LA CRYPTOGRAPHIE APPLIQUÉE DANS UN MXF

La cryptographie des MXF est définie dans la norme **SMPTE 429-6 - Essence Encryption**.

Cette partie est applicable pour les MXF suivants (non-exhaustifs) :

- **JPEG2000** (pistes images classiques)
- **WAV PCM** (pistes sonores classiques)
- **Subtitles** (pistes des sous-titres)
- **Dolby Atmos** (pistes sonores Immersive Audio)

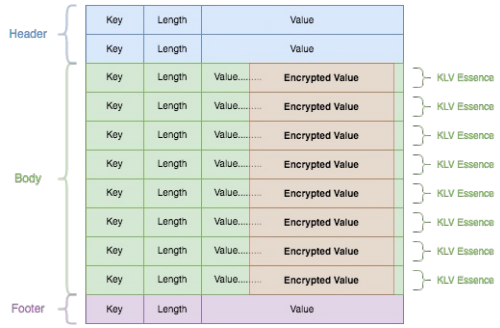
APERÇU DE L'INTÉRIEUR DES MXF

Tout d'abord, il faut comprendre un principe : Quand on dit qu'un MXF est chiffré, c'est un mensonge. Un MXF n'est pas entièrement chiffré.

Seuls certains KLV auront du chiffrement. Et dans ces derniers, seul **un segment** de la partie **Value** sera effectivement chiffré. Tout le reste sera en clair, parfaitement lisible et sans aucun chiffrement.

Par exemple, les KLV dans **Header** et **Footer** sont non-chiffrés, et dans les KLV du **Body**, seules les KLV **Essences** auront un segment chiffré dans leur partie **Value** :

Version simplifiée d'un MXF avec KLV chiffrées



Seul les segments en rouge seront chiffrés. Tout le reste de notre MXF reste en clair, donc non-chiffré. Ainsi, on peut lire des métadonnées nécessaires pour le déchiffrement des segments chiffrés.

LES ÉLÉMENTS NÉCESSAIRES

Entre un MXF normal et un MXF dit-chiffré, il n'existe que 2 KLV supplémentaires et un nouveau type de KLV :

- **2 KLV** en suppléments dans la **partition Header**
- Des KLV **Encrypted Essence Container** dans la **partition Body**

Voici la liste des KLV supplémentaires obligatoires :

Emplacement	Nom du KLV	Universal Label	Nombre d'occurence
Header	Cryptographic Framework	`060e2b34.02530101.0d010401.02010000`	1
	Cryptographic Context	`060e2b34.02530101.0d010401.02020000`	1
Body	Encrypted Essence Container	`060e2b34.02040101.0d010301.027e0100 (SMPTE)` `060e2b34.02040107.0d010301.027e0100 (Interop)`	∞

LES KLV HEADER : CRYPTOGRAPHIC FRAMEWORK & CRYPTOGRAPHIC CONTEXT

Présents dans **Header**, les deux KLV **Cryptographic Framework** et **Cryptographic Context** définissent tout un contexte cryptographique et notamment le type de cryptographie utilisée dans les KLV chiffrés **Encrypted Essence Container**.

Cryptographic Framework et **Cryptographic Context** sont simplement des containeurs d'informations qu'on peut ignorer sans problème pour l'instant car sans impact direct avec le processus de déchiffrement de nos données stockées dans **Encrypted Essence Container**.

En effet, sauf en cas de changement de méthode cryptographique dans les MXF DCP, si vous avez déjà la clef **AES**, vous pouvez déchiffrer **Encrypted Essence** directement en appliquant la méthode de déchiffrement que nous verrons ci-dessous.

Nous reviendrons plus en détails sur **Cryptographic Framework** & **Cryptographic Context** en peu plus tard.

LES KLV BODY : ENCRYPTED ESSENCE CONTAINER

Présents obligatoirement dans **Body**, il faut au moins un **Encrypted Essence Container** par MXF. Et il n'y a pas de limite théorique aux nombres de containeurs possibles.

Voyons d'abord nos KLV chiffrés, les **Encrypted Essence Container**.

LA CRYPTOGRAPHIE UTILISÉE

Actuellement, la cryptographie utilisée pour le chiffrement des données est **AES-128-CBC** :

- **AES** est un algorithme à **chiffrement symétrique** : une clef unique sert pour le chiffrement et le déchiffrement.
- Une **clef** de chiffrement de **128 bits** (16 octets).
- Un **chiffrement par bloc** de bits, chaque bloc sera de **128 bits** (16 octets)
- Un **mode d'opération** des blocs **Cipher Block Chaining (CBC)** : un bloc est chiffré par le résultat du précédent bloc avant d'être chiffré de nouveau avec la clef de chiffrement initiale.

AES est l'algorithme de chiffrement principal, c'est lui qui utilisera la clef de chiffrement initiale pour chiffrer le texte. C'est la partie la plus importante de notre cryptographie. **Cipher Block Chaining (CBC) est un mécanisme en supplément** qui va renforcer ce chiffrement.

Le sujet étant relativement long et (presque) complexe, si vous souhaitez une explication de comment marche cette cryptographie, vous pouvez lire notre paragraphe spécifique à la cryptographie AES-CBC.

Notez que si vous savez déjà comment marche cet algorithme, ce paragraphe est parfaitement dispensable. Tout ce que vous avez à retenir pour l'instant est que pour chiffrer et déchiffrer un contenu **AES-128-CBC**, il vous faudra :

- **Un contenu** en clair (pour le chiffrement) ou chiffré (pour le déchiffrement)
- **Une clef** de 16 octets (128 bits)
- **Un Initialization Vector** (IV) de 16 octets (128 bits)

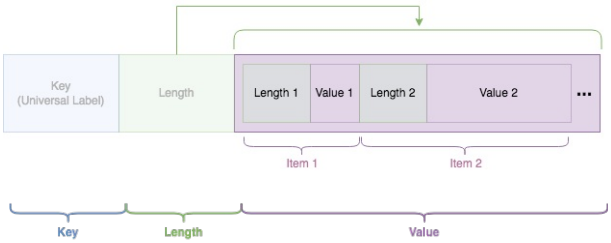
A partir de cela, nous pouvons entrer dans le vif du sujet.

A L'INTÉRIEUR DES KLV CHIFFRÉS

Pour résumé ce qu'on a vu dans les précédents chapitres :

- Les KLV chiffrés sont des **Encrypted Essence Container**.
- Leurs types sont Variable-Length Pack : Suite d'**items** avec un **Length** et une **Value**, l'un à la suite de l'autre.
- L'**Universal Label** pour **Encrypted Essence Container** est `060e2b34.02040101.0d010301.027e0100` [1]

KLV de type « Variable-Length Pack »

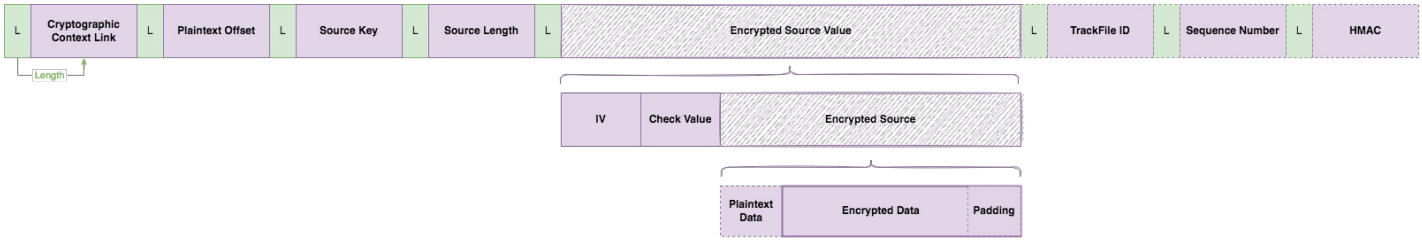


Comme nous l'avons vu dans notre chapitre sur les types de KLV, chaque item sera une métadonnée spécifique qui nous servira dans le traitement de nos données, par exemple, trouver l'emplacement du segment chiffré, initialiser le contexte cryptographique, etc.

