :

Un attribut content-type ayant comme valeur le type de contenu de la valeur EncapsulatedContentInfo à authentifier. La section 11.1 définit l'attribut content-type.

Un attribut de données compressées (message-digest), en ayant en tant que sa valeur le sommaire de message du contenu. La section 11.2 définit l'attribut message-digest.

mac est le code d'authentification de message.

unauthAttrs est une collection d'attributs qui ne sont pas authentifiés. Le champ est facultatif. Jusqu'ici, aucun attribut n'a été défini pour l'usage d'attributs non authentifiés, mais d'autres types d'attributs utiles sont définis dans la section 11.

9.2. La génération d'un MAC

Le processus de génération d'un MAC calcule un code d'authentification de message (MAC) soit à partir du contenu à authentifier soit à partir des données compressées du contenu à authentifier ainsi que des attributs authentifiés de l'expéditeur.

Si le champ authAttrs est absent, les données issues du processus de calcul du MAC, correspondent à la valeur de encapContentInfo → eContent → OCTET STRING. Seuls les octets contenus dans la valeur de eContent → OCTET STRING sont pris en compte dans l'algorithme du MAC ; l'étiquette (*tag*) et la taille des octets ne sont pas pris en compte. Ceci a l'avantage que la taille du contenu à authentifier n'a pas besoin d'être connue avant le processus de génération du MAC.

Si le champ authAttrs est présent, l'attribut de content-type (comme décrit dans section 11.1) et l'attribut de message-digest (comme décrit dans section 11.2) DOIVENT être inclus, et les données issues du processus de calcul du MAC correspondent à la valeur du champ authAttrs, encodé au format DER. Un codage séparé du champ authAttrs est effectué pour le calcul de données compressées (hash). L'étiquette IMPLICIT [2] dans le champ authAttrs n'est pas utilisée pour l'encodage DER, l'étiquette EXPLICIT SET OF est plutôt utilisée. C'est-à-dire que l'encodage au format DER de l'étiquette SET OF, plutôt que celui de l'étiquette IMPLICIT [2], doit être inclus dans le calcul des données compressées avec la taille et le contenu des octets de la valeur authAttrs.

Le processus de génération de données compressées calcule les données compressées à partir du contenu à authentifier. Les données initiales du processus de calcul de données compressées correspondent à la « valeur » du contenu encapsulé à authentifier. Spécifiquement, les données correspondent à la valeur du champ encapContentInfo → eContent → OCTET STRING à partir de laquelle le processus d'authentification est appliqué. Seuls les octets compris dans la valeur du champ encapContentInfo → eContent → OCTET STRING sont injectés dans l'algorithme de compression de données (hashage), mais pas l'étiquette ni la taille des octets. Ceci a l'avantage que la taille du contenu à authentifier n'a pas besoin d'être connue à l'avance. Bien que l'étiquette et la

taille des octets du champ encapContentInfo → eContent → OCTET STRING ne soient pas incluses dans le calcul de compression de données, elles sont encore protégés par d'autres moyens. La taille des octets est protégée par la nature de l'algorithme de compression de données puisqu'il est informatiquement infaisable de trouver deux contenus distincts de taille quelconque qui auraient les mêmes données compressées.

Les données issues du processus de génération d'un MAC incluent les données d'entrée du MAC, définies ci-dessus, et une clef d'authentification donnée dans une structure de recipientInfo. Les détails du calcul d'un MAC dépendent de l'algorithme du MAC utilisé (par exemple, HMAC). L'identifiant d'objet, avec tous ses paramètres, qui spécifie l'algorithme de MAC utilisé par l'expéditeur, est transporté par le champ macAlgorithm. La valeur du MAC générée par l'expéditeur est codée en chaîne d'octets (OCTETS STRING) et transporté dans le champ mac.

