KDM : AUTHENTICATED PRIVATE

PRÉFACE

L'**Authenticated Private** est le bloc où les données ne peuvent être lues que par le détenteur de la clef privée RSA.

On trouvera les **éléments centraux du KDM** : c'est ici où sont **stockés les clefs AES** qui nous permettront de déchiffrer les MXF chiffrés de notre DCP.

La partie **AuthenticatedPrivate** respecte la norme **XML Encryption** (xmlenc) archive

A L'INTÉRIEUR D'AUTHENTICATEDPRIVATE

Reprenons la partie **AuthenticatedPrivate** de notre précédent KDM :

```
<AuthenticatedPrivate Id="ID_AuthenticatedPrivate" xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
  <enc:EncryptedKey>
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
      <ds:DigestMethod xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </enc:EncryptionMethod>
    <enc:CipherData>
      <enc:CipherValue>MEXikAS/9WQTEG1mZs8gBICWym2OciZCq7hR0YCdsPr6jvPMeASr
mU0WxFbUHpkUASZa737KfgPo/XB2VXt2exUUvskbLetkoMlpICeLU5/JdNPBhWPbphDXAXqzO8wE
riNFSmUQaKd5ds5Z5nPDhfTsPX70vfrqX6HeUz6aX676/D2GfuZyhxnamLYYJprxbnTTridP5WUs
8JM8L5qEwqdxLZK0UoYEW9gRpfg3iYpBr8dqIdLC2kvGzaEuKU0F0kOHiTANlwnXKEZkRLcHBlti
5O0RN9Tzce+t/D1LZG2MvvzwVnWNYoyD/ozuww+wISOC5TOQwAUgPg2IrKIW1A==</enc:CipherValue>
    </enc:CipherData>
  </enc:EncryptedKey>
  <enc:EncryptedKey>
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
      <ds:DigestMethod xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </enc:EncryptionMethod>
    <enc:CipherData>
      <enc:CipherValue>SlaeGHfhbm2YgVosrN2qSLsxz+K7N7t09YGWxh3+Ud3HtBElFap5
ayhGBFgPEhnpof3XBsAxbW73C+D/GcXWHkCNEFIaDbYG6cL8EyWIwYeoYstdAPVjLatj4L0InD4H
DV4YeM5xnTpoqMHYS5ICSr0SlJYyHzCLp+Plj0Fv/IS0y9CXTrUisX+rFM8ivLdOOOZoBoDn4jyD
eKBm38Fcw9fTpve8hDFR5syGXM4bc5S17dUfO+IWcwJlSlq1GpTa3GrZ6Hpdse35GJRm3utk1ma1
VTJ602hjAMu+mjRWbBPFbfaUlmh0iTdCIjSfFo78ChGoJpHdTENCxbDaKC0cpw==</enc:CipherValue>
    </enc:CipherData>
  </enc:EncryptedKey>
  <enc:EncryptedKey>
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
      <ds:DigestMethod xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </enc:EncryptionMethod>
    <enc:CipherData>
      <enc:CipherValue>PmqCZD7E45n0Tor7cFJIZnI5sutG1wcgzsMj4wENiQYyiutYB8SH
7Jc2v8yKwVJGvTUurz2+kimRUb3pDEVW8BneQBZfdmx3McvUE9ZftTHTKkpOXJQmhtoEVor+6NcG
HSo4K+crPBbuRmJXmFPh1Nh+0jFCz1KW6MsMFgu1yNGBs54I6q1OwJXsX6vnYsLst6zYfxwSfpcK
OPJBebpfpe3L1xjpDe7iteoGg+yIObmunkBKvNXKf5rtIVUJ7mVkAnqHIUZpOd25S4dS6SOPW88k
mGtBDVClC+t1GYmgRk82VW6FoN7tU1HE6VOVHO61X0AQw3GUqIU2cGQAB9AAmw==</enc:CipherValue>
    </enc:CipherData>
  </enc:EncryptedKey>
  <enc:EncryptedKey>
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
      <ds:DigestMethod xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </enc:EncryptionMethod>
    <enc:CipherData>
      <enc:CipherValue>mNZp/GPjPaWDIzmbBvF1XcX5AoXVcYUOdeXkAcdUTHK+YxNrqm0N
HZoSp7watPwjFHS+FwcH5r62SDHf9hbWlLI9EBakj6xXQ1EkStd/i7gviUA6LJhAnTjjlMTdWclj
v5Dqto6yOZwg/yZbo+ea87GoscZvFO4BuMuwUic/U1iPjYg3eM87r8VsH9+MqriV/8XeJ8xnn38c
r6mI5/TbHf53Slu+Ft4dk49DsN6MMHos5WWr7J6GFtNRJICa9F7IrXMiCqG9CZxw7Zlt8Z90l5uz
4m3lip/a2DjuiruUKD/XAz4uSCGCMYD3Q8eUw5tpLKkPmKqlY1K4pkSbueuXKg==</enc:CipherValue>
    </enc:CipherData>
  </enc:EncryptedKey>
</AuthenticatedPrivate>
```