La cryptographie des KLV est relativement simple car elle utilise de la cryptographie ouverte et reconnue (AES-CBC) mais possèdent certaines subtilités (deux pour être précis) à prendre en compte pour pouvoir déchiffrer un MXF correctement.

Pour comprendre ces subtilités, nous allons faire un focus sur la partie **Value** d'un KLV chiffré :

Value d'un KLV chiffré



Les éléments en pointillés sont optionnels et peuvent ne pas exister (comme **Plaintext Data** ou **Padding**)

Pour rappel, un **item** possède son propre **Length** (en vert) et sa propre **Value** (en violet).

La taille de ces **Length** est variable car soumis à notre fameux format BER - que nous avons déjà vu dans la section **Length** du KLV. Actuellement, et malgré son format, elles sont toujours de 4 octets et débutent par `0x83` indiquant un BER variable avec 3 octets pour encoder la taille (ex. `0x83000000`)

Voici un descriptif rapide de chaque item : (n'oubliez pas que chaque item démarre par son propre Length)

- **Cryptographic Context Link** (UUID, 16 octets) : est l'identifiant faisant le lien entre notre KLV Essence et le KLV **Cryptographic Context** dans la section **Context ID**. Cet identifiant sera commun à chaque KLV **Encrypted Essence**, voyez cela comme un identifiant de groupe (GID).
- **Plaintext Offset** (Uint64, 8 octets) : est le nombre d'octets où le chiffrement va commencer dans notre partie "**Encrypted Source**". C'est notre première subtilité : Il peut arriver qu'une partie de notre partie "**Encrypted Source**" ne soit pas chiffrée. Si c'est le cas, alors sa valeur sera supérieure à 0.

  Pour donner un rapide exemple, si **Plaintext Offset** est à 128, cela indiquera que les 128 premiers octets de **Encrypted Source** ne seront pas chiffrés.

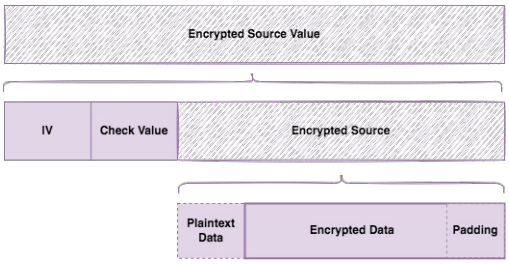
  Pour l'instant, la plupart des MXF sortant des laboratoires cinématographiques n'utilisent pas ce principe et chiffrent entièrement le contenu.

  Nous reviendrons sur cette partie plus en détail un peu plus tard.

- **Source Key** (8 octets) : est un simple identifiant de type "Universal Label" qui détermine le type de contenu. Les valeurs possibles sont (liste non-exhaustive) :

Type	Universal Label
Picture Essence	`060e2b34.01020101.0d010301.15010801`
Sound Essence	`060e2b34.01020101.0d010301.16010101`
Timed Text Essence	`060e2b34.0101010c.0d010509.01000000`
Timed Text Essence	`060e2b34.01020101.0d010301.17010b01`
Immersive Audio (Dolby Atmos)	`060e2b34.01020105.0e090601.00000001`

- **Source Length** (Uint64, 8 octets) : est la taille d'origine de notre contenu (avant chiffrage). Ce chiffre sera toujours égal ou inférieur à la taille des données chiffrées. Cela est dû au fait que le chiffrement s'effectue que sur des blocs de 16 bits, donc la taille de la partie chiffrée sera toujours un multiple de 16. Si la source n'est pas un multiple de 16, il sera complété par des données dit de rembourrage (padding)
- **Encrypted Source Value** (taille variable et définie dans son **Length**) :



  C'est notre partie où se trouve notre chiffrement.

  Le nom **Encrypted Source Value** peut induire en erreur car il semble indiquer que **Encrypted Source Value** est notre contenu chiffré. Or, cette partie est un containeur incluant des éléments nécessaires au déchiffrement de notre contenu chiffré et stocké dans **Encrypted Data**.

  Notez - et c'est important - que ces éléments sont l'un à la suite de l'autre sans aucun **Length** entre eux. Vous ne pouvez déterminer la taille de chaque que parce que certains sont fixes (**IV** et **Check Value** font 16 octets chacun) et que d'autres ont leurs tailles définies autre part dans **Value** :

  - La taille de **Plaintext Data** est déterminée par l'item **Plaintext Offset**
  - La taille d'**Encrypted Data** ne s'évalue qu'en récupérant le **Length** de **Encrypted Source Value** et en retranchant la taille de **IV**, **Check Value** et **Plaintext Data**)

  Voici un descriptif des différents éléments dans **Encrypted Source Value** :

  - **Initialization Vector (IV)** (16 octets) : est notre initialisateur du moteur cryptographique.
  - **Check Value** (16 octets) : est une valeur fixe définie par la norme qui permet de savoir si le déchiffrement se passe bien, indépendamment du type de contenu. C'est notre seconde subtilité. Nous reviendrons également sur cette partie plus en détail un peu plus tard.
  - **Plaintext Data** (taille variable définie dans **Plaintext Offset**) : est une portion - qui peut ne pas être présent - non-chiffrée provenant de notre source. Ainsi, on peut avoir tout ou partie de notre source en clair. Nous verrons ceci plus en détail dans la partie Etude de la value d'un KLV chiffré avec Plaintext Offset
  - **Encrypted Data** (taille variable) : est -enfin- notre partie chiffrée ! (le padding en fait partie)

  Comme vous le voyez, le nom n'indique pas forcément tout ce qui se cache à l'intérieur de cette partie.