9.3. La vérification d'un MAC

Les données du processus de vérification d'un MAC incluent les données d'entrée (déterminées sur la présence ou l'absence du champ authAttrs, comme définie dans la section 9.2), et la clef d'authentification donnée dans le champ recipientInfo. Les détails du processus de vérification d'un MAC dépendent de l'algorithme du MAC utilisé.

Le destinataire NE DOIT compter sur AUCUNE valeur de MAC ou aucune valeurs de compression de données calculées par l'expéditeur. Le contenu est authentifié comme il l'est décrit dans la section 9.2. Si l'expéditeur inclut des attributs d'authentification, alors la teneur des authAttrs est authentifiée comme décrit dans la section 9.2. Pour que l'authentification réussisse, la valeur du MAC calculée par le destinataire DOIT être identique à la valeur du champ mac. De même, pour que l'authentification réussisse lorsque le champ authAttrs est présent, la valeur des données compressées calculées par le destinataire DOIVENT être identiques à la valeur des données compressées incluses dans les attributs de authAttrs → message-digest.

Si le champ AuthenticatedData inclut de champ authAttrs, alors la valeur d'attribut content-type DOIT correspondre avec la valeur AuthenticatedData → encapContentInfo → eContentType.

10. TYPES UTILES

Cette section est divisée en deux parties. La première partie définit des identifiants d'algorithme, et la deuxième partie définit d'autres types utiles.

10.1. Les types d'identifiant d'algorithmes

Toutes les identifiants d'algorithmes ont le même type : AlgorithmIdentifier. La définition d'AlgorithmIdentifier est issue de la norme X.509 [X.509-88].

Il y a beaucoup de variantes pour chaque type d'algorithme.

10.1.1.« DigestAlgorithmIdentifier »

Le type DigestAlgorithmIdentifier identifie l'algorithme de compression de données (hashage).SHA-1, MD2 et MD5 sont des exemples les plus connus. Un algorithme de compression de données (hashage) « mappe » (mappage = succession de transposition et d'inversion de caractères) une chaîne d'octets (le contenu) dans une autre chaîne d'octets (les

données compressées ou le hash).

```
DigestAlgorithmIdentifier::= AlgorithmIdentifier
```

10.1.2.« SignatureAlgorithmIdentifier »

Le type `SignatureAlgorithmIdentifier` spécifie un algorithme de signature. RSA, DSA et ECDSA sont des exemples les plus connus. Un algorithme de signature comporte les opérations de génération et de vérification de signature. L'opération de génération de signature utilise les données compressées et la clef privée du signataire pour générer la valeur d'une signature. L'opération de vérification de signature utilise les données compressées et la clef publique du signataire pour déterminer si une valeur de signature est valide. Le contexte détermine quelle opération est attendue.

```
SignatureAlgorithmIdentifier::= AlgorithmIdentifier
```

10.1.3.« KeyEncryptionAlgorithmIdentifier »

Le type `KeyEncryptionAlgorithmIdentifier` spécifie un algorithme de chiffrement de clef (`key-encryption`) utilisé pour chiffrer une clef `content-encryption`. L'opération de chiffrement « mappe » une chaîne d'octets (la clef) dans une autre chaîne d'octets (la clef chiffrée) sous la commande d'une clef `key-encryption`. L'opération de déchiffrement est l'inverse de celle du chiffrement. Le contexte détermine quelle opération est attendue.

Les détails du chiffrement et du déchiffrement dépendent de l'algorithme de gestion des clefs, utilisé. Cet algorithme doit donc prendre en comte les clefs de transport (*key transport*), les clefs d'approbation (*key agreement*) et les clefs de chiffrement symétriques (*symmetric key-encrypting*), celles préalablement distribuées ou provenant de mot de passe.

```
KeyEncryptionAlgorithmIdentifier::= AlgorithmIdentifier
```

10.1.4.« ContentEncryptionAlgorithmIdentifier »