Comme vous le voyez, nous avons une liste de plusieurs **EncryptedKey** comprenant l'élément le plus important : la **CipherValue**.

C'est cette dernière qui va intégrer nos 4 clefs AES (avec ces métadonnées).

LES ENCRYPTEDKEYS ET LEUR CIPHERVALUE

Voici un exemple d'un seul **EncryptedKey** :

```
<enc:EncryptedKey>
  <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
    <ds:DigestMethod xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  </enc:EncryptionMethod>
  <enc:CipherData>
    <enc:CipherValue>UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUTOhUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==</enc:CipherValue>
  </enc:CipherData>
</enc:EncryptedKey>
```

Notre intérêt se portera sur la donnée dans **CipherValue** qui est notre contenu principal contenant notre clef AES et des métadatas.

La partie **EncryptedKey** est normalisée dans la section XML Encryption - EncryptedKey.

A l'intérieur, nous avons deux tags principaux :

- **EncryptionMethod** avec un algorithme défini à **RSA-OAEP-MGF1** (w3c-xmlenc)
- **DigestMethod** avec un algorithme défini à **SHA1**

La méthode principale pour le chiffrement est l'algorithme cryptographique RSA, nous avons donc affaire à un chiffrement **asymétrique**.

Chiffrement asymétrique ? un petit rappel

Précédemment, nous avions déjà vu ce qu'était un chiffrement symétrique dans le chapitre AES.

Pour rappel, un chiffrement symétrique nécessite une seule et unique clef pour pouvoir chiffrer ou déchiffrer. C'est comme pouvoir ouvrir et fermer une porte avec une clef.

A contrario, le chiffrement **A**symétrique nécessite absolument deux clefs :

- Une clef pour le chiffrement - appelée aussi clef publique et pouvant être diffusée à tous sans restriction.
- Une autre clef pour le déchiffrement - appelée aussi clef privée et devant être conservée uniquement par son propriétaire et n'être jamais diffusée.

Il n'est pas possible de déchiffrer à l'aide de la clef publique, et il n'est pas possible de chiffrer avec la clef privée.

(A contrario, il est possible de signer un document avec la clef privée et de vérifier la signature avec la clef publique, notion que nous verrons dans la partie Signature. Pour l'instant, cela reste en dehors de notre scope sur le chiffrement de CipherValue)

Les autres paramètres qui sont intégrés lors du chiffrement **RSA** sont :