Les éléments suivants sont optionnels, ils seront présents seulement si **Message Integrity Code (MIC)** est défini :

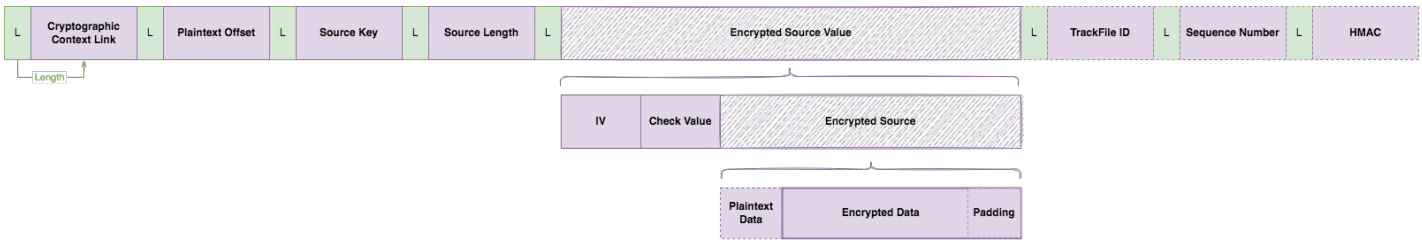
- **TrackFile ID** (UUID, 16 octets) : est un identifiant unique qui identifie le *TrackFile*. Chaque frame dans ce MXF aura le même identifiant. Cet identifiant est aussi l'identifiant **AssetUUID du MXF** qu'on retrouvera dans :
  - **Source Clip → Package UID**,
  - **Source Package → Package UID**
  - **Essence Container Data → Linked Package UID**
- **Sequence Number** (Uint64, 8 octets) : est un numéro qui s'incrémente à chaque nouvelle frame stockée dans ce MXF.
- **Message Integrity Code (MIC)** (20 octets) : est la somme de contrôle (checksum). L'algorithme utilisé est défini dans le KLV **Cryptographic Context**, section **MIC Algorithm**. Actuellement, l'algorithme utilisé est HMAC-SHA1-128.

Pour des raisons de simplification, ces derniers éléments optionnels seront écartés dans les paragraphes "**Etude de la Value d'un KLV chiffré**". Ils seront expliqués en détail dans le paragraphe "**Message Integrity Code**".

ETUDE DE LA VALUE D'UN KLV CHIFFRÉ

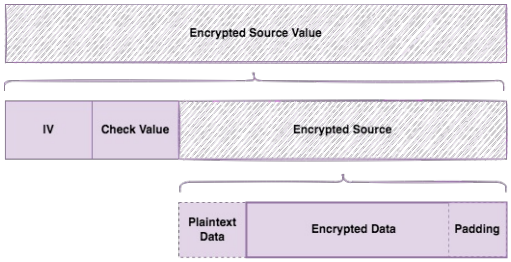
Pour notre étude, nous ne travaillerons que sur la partie **Value** d'un KLV **Encrypted Essence** :

Value d'un KLV Encrypted



Et notamment sur cette partie :

Encrypted Source Value (complet)



Qui - résumé et dans sa plus simple apparence - nous donne ceci :

Encrypted Source Value (simplifié)

Si vous avez lu notre paragraphe sur la cryptographie AES-CBC, cela devrait vous sembler plus familier.

Cependant, vous remarquez qu'il existe un élément en plus par rapport à un chiffrement AES-CBC classique : nous avons un **Check Value** entre notre **Initialization Vector (IV)** et notre **Encrypted Data**, et nous allons voir ce que c'est.

LE BLOC INTERMÉDIAIRE DE VÉRIFICATION "CHECK VALUE"

C'est notre première subtilité !

Check Value est un bloc de 16 octets (128 bits) inséré entre l'**Initialization Vector (IV)** et notre **Encrypted Source** (qui intègre **Plaintext Data** et **Encrypted Data**)

Check Value est une valeur **fixée par avance** et placée comme "en-tête" des données chiffrées.

La valeur en clair est (en hexadécimal) :

```
43 48 55 4B 43 48 55 4B 43 48 55 4B 43 48 55 4B
```

En version lisible :

```
CHUKCHUKCHUKCHUK
```

Elle sert principalement à vérifier rapidement si le déchiffrement s'applique bien, même si nous ne connaissons pas la nature ni le type des données chiffrées.

Par exemple, si nous avions un KLV d'une image d'un format propriétaire dont nous ne connaissons rien, nous saurions que nous avons pu déchiffrer la partie chiffrée sans avoir besoin de manipuler l'image en question.

Cela à plusieurs avantages :

- Pouvoir vérifier que les 16 premiers octets du segment chiffré et déterminer si le chiffrement est correctement appliqué (pas besoin de lire les autres octets ni même de manipuler l'essence)
- Pouvoir vérifier rapidement chaque KLV chiffré et déterminer si le chiffrement est correctement appliqué sur l'ensemble du MXF : Si vous avez 10.000 frames, vous avez donc un IV et un CheckValue dans le calcul : 32 octets * 10.000 = 32 Ko au lieu de Mo (ou même de Go) de données à vérifier.

Quand nous créeons un KLV **Encrypted Essence**, le **Check Value** fait partie intégrante du contenu chiffré, il suivra aussi le canal cryptographique avec la partie **Encrypted Data**.

Pis, sans **Check Value** dans le canal cryptographique, **Encrypted Data** ne pourra **jamais être déchiffré**. Si vous vous souvenez, chaque bloc (de 16 octets) est dépendant de son prédécesseur. Ainsi **Check Value** se comporte comme s'il était le tout premier bloc.

Et cela fait toute la différence dans notre processus de déchiffrement que nous allons voir maintenant.

ETUDE D'UN KLV CHIFFRÉ NORMAL

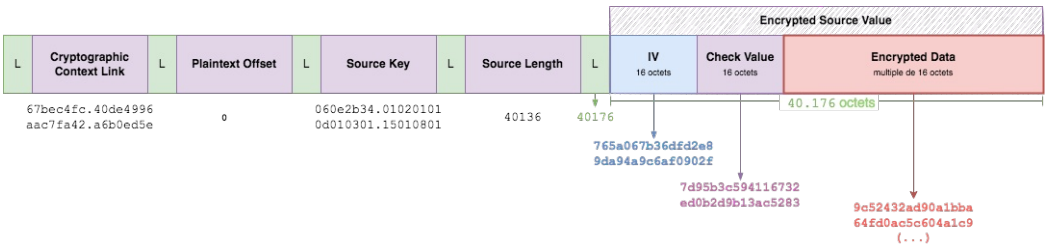
LECTURE DE LA STRUCTURE DE VALUE

Nous allons d'abord étudier la **Value** d'un KLV chiffré normal :

- La source a été **totalement chiffrée**
- Il n'existe pas de partie **Plaintext Data**.

Voici un exemple de **Value** d'un KLV **Encrypted Essence** avec de véritables données :

Value d'un KLV chiffré



Avec un éditeur hexadécimal, voici les premiers d'octets de la Value d'un KLV **Encrypted Essence** :

Vue hexadécimal de la Value d'un KLV chiffré

```
# xxd -c 8 -g 1 -l 128 EncryptedEssenceContainer.value.bin
offset            valeurs                 infos
00000000    83 00 00 10 67 be c4 fc
00000008    40 de 49 96 aa c7 fa 42       <-- Cryptographic Context Link
00000010    a6 b0 ed 5e 83 00 00 08           67bec4fc40de4996aac7fa42a6b0ed5e
00000018    00 00 00 00 00 00 00 00       <-- Plaintext Offset
00000020    83 00 00 10 06 0e 2b 34           0
00000028    01 02 01 01 0d 01 03 01       <-- Source Key
00000030    15 01 08 01 83 00 00 08           060e2b34010201010d01030115010801
00000038    00 00 00 00 00 00 9c c8       <-- Source Length
00000040    83 00 9c f0 76 5a 06 7b           0000000000009cc8
00000048    36 df d2 e8 9d a9 4a 9c       <-- Initialization Vector (IV)
00000050    6a f0 90 2f 7d 95 b3 c5           765a067b36dfd2e89da94a9c6af0902f
00000058    94 11 67 32 ed 0b 2d 9b       <-- Check Value
00000060    13 ac 52 83 9c 52 43 2a           7d95b3c594116732ed0b2d9b13ac5283
00000068    d9 0a 1b ba 64 fd 0a c5       <-- Encrypted Data
00000070    c6 04 a1 c9 0f b4 37 fb           9c52432ad90a1bba64fd0ac5c604a1c9...
00000078    52 1a 00 4f a6 0a 91 a9
```

Chaque item est positionné l'un après l'autre avec sa taille (en vert) et sa valeur.