Le type `ContentEncryptionAlgorithmIdentifier` spécifie un algorithme de chiffrement de contenu (`content-encryption`). Le Triple-DES et RC2 sont des exemples les plus connus. Un algorithme de chiffrement de contenu prend en compte les opérations de chiffrement et de déchiffrement. L'opération de chiffrement « mappe » une chaîne d'octets (le texte plein - *plaintext*) dans une autre chaîne d'octets (le texte chiffré - *ciphertext*) sous la commande d'une clef de chiffrement de contenu (*content-encryption key*). L'opération de déchiffrement est l'inverse de l'opération de chiffrement. Le contexte détermine quelle opération est attendue.

```
ContentEncryptionAlgorithmIdentifier::= AlgorithmIdentifier
```

10.1.5.« MessageAuthenticationCodeAlgorithm »

Le type `MessageAuthenticationCodeAlgorithm` spécifie un algorithme de code d'authentification de message (MAC). DES-MAC ou HMAC-SHA-1 sont des exemples les plus connus. Un algorithme de MAC prend en compte les opérations de génération et de vérification. Les opérations de génération et de vérification de MAC utilisent la même clef symétrique. Le contexte détermine quelle opération est attendue.

```
MessageAuthenticationCodeAlgorithm::= AlgorithmIdentifier
```

10.1.6.« KeyDerivationAlgorithmIdentifier »

Le type `KeyDerivationAlgorithmIdentifier` est indiqué dans la RFC 3211 [**PWRI**]. La définition de `KeyDerivationAlgorithmIdentifier` est répétée ici pour plus de clarté.

Les algorithmes de dérivation de clefs (*Key derivation*) convertissent un mot de passe ou la valeur d'un secret partagé en une clef de chiffrement (*key-encryption key*).

```
KeyDerivationAlgorithmIdentifier::= AlgorithmIdentifier
```

10.2. D'autres types utiles

Cette section définit des types qui sont utilisés dans d'autres endroits de ce document. Ces types ne sont pas énumérés dans un ordre particulier.

10.2.1.« CertificateRevocationLists »

Le type `CertificateRevocationLists` donne un ensemble de listes de révocation de certificat (CRLs). Il est bien entendu que l'ensemble contienne suffisamment d'information pour déterminer si les certificats et les attributs de certificats auxquels l'ensemble est associé soient révoqués. Cependant, il peut y avoir plus de CRLs que nécessaire ou il PEUT y avoir moins de CRLs que nécessaire.

Le champ `CertificateList` peut contenir une CRL, une liste de révocation d'autorités - ARL (*Authority Revocation List*), une Delta CRL, ou une liste de révocation d'attributs de certificat. Toutes ces listes partagent une syntaxe commune.

Les CRLs sont spécifiées dans la norme X.509 [**X.5009-97**] et leurs profils pour Internet se rapportent à la RFC 3280 [**PROFILE**].

La définition de CertificateList est tirée de la norme X.509.

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

10.2.2.« CertificateChoices »

Le type `CertificateChoices` fournit soit un certificat étendu PKCS #6 [**PKCS#6**], soit un certificat X.509, soit un certificat d'attribut X.509 de la version 1 (ACv1) [**X.509-97**], ou

de la version 2 (ACv2) [**X.509-00**]. PKCS #6 et ACv1 sont obsolètes, ils font partie des compatibilités antérieures. Les certificats étendu PKCS #6 et les certificats d'attribut X.509 de la version 1 (ACv1) NE DEVRAIENT PAS être employés. Le profil d'Internet pour les certificats X.509 est spécifié dans l'« infrastructure à clefs publiques de l'Internet X.509 : Certificats et profils sur les CRL » (« *Internet X.509 Public Key Infrastructure: Certificate and CRL Profile* ») [**PROFILE**]. Le profil d'Internet pour l'utilisation des ACv2 est spécifié dans le « profil de certificat d'attributs pour l'autorisation dans Internet » (« *An Internet Attribute Certificate Profile for Authorization* ») [**ACPROFILE**].

La définition de `Certificate` est tirée de la norme X.509.