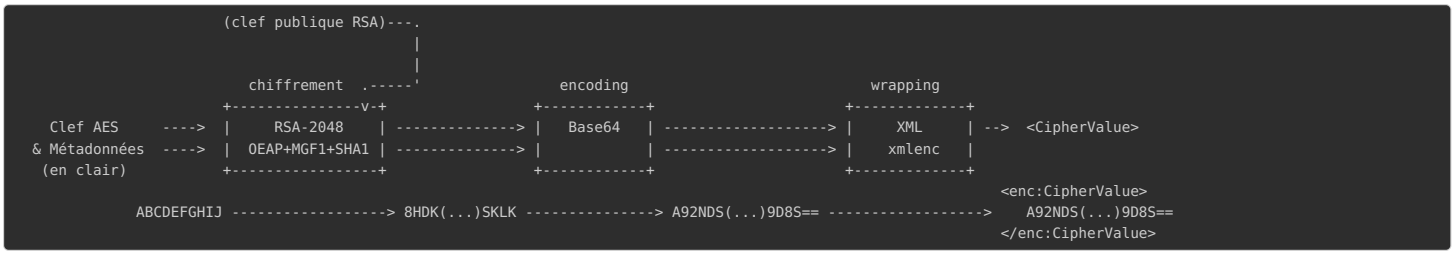
- **OAEP** : Optimal Asymmetric Encryption Padding
- **MGF1** : Mask Generation Function
- **SHA1** : Secure Hash Algorithm

Ces trois algorithmes seront utilisés à l'intérieur même du chiffrement RSA. (à contrario d'un base64 qui se fait à part, soit avant ou soit à la suite).

Dans notre cas, le chiffrement est effectué dans ce sens :

- **Chiffrement** des clefs AES (et des métadonnées) - à l'aide de notre **clef publique** - avec l'algorithme RSA - accompagné des algorithmes Optimal Asymmetric Encryption Padding (OEAP) + Mask Generation Function (MGF1) [1] et de SHA1.
- Encodage de la sortie binaire RSA en Base64.
- La sortie encodée en Base64 est notre **CipherValue**

Un exemple des différentes étapes :



Et le déchiffrement est effectué dans ce sens :

- Décodage de **CipherValue** encodé en Base64.
- **Déchiffrement** de la sortie binaire - à l'aide de notre **clef privée** - avec l'algorithme RSA - accompagné des algorithmes Optimal Asymmetric Encryption Padding (OEAP) + Mask Generation Function (MGF1) [2] et SHA1
- La sortie est du binaire contenant notre clef AES et des métadonnées.

Un exemple des différentes étapes :

COMMENT DÉCHIFFRER UNE CIPHERVALUE

Nous allons nous attarder sur le contenu dans **CipherValue** et tenter de le déchiffrer pour savoir ce qu'il cache.

Si nous suivons nos étapes, la première est le decodage **Base64**.

Pour nos exemples ci-dessous, nous utiliserons l'outil OpenSSL qui nous permettra de voir étape par étape le déchiffrement (vous verrez une version Python juste après).

Si nous enlevons cette couche de base64, voici ce que nous voyons :

```
echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUT0hUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \
    | xxd     # sert uniquement pour l'affichage ci-dessous
```

Et voici le résultat avec `xxd` :

```
 offset        données au format hexadécimal          format string
-------------------------------------------------------------------
00000000:  510e 451c 14a7 0913 35e5 dbea daf5 c64d  | Q.E..�..5������M
00000010:  cd43 9c02 1416 ff3a f6a8 748d 126f deb7  | �C....�:��t..o□
00000020:  3994 165a 1129 b00b 1039 e18f 1a45 2ef7  | 9..Z.)�..9�..E.�
00000030:  d82a 0ffc c384 6c1f f1ca 74d7 6da5 7575  | ø*.��.l.��t�m¥uu
00000040:  a78e 0ca3 c1a5 da57 e2cb 634b 80f9 a0ed  | §..£¡ÚW⌐ËcK.€ù
00000050:  43f0 feff 7b88 938a dd7d 6321 09ee ca61  | Cð▀ÿ{...Ê}c!.îÊa
00000060:  9893 1535 b242 98a9 ee12 95a4 1b36 5891  | ...5²B.®©..¤.6X.
00000070:  7f74 d4c2 038b 162c 3860 b954 764c c114  | .t¶¶...,8`¹TvL¡.
00000080:  20dc 1731 66e8 fc5e 817f 5461 a16f db8e  |  Ü.1fè¸^..Ta¡oÛ.
00000090:  916d ef8a c6e2 fcaa 45e4 b45c 7e36 ece4  | æm▀.ÆâüªEä´\~6ìä
000000a0:  56c3 422a 5b56 7bba 4dc6 d1f2 2d0c faf1  | VÃB*[V{ºMÆÑò-.úñ
000000b0:  d2f8 a7fc e16b f433 4a1c 4d2f 9251 33a1  | Òø§ükáô3J.M/.Q3¡
000000c0:  533c 62ff 7700 ab1b 771a 4aa7 6afd 8786  | S<bÿw.«.w.J§jý‡†
000000d0:  d508 8108 4a26 4999 cb1d 7529 4917 1427  | Õ.ü.J&I.Ë.u)I..'
000000e0:  197b 9548 e56c 18a2 aa22 051d ae78 de40  | .{.H•l.¢ª"..®xÞ@
000000f0:  9289 a040 a341 d135 9143 2b8b f31c 2757  | ’‰ @£AÑ5.C+.ó.'W
```