Voici un code Python très simpliste pour parser la **Value** d'un **KLV Encrypted Data** :

```
import sys
with open(sys.argv[1], "rb") as file:
    print("CryptographicContextLink Length         : %s" % file.read(4).hex())
    print("CryptographicContextLink Value          : %s" % file.read(16).hex())
    print("PlaintextOffset Length                  : %s" % file.read(4).hex())
    print("PlaintextOffset Value                   : %s" % file.read(8).hex())
    print("SourceKey Length                        : %s" % file.read(4).hex())
    print("SourceKey Value                         : %s" % file.read(16).hex())
    print("SourceLength Length                     : %s" % file.read(4).hex())
    print("SourceLength Value                      : %s" % file.read(8).hex())
    print("Encrypted Source Length                 : %s" % file.read(4).hex())
    print("Encrypted Source Value - IV             : %s" % file.read(16).hex())
    print("Encrypted Source Value - CheckValue     : %s" % file.read(16).hex())
    print("Encrypted Source Value - Encrypted Data : %s" % file.read(16).hex())
```

Ce code se veut ultra-simpliste pour la compréhension. Dans le meilleur des mondes, il faudrait lire chaque Length, les convertir puis lire les Values avec leur bonne taille. Mais vu que la norme indique des tailles fixes, autant en profiter pour l'instant :)

Il faudrait également convertir **Encrypted Source Length** pour l'utiliser dans la lecture complète (ou par segment) de notre **Encrypted Source Value**. Ici, nous ne lirons que les 16 premiers octets.

Et enfin, on ne gère pas le **Plaintext Offset**, qu'on verra au paragraphe suivant :)

Voici le résultat du parsing sur la **Value** d'un KLV Encrypted Data :

```
# mxf-encrypted-parse.py KLVEncryptedEssenceContainer.value.bin
CryptographicContextLink Length         : 83000010
CryptographicContextLink Value          : 67bec4fc40de4996aac7fa42a6b0ed5e
PlaintextOffset Length                  : 83000008
PlaintextOffset Value                   : 0000000000000000
SourceKey Length                        : 83000010
SourceKey Value                         : 060e2b34010201010d01030115010801
SourceLength Length                     : 83000008
SourceLength Value                      : 0000000000009cc8  <====================  40136
Encrypted Source Length                 : 83009cf0          <====================  40176
Encrypted Source Value - IV             : 765a067b36dfd2e89da94a9c6af0902f  <=====  16
Encrypted Source Value - CheckValue     : 7d95b3c594116732ed0b2d9b13ac5283  <=====  16
Encrypted Source Value - Encrypted Data : 9c52432ad90a1bba64fd0ac5c604a1c9  <=====  40144
```

Comme vous constatez, chaque **Length** est au format BER - facilement identifiable grâce à leur `0x83`. On voit que les **Values** intégrant des UUID (donc de 16 octets) auront `0x10` (16 en décimal) et que les tailles ne nécessitent que des **Values** de 8 octets, donc `0x08` (8 en décimal).

Une exception avec **Encrypted Source Length** - facilement compréhensible - qui possède une grande valeur `0x009cf0` (40176 en décimal). Notre **Encrypted Source Value** (IV, CheckValue et Encrypted Data) sera donc de 40.176 octets. Étant donné que IV et CheckValue sont de 16 octets chacun, nous savons donc que **Encrypted Data** sera de 40.144 octets (40176 - 32)

Si on décode **SourceLength Value**, nous constatons que la taille de la source était de 40.136 octets. Notre **Encrypted Data** est de 40.144 octets (40144 - 40136), nous savons donc que son padding est de 8 octets. C'est parfaitement normal, la taille d'origine - 40.136 octets - n'est pas un multipe de 16, et notre chiffrement AES-128-CBC nécessite que des blocs de 16 octets. Il faut donc bourrer le dernier bloc avec des données inutiles qu'on écartera après déchiffrement.

À partir de là, nous avons tous les éléments nécessaires pour un déchiffrement :

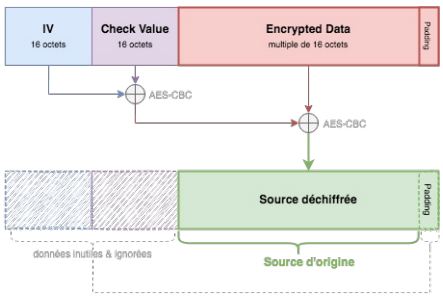
- Notre **Initialization Vector** : `765a067b36dfd2e89da94a9c6af0902f`
- Notre **Check Value** : `7d95b3c594116732ed0b2d9b13ac5283`
- Notre **Encrypted Data** : `9c52432ad90a1bba64fd0ac5c604a1c9` (16 octets seulement)

Avec une clef AES, nous pouvons lancer un processus de déchiffrement.

DÉCHIFFREMENT DE NOS DONNÉES STOCKÉES DANS ENCRYPTED DATA

En se focalisant sur la partie **Encrypted Source Value**, voici le processus de déchiffrement :

Workflow AES-CBC sur un KLV Encrypted



Le processus de déchiffrement est relativement simple : nous utilisons l'IV pour initialiser le déchiffrement et on utilisera comme premier bloc, la **CheckValue**. Puis, nous passons directement à notre contenu chiffré **Encrypted Data**.

Pendant (ou après) le processus de déchiffrement, nous aurons écarté le résultat du déchiffrement de **Check Value** et le padding inutile de notre **Encrypted Data**. La taille de notre **source déchiffrée** sera la même que celle d'origine conservée dans l'item **Source Length** (si ce n'est pas le cas, c'est qu'il y a eu un problème :)

En **pseudo-code**, voici le processus de déchiffrement de notre **Encrypted Source Value** :

```
# On initialise le moteur cryptographique AES-CBC
init_aes_cbc_engine( aeskey, iv )

# On déchiffre le premier bloc "CheckValue"
decrypt_block( checkvalue )

# On déchiffre chaque bloc venant de Decrypted Data
foreach block from decrypted_data :
    plaintext += decrypt_block( block )
```

Vous remarquerez que le résultat du premier appel à `decrypt_block` n'est pas ajouté à notre résultat final (appelé plaintext). C'est normal, le déchiffrement de `checkvalue` n'est pas le contenu de notre source. Si nous récupérons le résultat, nous aurions tout simplement la valeur `CHUKCHUKCHUKCHUK` qui n'appartient bien évidemment pas à notre contenu initial :)

Ce premier appel de `decrypt_block` avec `checkvalue` permet de lancer le processus de déchiffrement juste avant celui de **Decrypted Data**.