Les définitions d'`AttributeCertificate` sont tirées des normes X.509-1997 et X.509-2000. La définition de X.509-1997 est assignée à `AttributeCertificateV1` (voir la section 12.2), et la définition de X.509-2000 est assignée à `AttributeCertificateV2`.

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

10.2.3. « CertificateSet »

Le type `CertificateSet` fournit un ensemble de certificats. Il est bien entendu que le nombre de certificats dans cet ensemble doit être suffisant pour contenir des chaînes hiérarchiques de certification depuis l'autorité racine ou autorité supérieure de certification reconnue, jusqu'à tous les expéditeurs dont les certificats sont associés à cet ensemble. Cependant, il peut y avoir plus de certificats que nécessaires, ou il PEUT y en avoir moins que nécessaire.

La signification précise d'une « chaîne hiérarchique de certification » est en dehors du champ de ce document. Quelques applications peuvent imposer des limites supérieures à la longueur d'une chaîne ; d'autres peuvent imposer certains liens entre le contenu des certificats et leurs émetteurs dans une chaîne.

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

10.2.4. « IssuerAndSerialNumber »

Le type `IssuerAndSerialNumber` identifie un certificat, et de ce fait une entité et une clef publique, grâce au nom distingué de l'émetteur du certificat et d'un numéro de série spécifique de l'émetteur (*issuer-specific*) du certificat.

La définition de `Name` est tirée de la norme X.501 [**X.501-88**], et la définition de `CertificateSerialNumber` est tirée de la norme X.509 [**X.509-97**].

```
IssuerAndSerialNumber ::= SEQUENCE {
```

```
   issuer Name,
   serialNumber CertificateSerialNumber }

CertificateSerialNumber ::= INTEGER
```

10.2.5. « CMSVersion »

Le type `CMSVersion` fournit le numéro de version de la syntaxe, pour la compatibilité avec de futures révisions de ces spécifications.

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

10.2.6. « UserKeyingMaterial »

Le type `UserKeyingMaterial` fournit une syntaxe pour les utilisateurs de clefs matérielles – UKM (*User Keying Material*). Quelques algorithmes générant des clefs d'approbation (*key agreement*) exigent d'UKMs de s'assurer qu'une clef différente est produite à chaque fois que les mêmes deux parties (expéditeur et destinataire) produisent une clef commune confidentielle (*pairwise key*). L'expéditeur fournit un UKM à utiliser avec un algorithme spécifique générant la clef d'approbation.

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

10.2.7. « OtherKeyAttribute »

Le type `OtherKeyAttribute` fournit une syntaxe pour l'inclusion d'autres attributs de clefs qui permettent au destinataire de choisir la clef employée par l'expéditeur. L'identifiant d'objet d'attribut doit être enregistré avec la syntaxe de l'attribut lui-même. L'utilisation de cette structure devrait être évitée puisqu'elle pourrait empêcher l'interopérabilité.

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

11. ATTRIBUTS UTILES

Cette section définit les attributs qui peuvent être employés avec les données signées (`signed-data`), les données mises sous enveloppe (`enveloped-data`), les données chiffrées (`encrypted-data`), ou les données d'authentification (`authenticated-data`). La syntaxe de l'attribut est compatible avec la norme X.501 [**X.501-88**] et la RFC 3280 [**PROFILE**]. Certains des attributs définis dans cette section, ont été à l'origine définis dans PKCS #9 [**PKCS#9**] ; d'autres ont été à l'origine définis dans une version précédente de ces spécifications [**OLDCMS**]. Ces attributs ne sont pas énumérés dans un ordre particulier.

Des attributs supplémentaires sont définis dans beaucoup de cas, notamment dans les spécifications de message de la version 3 de S/MIME [**MSG**] et les services de sécurité augmentés pour S/MIME [**ESS**] (*Enhanced Security Services*), qui incluent également des recommandations concernant la mise en place de ces attributs.