Cette sortie ne vous dira absolument rien, c'est normal, c'est notre contenu cryptographique **RSA binaire brute**.

Nous devons passer à l'étape 2 :

Le déchiffrement **RSA** avec les paramètres **OAEP**, **MGF1** et **SHA1** ainsi que **notre clef privée RSA**.

Pour cela, nous reprenons la base de notre précédent exemple, et nous rajoutons notre étape RSA à l'aide de `openssl pkeyutl -decrypt` et des options adéquates (`-pkeyopt`) :

```
$ echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUT0hUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \
    | openssl pkeyutl \
        -decrypt \
        -inkey private_key.pem \
        -pkeyopt rsa_padding_mode:oaep \
        -pkeyopt rsa_oaep_md:sha1 \
        -pkeyopt rsa_mgf1_md:sha1 \
    | xxd     # sert uniquement pour l'affichage ci-dessous
```

Et voici le résultat avec `xxd` :

```
 offset       données au format hexadécimal          format string
-------------------------------------------------------------------------
00000000: f1dc 1244 6016 9a0e 85bc 3006 42f8 66ab   | ...D`.....0.B.f.
00000010: 09d5 3df0 13c0 7fa4 341f ded0 eb57 cf7a   | ..=.....4....W.z
00000020: 807a 687d 1ce4 61e5 548f 4b91 a70a 60b7   | .zh}..a.T.K...`.
00000030: bc07 7fb5 4d44 454b 205e f3c1 e260 4b13   | ....MDEK ^...`K.
00000040: 90d9 b961 9825 37e5 3230 3136 2d30 372d   | ...a.%7.2016-07-
00000050: 3239 5432 333a 3539 3a30 302b 3032 3a30   | 29T23:59:00+02:0
00000060: 3032 3032 322d 3037 2d30 3154 3030 3a30   | 02022-07-01T00:0
00000070: 303a 3030 2b30 323a 3030 b16f 5fd3 860c   | 0:00+02:00.o_...
00000080: 11ad 24cd eaaf bafc 7b90                  | ..$.....{.
```

Tout de suite, vous remarquez certaines éléments parfaitement lisibles dans la dernière colonne, comme des dates ou ce MDEK.

Comme annoncé auparavant, **CipherValue** ne contient pas uniquement notre clef AES mais également des métadonnées : c'est toute une structure binaire avec des données à taille fixe qu'il faudra découper et analyser pour pouvoir extraire les bonnes données.

Voici la définition de notre **structure interne CipherValue <u>déchiffrée</u>**, les emplacements et tailles des différentes données :

Nom de la valeur	Position	Taille	Format	
Structure ID	0	16 octets	UUID	Identifiant fixe qui sera toujours `f1 dc 12 44 60 16 9a 0e 85 bc 30 06 42 f8 66 ab`
Certificate ThumbPrint	16	20 octets	Hash	L'empreinte du certificat public de notre encodeur (Signer) (à ne pas confondre avec le Public Key Thumbprint)
CPL Id	36	16 octets	UUID	L'identifiant de la CPL qui a été utilisé
Key Type	52	4 octets	String	Le type de la clef (ex. MDEK)
Key Id	56	16 octets	UUID	L'identifiant de la clef AES
Date Not Valid Before	72	25 octets	Datetime	Notre date de validité du KDM (à titre d'information)
Date Not Valid After	97	25 octets	Datetime	Notre date de validité du KDM (à titre d'information)
AES key	122	16 octets	AES	La précieuse clef de déchiffrement pour notre MXF
Total		**138 octets**		

En tout, la structure de données fait 138 octets.