Chaque appel de `decrypt_block` sur un bloc de **Decrypted Data** permet d'obtenir une partie de la source déchiffrée. Nous ajoutons donc chaque bloc déchiffré à notre plaintext.

Voici un exemple concret de code Python avec les différents éléments nécessaires d'une **Value** d'un KLV chiffré (IV, CheckValue et les 16 premiers octets de l'**Encrypted Data**) :

```
from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
from cryptography.hazmat.backends import default_backend

aes_key        = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
iv             = b'\x76\x5a\x06\x7b\x36\xdf\xd2\xe8\x9d\xa9\x4a\x9c\x6a\xf0\x90\x2f'
checkvalue     = b'\x7d\x95\xb3\xc5\x94\x11\x67\x32\xed\x0b\x2d\x9b\x13\xac\x52\x83'
encrypted_data = b'\x9c\x52\x43\x2a\xd9\x0a\x1b\xba\x64\xfd\x0a\xc5\xc6\x04\xa1\xc9'

# On définit le moteur cryptographique
cipher = Cipher(
    algorithms.AES(key=aes_key),
    modes.CBC(initialization_vector=iv),
    backend=default_backend()
)
decryptor = cipher.decryptor()

# On rajoute CheckValue dans le processus de déchiffrement
decryptor.update(data=checkvalue)

# On déchiffre notre message chiffré
plaintext = decryptor.update(data=encrypted_data)

# Notre entête JPEG2000 (16 octets)
print(plaintext.hex())

# Résultat: ff4fff51002f00040000100000000870
```

Notre variable `plaintext` a la valeur `ff4fff51002f00040000100000000870`. Les valeurs `0xFF 0x4F 0xFF 0x51` représentent un entête JPEG2000.

Déchiffrement accompli !

Notez que le retour de `decryptor.update(data=checkvalue)` serait `CHUKCHUKCHUKCHUK`.

Bien entendu, notre exemple ne déchiffre que les 16 premiers octets. Pour déchiffrer l'ensemble de Decrypted Data, il suffit d'appliquer la fonction `.update()` sur les blocs suivants.

Notez également qu'on ne parle pas du padding actuellement : il faudra "couper" le padding avant la sortie finale - notamment grâce à **Source Length** qui nous indique la taille d'origine du fichier.

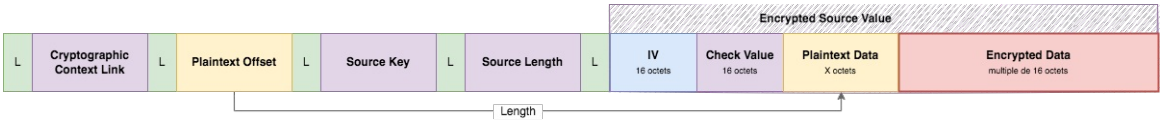
Comme vous le voyez, la différence entre un déchiffrement AES-CBC classique et un déchiffrement AES-CBC KLV est la présence de notre Check Value. Sans la ligne de déchiffrement **Check Value**, notre source déchiffrée serait totalement différente et donc totalement inexploitable.

ETUDE DE LA VALUE D'UN KLV CHIFFRÉ AVEC PLAINTEXT OFFSET

Voici notre seconde subtilité après la **Check Value** : celui du **Plaintext**.

Par convention dans ce paragraphe, l'item indiquant la taille du segment non-chiffré sera nommé **Plaintext Offset** et le segment non-chiffrée dans la partie chiffrée sera nommée **Plaintext Data** (nom non-officiel) :

Value d'un KLV chiffré - avec Plaintext Offset



Alors qu'est-ce que c'est cette histoire de **Plaintext Offset** et **Plaintext Data** ?

Selon la norme SMPTE 429-6, il est possible d'avoir une partie du segment **Encrypted Data** qui ne soit pas du tout chiffrée, appelé **Plaintext Data**. Cette fonctionnalité permet d'avoir accès aux entêtes des essences directement depuis **Encrypted Data** sans avoir à déchiffrer l'ensemble d'**Encrypted Data**.

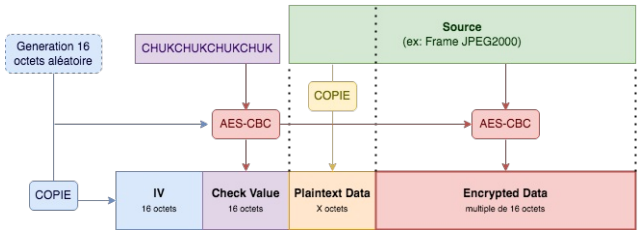
Voici les quelques règles majeures concernant **Plaintext Offset** / **Plaintext Data** :

- Si la valeur de **Plaintext Offset** est supérieure à 0, il détermine la taille de **Plaintext Data** et donc de sa présence dans la partie **Encrypted Source Value**.
- Si **Plaintext Data** est présent, il sera placé entre **Check Value** et **Encrypted Data**.

- Les données en clair dans **Plaintext Data** s'arrêteront là où les données chiffrés dans **Encrypted Data** débuteront. Par exemple, si notre source était `0123456789` et que **Plaintext Data** est `01234`, alors **Encrypted Data** sera le chiffrement de `56789`.

Plaintext Data sont les premiers octets de notre source, non-chiffrées. Et **Encrypted Data**, la suite de notre source, chiffrée. Littéralement, les premiers octets de notre source sont copiés directement dans **Plaintext Data**.

Workflow de chiffrement : de la source à un Encrypted Data



Voici un code Python très simpliste pour parser la **Value** d'un KLV avec du **Plaintext Offset** :

```
import sys
with open(sys.argv[1], "rb") as file:
    print("CryptographicContextLink Length         : %s" % file.read(4).hex())
    print("CryptographicContextLink Value          : %s" % file.read(16).hex())
    print("PlaintextOffset Length                  : %s" % file.read(4).hex())
    # Convertion valeur hexadécimal en décimal (integer)
    plaintextOffset = int.from_bytes(
        file.read(8),
        byteorder='big'
    )
    print("PlaintextOffset Value                   : %s" % plaintextOffset)
    print("SourceKey Length                        : %s" % file.read(4).hex())
    print("SourceKey Value                         : %s" % file.read(16).hex())
    print("SourceLength Length                     : %s" % file.read(4).hex())
    print("SourceLength Value                      : %s" % file.read(8).hex())
    print("Encrypted Source Length                 : %s" % file.read(4).hex())
    print("Encrypted Source Value - IV             : %s" % file.read(16).hex())
    print("Encrypted Source Value - CheckValue     : %s" % file.read(16).hex())
    print("Encrypted Source Value - Plaintext Data : %s" % file.read(plaintextOffset).hex())
    print("Encrypted Source Value - Encrypted Data : %s" % file.read(16).hex())
```

Et son exécution sur la **Value** d'un KLV Encrypted Data avec **Plaintext Offset** :

```
# mxf-encrypted-parse.py KLVEncryptedEssenceContainerWithPlaintextOffset.value.bin
CryptographicContextLink Length         : 83000010
CryptographicContextLink Value          : 1f5d16c78fbe4dd990b567533fd9bd34
PlaintextOffset Length                  : 83000008
PlaintextOffset Value                   : 16
SourceKey Length                        : 83000010
SourceKey Value                         : 060e2b34010201010d01030115010801
SourceLength Length                     : 83000008
SourceLength Value                      : 0000000000009cc8  <====================  40136
Encrypted Source Length                 : 83009cf0          <====================  40176
Encrypted Source Value - IV             : b4d6394b5d1ad1c7bdfcd6d300cad5de   <====  16
Encrypted Source Value - CheckValue     : 3aabe914eaed2d714584cfe5bb8cc762   <====  16
Encrypted Source Value - Plaintext Data : ff4fff51002f00040000100000000870   <====  16
Encrypted Source Value - Encrypted Data : 79de6f3aab54fb6b0f8b228371a40cd8   <====  40144
```

On constate que dans notre **Plaintext Data**, nous avons bien notre entête JPEG2000 `0xff 0x4f 0xff 0x51` en clair et lisible. La taille du **Plaintext Data** est de 36 octets.