11.1. Le type « content »

Le type d'attribut de `content-type` indique le type de contenu de `ContentInfo` dans le champ `signed-data` ou `authenticated-data`. Le type d'attribut `content-type` DOIT être présent même lorsque les attributs signés sont présents dans les `signed-data` ou dans les attributs d'authentification présents dans le champ `authenticated-data`. La valeur d'attribut de `content-type` DOIT correspondre avec la valeur `eContentType` → `encapContentInfo` dans le champ `signed-data` ou `authenticated-data`.

L'attribut `content-type` DOIT être un attribut signé ou un attribut authentifié ; il NE DOIT PAS être un attribut non signé, un attribut non authentifié, ou un attribut non protégé.

L'identifiant d'objet suivant spécifie l'attribut de `content-type` :

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

Les valeurs d'attribut de `content-type` sont sous la forme ASN.1 pour le type `ContentType` :

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

Bien que la syntaxe soit définie comme `SET OF AttributeValue`, un attribut `content-type` DOIT avoir une valeur simple d'attribut ; des instances nulle (zéro) ou multiples d'`AttributeValue` ne sont pas autorisés.

Les syntaxes `SignedAttributes` et `AuthAttributes` sont chacune définie comme `SET OF Attributes`. Le champ `SignedAttributes` dans `signerInfo` NE DOIT PAS inclure des instances multiples de l'attribut `content-type`. De même, le champ `AuthAttributes` dans `AuthenticatedData` NE DOIT PAS inclure des instances multiples de l'attribut `content-type`.

11.2. Les données compressées (hash) (Message Digest)

Le type d'attribut `message-digest` spécifie les données compressées dans le champ `encapContentInfo` → `eContent` → `OCTET STRING` à signer du champ `signed-data` (voir la section 5.4) ou authentifiée dans le champ `authenticated-data` (voir la section 9.2). Pour `signed-data`, les données compressées sont calculées en utilisant l'algorithme de compression des données du signataire. Pour `authenticated-data`, les données compressées sont calculées en utilisant l'algorithme de compression des données de l'expéditeur.

Dans le champ signed-data, le type d'attribut signé message-digest DOIT être présent lorsque tous les attributs signés sont présents. Dans le champ authenticated-data, le type d'attribut authentifié message-digest DOIT être présent lorsque tous les attributs authentifiés sont présents. L'attribut message-digest DOIT être un attribut signé ou un attribut authentifié ; il NE DOIT PAS être un attribut non signé, ni un attribut non authentifié, ni un attribut non protégé.

L'identifiant d'objet suivant spécifie l'attribut message-digest :

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

Les valeurs d'attribut de Message-digest sont sous la forme ASN.1 pour le type MessageDigest :

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

Un attribut de Message-digest DOIT avoir une valeur simple d'attribut, même si la syntaxe est définie comme SET OF AttributeValue. Il NE DOIT PAS y avoir d'instances nulle (zéro) ou multiples d'AttributeValue présentes. Les syntaxes de SignedAttributes et d'AuthAttributes sont, toutes les deux, définies comme SET OF Attributes. Le champ SignedAttributes dans signerInfo DOIT inclure seulement une instance de l'attribut message-digest. De même, le champ AuthAttributes dans AuthenticatedData DOIT inclure seulement une instance de l'attribut message-digest.

11.3. L'horodatage (Signing Time)

Le type d'attribut signing-time spécifie l'instant où le signataire a (soi-disant) exécuté le processus de signature. Le type d'attribut signing-time est prévu d'être utilisé dans les signed-data.

L'attribut signing-time DOIT être un attribut signé ou un attribut authentifié ; il NE DOIT PAS être un attribut non signé, ni un attribut non authentifié, ni un attribut non protégé.

L'identifiant d'objet suivant spécifie l'attribut signing-time :

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

Les valeurs d'attribut signing-time sont sous la forme ASN.1 pour le type SigningTime :

```
SigningTime ::= Time
```

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

Note : La définition du temps doit correspondre avec celle spécifiée dans la norme X.509 de la version de 1997 [X.509-97].