Mais pourquoi 138 octets, pourquoi pas nos 256 octets ?

Etant donné que nous utilisons des clefs RSA de 2048 bits (256 octets), le **CipherText** (notre **CipherValue** sans sa couche base64, donc notre RSA binaire brut)

sera de 256 octets.

```
printf 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUTOhUzxi/3cAqxt3GkqnavHhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \  # on enlève la couche base64 pour avoir le RSA chiffré
    | wc -c   # on calcule le nombre d'octets en sortie
256 # octets
```

Nous avons un entête OAEP de 42 octets, il reste donc 214 octets de libre pour notre structure (chiffrée ou non) [3]

Ici, nous constatons que nous n'utilisons que 138 octets sur les 214 octets restants.
Cela veut dire qu'il est possible de rajouter 76 octets de métadonnées supplémentaires si besoin (ne le faites pas :)

Notez que le RSA-2048 effectue du chiffrement par bloc de 256 octets, mais rien n'empêche d'avoir un CipherText plus gros, multiple de 2048. Mais nous sommes hors normes SMPTE sur le KDM (pour l'instant, peut-être dans un futur ?)

Si nous reprenons notre sortie et mettons en avant les différents éléments avec un peu de couleur :

```
 offset       données au format hexadécimal           format string
------------------------------------------------------------------------
00000000: f1dc 1244 6016 9a0e 85bc 3006 42f8 66ab   | ...D`.....0.B.f.
00000010: 09d5 3df0 13c0 7fa4 341f ded0 eb57 cf7a   | ..=.....4....W.z
00000020: 807a 687d 1ce4 61e5 548f 4b91 a70a 60b7   | .zh}..a.T.K...`.
00000030: bc07 7fb5 4d44 454b 205e f3c1 e260 4b13   | ....MDEK ^...`K.
00000040: 90d9 b961 9825 37e5 3230 3136 2d30 372d   | ...a.%7.2016-07-
00000050: 3239 5432 333a 3539 3a30 302b 3032 3a30   | 29T23:59:00+02:0
00000060: 3032 3032 322d 3037 2d30 3154 3030 3a30   | 02022-07-01T00:0
00000070: 303a 3030 2b30 323a 3030 b16f 5fd3 860c   | 0:00+02:00.o_...
00000080: 11ad 24cd eaaf bafc 7b90                  | ..$.....{.
```

Nous avons donc les données comme-ci :

Nom de la valeur	Position	Taille	Format	Valeur
Structure ID	0	16 octets	UUID	`f1 dc 12 44 60 16 9a 0e 85 bc 30 06 42 f8 66 ab`
Certificate ThumbPrint	16	20 octets	Hash	`09 d5 3d f0 13 c0 7f a4 34 1f de d0 eb 57 cf 7a 80 7a 68 7d`
CPL Id	36	16 octets	UUID	`1c e4 61 e5 54 8f 4b 91 a7 0a 60 b7 bc 07 7f b5`
Key Type	52	4 octets	4 chars	`4d 44 45 4b` (**MDEK**)
Key Id	56	16 octets	UUID	`20 5e f3 c1 e2 60 4b 13 90 d9 b9 61 98 25 37 e5`
Date Not Valid Before	72	25 octets	Datetime	**`2016-07-29T23:59:00+02:00`**
Date Not Valid After	97	25 octets	Datetime	**`2022-07-01T00:00:00+02:00`**
AES key	122	16 octets	AES	`b1 6f 5f d3 86 0c 11 ad 24 cd ea af ba fc 7b 90`
Total		**138 octets**		

Nous venons de déchiffrer notre premier **EncryptedKey** avec succès !