Notez que ce code marche aussi sa version sans **Plaintext Offset** : Et oui, vu que le **PlaintextOffset Length** sera à 0, notre **Plaintext Data** sera vide et il passera de suite à **Encrypted Data** :)

Tout comme notre précédent code, nous avons aussi tous les éléments nécessaires pour un déchiffrement dans ce contexte précis :

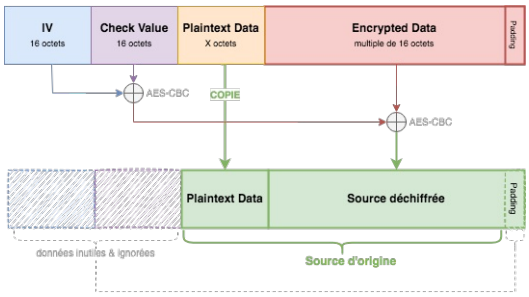
Bloc	Valeur
Notre **Initialization Vector**	`b4d6394b5d1ad1c7bdfcd6d300cad5de`
Notre **Check Value**	`3aabe914eaed2d714584cfe5bb8cc762`
Notre **Plaintext Data**	`ff4fff51002f00040000100000000870`
Notre **Encrypted Data**	`79de6f3aab54fb6b0f8b228371a40cd8` (16 octets seulement)

Avec une clef AES, nous pouvons lancer un processus de déchiffrement.

DÉCHIFFREMENT DE NOS DONNÉES (AVEC NOTRE PLAINTEXT DATA)

En se focalisant sur la partie **Encrypted Source Value**, voici le processus de déchiffrement avec **Plaintext Data** :

Worflow AES-CBC sur un KLV chiffré - avec Plaintext Offset



Voyez la petite subtilité en action : Comme nous l'avons vu précédemment, nous allons appliquer un déchiffrement en utilisant **IV**, **CheckValue** et bien entendu **Encrypted Data**. **Plaintext Data** sera simplement copié en début de notre **source déchiffrée**.

D'un point de vue programmatique, nous avons plusieurs choix :

- Soit vous lancez votre chiffrement en lisant l'IV, puis Check Value et vous vous déplacez directement vers Encrypted Data en enjambant Plaintext Data, pour revenir le chercher après afin de l'insérer en début de votre source déchiffrée : un peu laborieux mais pourquoi pas.
- Soit vous lancez le déchiffrement dès le début, et quand vous passez sur Plaintext Data, il suffit de bypasser le chiffrement : moins laborieux et ne demande pas de courir de droite à gauche dans le fichier.
- Soit vous utilisez votre propre idée :)

En **pseudo-code**, voici le processus de déchiffrement de notre **Encrypted Source Value** avec notre **Plaintext Data** :

```
# On initialise le moteur cryptographique AES-CBC
init_aes_cbc_engine( aeskey, iv )

# On applique le premier bloc "CheckValue"
decrypt_block( checkvalue )

# On copie directement "Plaintext Data" sans traitement
plaintext = plaintext_data

# On déchiffre chaque bloc venant de Decrypted Data
foreach block from decrypted_data :
    plaintext += decrypt_block( block )
```

Vous constatez qu'entre notre précédente version et celle-ci, il n'existe qu'une ligne de différence, celle de copier directement Plaintext Data dans notre source déchiffrée, juste avant de reprendre le déchiffrement des différents blocs.

Voici donc notre précédent exemple de code Python agrémenté de la gestion du Plaintext Offset :

```
from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
from cryptography.hazmat.backends import default_backend

aes_key        = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
iv             = b'\xb4\xd6\x39\x4b\x5d\x1a\xd1\xc7\xbd\xfc\xd6\xd3\x00\xca\xd5\xde'
checkvalue     = b'\x3a\xab\xe9\x14\xea\xed\x2d\x71\x45\x84\xcf\xe5\xbb\x8c\xc7\x62'
plaintext_data = b'\xff\x4f\xff\x51\x00\x2f\x00\x04\x00\x00\x10\x00\x00\x00\x00\x08\x70'
encrypted_data = b'\x79\xde\x6f\x3a\xab\x54\xfb\x6b\x0f\x8b\x22\x83\x71\xa4\x0c\xd8'

# On définit le moteur cryptographique
cipher = Cipher(
    algorithms.AES(key=aes_key),
    modes.CBC(initialization_vector=iv),
    backend=default_backend()
)
decryptor = cipher.decryptor()

# On rajoute CheckValue dans le processus de déchiffrement
decryptor.update(data=checkvalue)

# On copie simplement notre Plaintext Data
plaintext = plaintext_data

# On déchiffre notre message chiffré
plaintext += decryptor.update(data=encrypted_data)

# Notre entête JPEG2000 (16 octets)
print(plaintext.hex())

# Résultat :
# ff4fff51002f000400001000000008700000000000000000000000100000000870

# ff4fff51002f00040000100000000870   <== partie plaintext_data
# 00000000000000000000100000000870   <== partie encrypted_data déchiffrée
```

Le résultat est -bien entendu- un entête JPEG2000 de part la valeur initiale de Plaintext Data, mais le reste est bien sa suite déchiffrée qu'on peut voir dans l'output de notre JPEG2000 disponible ici :

```
# xxd frame.j2c | head -n14
ff4f ff51 002f 0004 0000 1000 0000 0870    .O.Q./.........p    <== Notre Plaintext Data copié
0000 0000 0000 0000 0000 1000 0000 0870    ..............p    <== Notre Encrypted Data déchiffré
0000 0000 0000 0000 0003 0b01 010b 0101    ................    <== Le reste des données venant de notre source
0b01 01ff 5200 1301 0400 0101 0603 0300    ....R...........
0077 8888 8888 8888 ff5c 0029 427f 347e    .w.......\.)B.4~
f17e f17e ae76 f176 f176 ae6f 026f 026e    .~.~.v.v.v.o.o.n
e067 4d67 4d67 6750 0350 0350 4457 d257    .gMgMggP.P.PDW.W
d257 60ff 5f00 1000 0000 0106 0304 0600    .W`._...........
0001 0703 04ff 5500 2200 5000 0000 43e1    ......U."P...C.
0000 0014 a900 0000 0caf 0000 002e cc00    ................
0000 0639 0000 0001 b5ff 6400 2800 0143    ...9......d.(..C
7265 6174 6564 2077 6974 6820 4456 5320    reated with DVS
436c 6970 7374 6572 2035 2e31 302e 302e    Clipster 5.10.0.
3134 00ff 9000 0a00 0000 0043 e100 06ff    14.........C....
```

BEYOND THE VALUE

Maintenant que nous avons fait un tour de ce qu'est KLV **Encrypted Essence**, nous allons aller une étape supplémentaire : un parseur MXF qui ne va lire que les KLV **Encrypted Essence** et en extraire les données déchiffrés.