Des dates entre le 1er janvier 1950 et le 31 décembre 2049 (incluses) DOIVENT être codées au format `UTCTime`. Toutes les dates avec des valeurs d'année avant 1950 ou après 2049 DOIVENT être codés au format `GeneralizedTime`.

Des valeurs en `UTCTime` DOIVENT être exprimées en temps moyen de Greenwich (`Zulu`) et DOIVENT inclure des secondes (c.-à-d., les temps sont sous la forme : `YYMMDDHHMMSSZ`), même si le nombre de secondes est nul. Le minuit (`GMT`) DOIT être représenté sous la forme : `YYMMDD000000Z`. L'information sur le siècle est implicite et le siècle DOIT être déterminé comme suit :

si `YY` est supérieur ou égal à 50, l'année DOIT être interprétée comme `19YY` ;

si `YY` est inférieur à 50, l'année DOIT être interprétée comme `20YY`.

Les valeurs en `GeneralizedTime` DOIVENT être exprimées en temps moyen de Greenwich (`Zulu`) et DOIVENT inclure des secondes (c.-à-d., les temps sont sous la forme : `YYYYMMDDHHMMSSZ`), même si le nombre de secondes est nul. Les valeurs en `GeneralizedTime` NE DOIVENT PAS inclure des secondes partielles.

Un attribut de `signing-time` DOIT avoir une valeur d'attribut simple, même si la syntaxe est définie comme `SET OF AttributeValue`. Il NE DOIT PAS y avoir des instances nulles (zéro) ou multiples d'`AttributeValue` présent.

Les syntaxes `SignedAttributes` et `AuthAttributes` sont, toutes les deux, définies comme `SET OF Attributes`. Le champ `SignedAttributes` dans `signerInfo` NE DOIT PAS inclure des instances multiples de l'attribut `signing-time`. De même, le champ `AuthAttributes` dans `AuthenticatedData` NE DOIT PAS inclure des instances multiples de l'attribut `signing-time`.

Aucune condition n'est imposée au sujet de l'exactitude du temps de signature, et de l'acceptation d'un temps prétendu de signature qui est une question de discrétion d'un destinataire. Il est prévu, cependant, que quelques signataires, tels que des serveurs d'horodatage, seront implicitement de confiance.

11.4. Les « Countersignature »

Le type d'attribut `countersignature` spécifie une ou plusieurs signatures à partir des octets du contenu encodé au format DER du champ `signatureValue` d'une valeur `SignerInfo` dans `signed-data`. Ainsi, le type d'attribut de `countersignature` contresigne (signe en série) une autre signature.

L'attribut `countersignature` DOIT être un attribut non signé ; il NE DOIT PAS être un attribut signé, authentifié, non authentifié, ou non protégé.

L'identifiant d'objet suivant spécifie l'attribut de `countersignature` :

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

Les valeurs d'attribut `Countersignature` sont sous la forme ASN.1 pour le type `Countersignature` :

```
Countersignature ::= SignerInfo
```

Les valeurs `Countersignature` ont les mêmes significations que les valeurs `SignerInfo` pour les signatures ordinaires, sauf dans les cas suivants :

1. Le champ `signedAttributes` NE DOIT PAS contenir un attribut `content-type` ; il n'y a aucun type de contenu pour des `countersignatures`.
2. Le champ `signedAttributes` DOIT contenir l'attribut `message-digest` s'il contient n'importe quels autres attributs.
3. Les données du processus de compression de données (`message-digesting`) correspondent aux octets du contenu encodés au format DER du champ `signatureValue` de la valeur de `SignerInfo` à laquelle l'attribut est associé.

Un attribut `countersignature` peut avoir des valeurs d'attributs multiples. La syntaxe est définie comme `SET OF AttributeValue` et il DOIT y avoir une ou plusieures instances d'`AttributeValue` présentes.