Il nous reste plus qu'à faire exactement la même procédure sur l'ensemble des **EncryptedKey**.

Notez que dans l'ensemble des valeurs précédentes dans nos métadonnées, notre seule donnée véritablement utile est notre clef AES, le reste étant de la métadonnée que nous pouvons ignorer (si on ne souhaite pas vérifier et valider un KDM ou si nous ne respectons pas les demandes de vérifications normées par le SMPTE et le DCI-CTP [4]. Dans le cas contraire, ces données nous seront utiles afin de valider notre KDM)

QUELLE EST LA LIMITE DU NOMBRE DE CLEFS (ENCRYPTEDKEY) DANS UN KDM ?

Selon la norme **SMPTE 429-2 - DCP Operational Constraints**, il ne peut y avoir plus de 256 clefs cryptographiques uniques [5]. Cependant, cette règle est dans les contraintes de la CPL et non du KDM.

Dans la norme **SMPTE 430-1 - KDM** et **SMPTE 430-3 - ETM**, il n'existe aucune règle dans ce sens [6], on peut donc estimer qu'il n'existe aucune limitation. Cependant, il se peut que les players et encodeurs appliquent une limitation de 256 clefs en lien avec la restriction CPL. Mais techniquement parlant, vous pouvez générer un KDM avec une infinité de clefs.

POURQUOI STRUCTURE ID EST FIXE ?

La norme impose une valeur pour la StructureID et doit être `f1dc124460169a0e85bc300642f866ab`

Selon la norme, ce choix s'explique par le fait d'éviter un type d'attaque sur le KDM :

« By including a known unique plaintext value in the structure, the system is protected in the event that an attacker attempts compromise by substituting the excepted encrypted key block by other data encrypted with the same public key. » -- SMPTE 430-1 KDM.

Cela permet aussi de vérifier rapidement si les données sont correctement déchiffrés, il suffit de lire les 16 premiers octets et de voir si c'est notre identifiant.

QU'EST-CE QUE LE CERTIFICATE THUMBPRINT ?

Le **Certificate ThumbPrint** est souvent confondu avec le **Public Key Thumbprint**.

Même si les deux utilisent le certificat public, les deux ne prennent pas les mêmes infos du certificat pour créer l'empreinte (Thumbprint)

Pour en savoir plus, reportez-vous au chapitre Certificats, paragraphe Certificate Thumbprint et Public Key Thumbprint.

LECTURE DE CIPHERVALUE :