Toujours pour des raisons de simplicité, on fera l'impasse sur pas mal d'éléments comme des vérifications d'usage, un (bon) calcul BER et autres joyeusetés.

Première étape, on va lire un MXF et détecter chaque KLV sans distinction :

```
# Conversion en int
def to_int(length : bytes = b'') -> int:
    return int.from_bytes(length, byteorder='big')

with open("encrypted-key-00000000000000000000000000000000-plaintextoffset.mxf", "rb") as file:

    while True:

        # Key : Universal Label
        key = file.read(16)

        # End of file
        if not key:
            break

        # Length (BER format)
        length = to_int(file.read(4)[1:])  # BER format - read last 3 bytes

        # Value
        value = file.read(length)

        # Show each KLV
        print("{key} - {length:>6d} - {data}...".format(
            key = key.hex(),
            length = length,
            data = value[0:16].hex()
        ))
```

Notre output sera :

```
060e2b34020501010d01020101020400 -    120 - 0001000200000001000000000000000...
060e2b34020501010d01020101050100 -   1574 - 0000005700000012020106 0e2b340101...
060e2b34025301010d01010101012f00 -    206 - 3c0a00102113fbd416404dfa81d5cf33...
060e2b34025301010d01010101013000 -    214 - 3c0a0010dfd8ce6a95b148c19ba32dff...
060e2b34025301010d01010101011800 -     92 - 3c0a00103f36ccaa62af4a12a1d88c5d...
060e2b34025301010d01010101012300 -     72 - 3c0a0010a4cfdd4389e1473196a7bf13...
060e2b34025301010d01010101013600 -    160 - 3c0a001047b923528b0e4e428a229eae...
060e2b34025301010d01010101013b00 -    112 - 3c0a0010dc9c6229026c43c9a8671e4f...
060e2b34025301010d01010101010f00 -     80 - 3c0a0010fe22b0f2aad54d4882527c1c...
060e2b34025301010d01010101011400 -     75 - 3c0a001066a514ed413b4f509972ff80...
060e2b34025301010d01010101013b00 -    110 - 3c0a00106af5f622912147b793934c2f...
060e2b34025301010d01010101010f00 -     80 - 3c0a0010cf9f60b3703a42f2b72db873...
060e2b34025301010d01010101011100 -    108 - 3c0a0010159e18f6b52542d3948fe1c3...
060e2b34025301010d01010101013700 -    294 - 3c0a0010cccd0ddda60b4ba8a16fdcfd...
060e2b34025301010d01010101013b00 -    112 - 3c0a0010b7011601be7448d0a7095922...
060e2b34025301010d01010101010f00 -     80 - 3c0a001062016d762c1e4029b37e2be5...
060e2b34025301010d01010101011400 -     75 - 3c0a00106bd4ed968bef41f695adefee...
060e2b34025301010d01010101013b00 -    110 - 3c0a0010c065e009fb2646c59a12af65...
060e2b34025301010d01010101010f00 -     80 - 3c0a0010123653c947f84490808f8a88...
060e2b34025301010d01010101011100 -    108 - 3c0a0010c09f8dd674b845fa913e67ef...
060e2b34025301010d01010101013a00 -     94 - 3c0a001048ded21b46b14b4a8f742af6...
060e2b34025301010d01010101010f00 -     68 - 3c0a00109c5aec7f2cb94378a18dda84...
060e2b34025301010d01010101014100 -    106 - 3c0a00104b90cd87f20744dba2eff1c4...
060e2b34025301010d01040102010000 -     40 - 3c0a001059b2cb66c4864536a9aaf40c...
060e2b34025301010d01040102020000 -    120 - 3c0a00100e6593f6d4694e60a78bf138...
060e2b34025301010d01010101012900 -    189 - 3c0a001009a466f780894b19a8b141d4...
060e2b34025301010d01010101015a00 -    181 - 3c0a0010f10b60d72fb440e4b53f4020...
060e2b34010101020301021001000000 -  11164 - 0000000000000000000000000000000...
060e2b34020501010d01020101030400 -    120 - 0001000200000001000000000004000...
060e2b34020401010d01030102 7e0100 -  40300 - 830000101f5d16c78fbe4dd990b56753...  <-- :)
060e2b34020501010d01020101040400 -    120 - 0001000200000001000000000000de0c...
060e2b34025301010d01020101100100 -    131 - 3c0a0010d7d70d8385564599af3ce7ec...
060e2b34020501010d01020101110100 -     40 - 0000000000000000000000000000001...
```

On remarque que nos données sont correctement structurées : nous avons nos **Universal Label**, puis la taille de chaque **Value** et les débuts de chaque **Value**.

Rapidement, on distingue un KLV avec une taille plus imposante que les autres : c'est notre KLV contenant notre essence. Pour être plus précis, l'**Universal Label** correspond à celui d'un KLV **Encrypted Essence Container** : `060e2b34020401010d010301027e0100` .

Nous allons maintenant filtrer nos KLV pour ne conserver que nos **Encrypted Essence Container** :

```
        # Value
        value = file.read(length)
+-------------------------------------------------------------------+
|       # KLV SMPTE & Interop                                       |
|       if key.hex() != "060e2b34020401010d010301027e0100" and \    |
|          key.hex() != "060e2b34020401070d010301027e0100":         |
|          continue                                                 |
+-------------------------------------------------------------------+
        # Show each KLV
        print("{key} - {length:>6d} - {data}...".format(
```

Il existe d'autres méthodes pour filtrer, comme comparaison bytes à bytes, ou bien trouver la bonne catégorie, la bonne version, etc. Mais utilisons plutôt une méthode rapide, simple et lisible pour l'instant avec une simple conversion de bytes en string à l'aide de `.hex()` et notre **Universal Label**.

Notre output nous donne maintenant :

```
060e2b34020401010d010301027e0100 -  40300 - 830000101f5d16c78fbe4dd990b56753...
```

Maintenant, nous allons lire la **Value** de notre KLV.

Pour rappel, la structure **Variable-Length Value** est une **suite d'items** ne contenant que des **item.Length** et **item.Value**.