La syntaxe `UnsignedAttributes` est définie comme `SET OF Attributs`. L'`UnsignedAttributes` dans un `signerInfo` peut inclure des instances d'attributs multiples de `countersignature`.

Un `countersignature`, puisqu'il a le type `SignerInfo`, peut lui-même contenir un attribut de `countersignature`. Ainsi, il est possible de construire une longue série arbitraire de `countersignatures`.

12. MODULES ASN.1

La section 12.1 contient le module ASN.1 pour le CMS, et la section 12.2 contient le module ASN.1 pour les attributs de certificat de la version 1.

12.1. Le module ASN.1de la CMS

```
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 mai 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. Le module ASN.1des attributs de certificat version 1

```
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, Avril 2002.

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

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

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

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

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

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

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

[**PKCS#6**] RSA Laboratories. PKCS #6: Extended-Certificate Syntax Standard, Version 1.5. Novembre 1993.

[**PKCS#7**] Kaliski, B., "PKCS #7: Cryptographic Message Syntax, Version 1.5.", RFC 2315, Mars 1998.

[**PKCS#9**] RSA Laboratories. PKCS #9: Selected Attribute Types, Version 1.1. Novembre 1993.

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

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

[**STDWORDS**] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, Mars 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. CONSIDERATIONS DE SECURITE

La syntaxe cryptographique de message fournit une méthode pour signer numériquement, compresser (hasher), chiffrer et authentifier des données.

Les implémentations doivent protéger la clef privée du signataire. La compromission de la clef privée du signataire permet la masquerade.

Les implémentations doivent protéger la clef de gestion (*key management*) de la clef privée (*private key*), la clef de chiffrement (*key-encryption key*) et la clef de chiffrement du contenu (*content-encryption key*). La compromission de la clef de gestion de la clef privée ou de la clef de chiffrement peut avoir comme conséquence la révélation de tout le contenu protégé avec cette clef. De même, la compromission de la clef de chiffrement du contenu peut avoir comme conséquence la révélation du contenu chiffré associé.

Les implémentations doivent protéger la clef de gestion de la clef privée et la clef d'authentication des données. La compromission de la clef de gestion de la clef privée permet la masquerade des données authentifiées. De même, la compromission de la clef d'authentication des données peut avoir comme conséquence la modification indétectable du contenu authentifié.

La technique de gestion des clefs utilisée pour distribuer des clefs d'authentification de données doit elle-même fournir l'authentification d'origine des données, autrement le contenu est fourni avec l'intégrité d'une source inconnue. Ni le RSA [PKCS#1, NEWPKCS#1] ni Diffie-Hellman Ephemeral-Static [DH-X9.42] ne fournissent l'authentification nécessaire de l'origine des données. Diffie-Hellman Ephemeral-Static [DH-X9.42] fournit l'authentification nécessaire de l'origine des données seulement lorsque les clefs publiques de l'expéditeur et du destinataire sont liées aux identités appropriées des certificats X.509.

Lorsque plus de deux parties partagent la même clef d'authentication de données, l'authentification de l'origine des données n'est pas disponible. N'importe quelle partie qui connait la clef d'authentication des données peut calculer un MAC valide, donc le contenu pourrait provenir de n'importe la quelle des parties.

Les implémentations doivent produire aléatoirement les clefs de chiffrement du contenu, les clefs d'authentication des données, les vecteurs d'initialisation (IVs), et le remplissage (padding). En outre, la génération de biclefs publique/privée se fonde sur des nombres aléatoires. L'utilisation des générateurs pseudo-aléatoires de nombre (PRNGs) (*pseudo-random number generators*) inadéquates produisant des clefs cryptographiques peut avoir comme conséquence peu ou pas de sécurité. Un attaquant peut la trouver beaucoup plus facilement pour reproduire l'environnement PRNG qui a produit les clefs, en recherchant un petit nombre de résultats possibles, plutôt que d'effectuer une attaque en « force brute » en recherchant l'espace entier des clefs. La génération des nombres aléatoires de qualité est difficile. La RFC 1750 [**RANDOM**] offre des conseils importants dans ce secteur, et l'annexe 3 de FIPS Pub 186 [**DSS**] fournit une technique de la qualité PRNG.