Voici l'exemple d'un code très simpliste pour la lecture de l'ensemble des **CipherValues** d'un KDM et de leurs déchiffrement :

```
#!/usr/bin/env python3

from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
import base64
from lxml import etree

PRIVATE_KEY_FILENAME = "private_key.pem"
PRIVATE_KEY_PASSWORD = None
KDM_FILENAME = "kdm.xml"

# open private key (RSA PRIVATE KEY)
with open(PRIVATE_KEY_FILENAME, "rb") as file:
    private_key_content = file.read()

# load private key
private_key = serialization.load_pem_private_key(
    data = private_key_content,
    password = PRIVATE_KEY_PASSWORD
)

# create padding with OAEP and SHA-1
oaep_padding = padding.OAEP(
    mgf = padding.MGF1(hashes.SHA1()),
    algorithm = hashes.SHA1(),
    label = None
)

# open and read kdm file
with open(KDM_FILENAME, "rb") as xml:
    tree = etree.fromstring(
        text = xml.read()
    )

# show all KeyId from KeyIdList
# AuthenticatedPublic > RequiredExtensions > KDMRequiredExtensions > KeyIdList > TypedKeyId[…] > KeyId
keys = tree.xpath("//*[local-name()='TypedKeyId']")

# read each public KeyId and KeyType
# this part is not encrypted
for key in keys:
    keyType = key.xpath("./*[local-name()='KeyType']/text()")[0]
    keyId = key.xpath("./*[local-name()='KeyId']/text()")[0]
    print("KeyId %s - KeyType %s" % (keyId, keyType))

# find all cipher values, each contains lot of data, including AES key
# AuthenticatedPrivate > enc:EncryptedKey[…] > enc:CipherData
cipher_values = tree.xpath("//*[local-name()='CipherValue']")

# read each private cipherValue
for cipher_value in cipher_values:

    # base64 decryption
    encrypted_value = base64.b64decode(cipher_value.text)

    # RSA decryption
    plaintext = private_key.decrypt(
        ciphertext = encrypted_value,
        padding = oaep_padding
    )

    # parse all plaintext datas
    # data is a static structure
    print("* Cipher Base64          : %s" % cipher_value.text)
    print("* Cipher Text            : %s" % encrypted_value.hex())
    print("* Plaintext              : %s" % plaintext.hex())
    print("* Structure ID           : %s" % plaintext[0:0+16].hex())
    print("* Certificate ThumbPrint : %s" % plaintext[16:16+20].hex())
    print("* CPL Id                 : %s" % plaintext[36:36+16].hex())
    print("* Key Type               : %s" % plaintext[52:52+4].decode('utf-8'))
    print("* Key Id                 : %s" % plaintext[56:56+16].hex())
    print("* Date Not Valid Before  : %s" % plaintext[72:72+25].decode('utf-8'))
    print("* Date Not Valid After   : %s" % plaintext[97:97+25].decode('utf-8'))
    print("* AES Key                : %s" % plaintext[122:122+16].hex())
    print("")
```

CRÉER UNE CIPHERVALUE

Pour créer un CipherValue, il suffit de faire exactement l'inverse :)

Quelques codes utiles :

- Un exemple de programme en Python qui permet un chiffrement utilisé dans CipherValue : ciphervalue-write.py (sans sa structure)
- Un exemple de code pour créer une **CipherValue** (sans sa structure également) et sa version ligne de commande avec OpenSSL, dans la partie KDM Codes

SOUS CHAPITRES

- **KDM - Authenticated Public** : notre partie lisible par tous
- **KDM - Authenticated Private** : notre partie lisible uniquement par le récepteur

- **KDM - Signature** : notre partie pour authentifier et valider le KDM
- **KDM - Codes** : Codes sources et techniques sur les différentes parties du KDM

CHAPITRES CONNEXES

- **La cryptographie** : Pour tout savoir sur la cryptographie utilisée dans le cinéma numérique.
- **Certificats** : Les certificats utilisés dans le cinéma numérique, leurs vies, leurs oeuvres.
- **Cryptographie AES** : La cryptographie symétrique utilisée pour chiffrer les MXF.
- **Cryptographie RSA** : La cryptographie asymétrique utilisée dans les KDM pour chiffrer les clefs AES - entre autres.

NOTES

1. Référence(s) à propos du MGF1: RFC 2437 - PKCS #1: RSA Cryptography Specifications. ↩
2. Référence(s) à propos du MGF1: RFC 2437 - PKCS #1: RSA Cryptography Specifications. ↩
3. Nous retrouverons cette référence dans la norme SMPTE sur le KDM : *« The D-Cinema system uses 2048-bit RSA keys, so the ciphertext is 256-bytes long. Due to the 42-byte header that is part of the OAEP padding [see PKCS1], the plaintext can be at most 214-bytes long. »* -- SMPTE 430-1 - KDM ↩
4. *« 3.4.16. KDM CipherData Structure ID : Verify that the value of the CipherData Structure ID in the RSA protected structure is f1dc124460169a0e85bc300642f866ab »* -- DCI Compliant Test Plan v1.4.2. ↩
5. *« No more than 256 distinct cryptographic keys, as uniquely identified by their Key ID, shall be used to encrypt the assets referenced by a Composition Playlist. »* -- SMPTE 429-2-2013 - DCP Operational Constraints + Specs DCI. ↩
6. *« This segment contains zero or more EncryptedKey fields »* -- SMPTE 430-3 - ETM ↩