En utilisant les algorithmes de génération de clefs d'approbation (*key agreement*) ou des clefs symétriques de chiffrement de clef (*symmetric key-encryption keys*), préalablement distribuées, une clef de chiffrement de clef (*key-encryption key*) est utilisée pour chiffrer la clef de chiffrement de contenu (*content-encryption key*). Si des algorithmes générant la clef de chiffrement de clef et générant la clef de chiffrement de contenu, sont différents, la sécurité la plus efficace est déterminée par le plus faible des deux algorithmes. Si, par

exemple, le contenu est chiffré avec du Triple-DES en utilisant une clef de chiffrement de contenu, utilisant leTriple-DES-168-bit, et cette clef de chiffrement de contenu est enveloppée avec l'algorithme RC2 en utilisant une clef de chiffrement de clef de 40-bit RC2, alors, tout au plus, 40 bits de protection sont assurés. Une recherche triviale pour déterminer la valeur de la clef de 40-bit RC2 peut permettre de récupérer la clef Triple-DES qui permet ainsi de déchiffrer le contenu. Par conséquent, les personnes qui implémentent ces algorithmes doivent s'assurer que les algorithmes de chiffrement de clefs sont aussi forts voire plus forts que les algorithmes de chiffrement de contenu.

Les personnes qui implémentent ces algorithmes devraient se rendre compte que les algorithmes cryptographiques deviennent plus faibles avec le temps. Alors que de nouvelles techniques de cryptanalyse sont développées mettant à profit la performance croissante de calcul des ordinatateurs, le facteur « travail » pour casser un algorithme cryptographique particulier sera réduit. Par conséquent, les implémentations cryptographiques d'algorithme devraient être modulaires, permettant à de nouveaux algorithmes d'être aisément insérés. C'est-à-dire que les personnes, qui implémentent ces algorithmes, devraient être préparés à utiliser un jeu d'algorithmes qui doit être implémenté pour un temps fini.

L'attribut non signé de `countersignature` inclut une signature numérique qui est calculée sur la valeur de la signature du contenu, ainsi le processus de contre signature n'a pas besoin de connaître le contenu signé d'origine. Cette structure permet des avantages d'efficacité d'exécution ; cependant, cette structure peut également permettre de contre signer une valeur de signature inadéquate. Par conséquent, les implémentations qui exécutent des countresignatures devraient vérifier la valeur de la signature d'origine avant de la contresigner (cette vérification exige le traitement du contenu d'origine), ou bien, des implémentations devraient exécuter la contresignature dans un contexte qui s'assure que seulement des valeurs appropriées de signature sont contresignées.

15. REMERCIEMENTS

Ce document est le résultat des contributions de beaucoup de professionnels. M. Russel Housley (auteur de cette RFC 3369) apprécie le travail conséquent de tous les membres du groupe de travail de S/MIME de l'IETF. R. Housley remercie spécialement Ankney riche, Simon Blake-Wilson, Tim Dean, Steve Dusse, Carl Ellison, Peter Gutmann, Bob Jueneman, Stephen Henson, Paul Hoffman, Scott Hollenbeck, mets Johnson, Burt Kaliski, John Linn, John Pawling, Blake Ramsdell, Francois Rousseau, Jim Schaad, et Dave Solo pour leurs efforts et leur soutien.

16. L'ADRESSE DE L'AUTEUR

Russell Housley

RSA Laboratories

918 Spring Knoll Drive

Herndon, VA 20170

USA

Email : rhousley@rsasecurity.com

17. COPYRIGHT

Email : macario.olivier@free.fr