- **item.Length** est de 4 octets
- **item.Value** est officiellement toujours variable, mais dans notre cas, nous aurons principalement des tailles de 8 octets, 16 octets et le reste véritablement variables.
  - **La taille 8 octets** est généralement utilisée pour stocker une valeur décimale, comme ... une taille :) par exemple la taille du **Plaintext Offset**, la taille d'origine de la source. Avec 8 octets, vous pouvez avoir une valeur décimale jusqu'à (2^{64} -1) soit 18.446.744.073.709.551.615, ce qui est normalement assez large :)
  - **La taille 16 octets** est généralement utilisée pour stocker un UUID, comme **Cryptographic Context Link** ou **Source Key**.
  - **La taille variable** est utilisée pour notre **Encrypted Source Value** qui va stocker notre essence.

```
+------------------------------------------------------------------------+
|  import io
+------------------------------------------------------------------------+
        (...)
        print("{key} - {length:>6d} - {data}...".format(
            key = key.hex(),
            length = length,
            data = value[0:16].hex()
        ))
+------------------------------------------------------------------------+
|       # read Value
|       value = io.BytesIO(value)
|
|       print("CryptographicContextLink Length         : %s" % value.read(4).hex())
|       print("CryptographicContextLink Value          : %s" % value.read(16).hex())
|       print("PlaintextOffset Length                  : %s" % value.read(4).hex())
|
|       plaintextOffsetValue = to_int(value.read(8))
|       print("PlaintextOffset Value                   : %s bytes" % plaintextOffsetValue)
|
|       print("SourceKey Length                        : %s" % value.read(4).hex())
|       print("SourceKey Value                         : %s" % value.read(16).hex())
|       print("SourceLength Length                     : %s" % value.read(4).hex())
|
|       sourceLengthValue = to_int(value.read(8))
|       print("SourceLength Value                      : %s bytes" % sourceLengthValue)
|
|       encryptedSourceLength = to_int(value.read(4)[1:])  # BER format - read last 3 bytes
|       print("Encrypted Source Length                 : %s bytes" % encryptedSourceLength)
|
|       IV = value.read(16)
|       print("Encrypted Source Value - IV             : %s" % IV.hex())
|
|       checkValue = value.read(16)
|       print("Encrypted Source Value - CheckValue     : %s" % checkValue.hex())
|
|       plaintextData = value.read(plaintextOffsetValue)
|       print("Encrypted Source Value - Plaintext Data : %s" % plaintextData.hex())
|
|       # EncryptedData excludes plaintextData + IV + CheckValue
|       encryptedDataLength = ( encryptedSourceLength - plaintextOffsetValue - 16 - 16 )
|       encryptedData = value.read(encryptedDataLength)
|
|       print("Encrypted Source Value - Encrypted Data : %s..." % encryptedData[0:16].hex())
+------------------------------------------------------------------------+
```

J'utilise `io.BytesIO` car cela permet d'utiliser des fonctions I/O - comme `read()`, `write()`, `seek()`, `tell()`, mais on pourrait très bien faire `data[0:4]` ou même memoryview.

Notre output nous donne maintenant :

```
060e2b34020401010d010301027e0100 - 40300 - 830000101f5d16c78fbe4dd990b56753...
CryptographicContextLink Length         : 83000010
CryptographicContextLink Value          : 1f5d16c78fbe4dd990b567533fd9bd34
PlaintextOffset Length                  : 83000008
PlaintextOffset Value                   : 16 bytes
SourceKey Length                        : 83000010
SourceKey Value                         : 060e2b34010201010d01030115010801
SourceLength Length                     : 83000008
SourceLength Value                      : 40136 bytes
Encrypted Source Length                 : 40176 bytes
Encrypted Source Value - IV             : b4d6394b5d1ad1c7bdfcd6d300cad5de
Encrypted Source Value - CheckValue     : 3aabe914eaed2d714584cfe5bb8cc762
Encrypted Source Value - Plaintext Data : ff4fff51002f00040000100000000870
Encrypted Source Value - Encrypted Data : 79de6f3aab54fb6b0f8b228371a40cd8...
```

Nous constatons que, dans **Plaintext Data**, nous voyons nos fameux `0xff4ffff5` de nos headers JPEG2000, ce qui indique que notre parsing se déroule correctement pour l'instant.

Maintenant que nous avons nos principaux éléments pour un déchiffrement, poursuivons dans le vif du sujet, nous allons rajouter notre coeur cryptographique à l'intérieur de notre parseur.

```
  import io
+------------------------------------------------------------------------+
| from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
| from cryptography.hazmat.backends import default_backend
+------------------------------------------------------------------------+
        (...)
        print("Encrypted Source Value - Encrypted Data : %s..." % encryptedData[0:16].hex())
+------------------------------------------------------------------------+
|       # Set cryptographic engine
|       cipher = Cipher(
|           algorithms.AES(key=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'),
|           modes.CBC(initialization_vector=IV),
|           backend=default_backend()
|       )
|       decryptor = cipher.decryptor()
|
|       # add CheckValue on decryption workflow
|       decryptor.update(data=checkValue)
|
|       # add PlaintextData directly to Plaintext
|       plaintext = plaintextData
|
|       # add chunk of encryptedData to Plaintext
|       encryptedData = io.BytesIO(encryptedData)
|       while True:
|           chunk = encryptedData.read(16)
|           if not chunk:
|               break
|           plaintext += decryptor.update(data=chunk)
|
|       print("Plaintext Source Value : %s" % len(plaintext))
|       print("Padding: %d" % (len(plaintext) - sourceLengthValue))
|
|       # write Plaintext to file
|       with open("output_%d.j2c" % file.tell(), "wb") as f:
|           f.write(plaintext)
+------------------------------------------------------------------------+
```

Nous initialisons notre moteur cryptographique avec notre IV, et notre clef AES (0x00).

On rajoute notre **Plaintext Data** directement dans notre sortie (plaintext).

Parce que io.BytesIO apporte son lot de fonctions utiles, nous l'utilisons également ici, mais vous pouvez tout autant lire directement encryptedData par portion avec un offset pour décaler à chaque fois le segment en cours de lecture.

Nous lisons des portions (chunk) de **Encrypt Data** d'une taille de 16 octets que nous passons à notre fonction `.update()`.

Si nous lançons maintenant notre programme, nous aurons notre frame JPEG2000 :

```
39K    output_56844.j2c
```

Les numéros n'ont pas d'intérêt, ils sont simplement là pour différencier les différentes sorties et ne représentent que le pointeur de position dans le fichier au moment de l'écriture.

Vous remarquerez que nous n'avons pas traité notre **padding** : il se trouve encore dans notre output.

Regardons la fin de notre output d'une de nos frames JPEG2000 :

```
00009ca0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cb0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cc0: 8080 8080 8080 ffd9 0001 0203 0405 0607  ................
```

Comparons avec la même frame d'origine :

```
00009ca0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cb0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cc0: 8080 8080 8080 ffd9                      ........
```

Nous constatons 8 octets supplémentaires :

```
00009ca0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cb0: 8080 8080 8080 8080 8080 8080 8080 8080  ................
00009cc0: 8080 8080 8080 ffd9 0001 0203 0405 0607  ................
```

Par ailleurs, nous savons qu'un JPEG2000 se termine par `0xffd9` .

Ces 8 octets sont nos octets de padding pour compléter notre dernier bloc de 16 octets.

Le moyen de supprimer ce padding est de prendre sa taille d'origine stockée dans **Source Length Value** et de l'utiliser pour couper avant la finalisation ou l'écriture dans le fichier :

```
+-------------------------------------------------------------------------+
|       # write Plaintext to file
|       with open("output_%d.j2c" % file.tell(), "wb") as f:
|           f.write(plaintext[0:sourceLengthValue])
+-------------------------------------------------------------------------+
```

Et voila, votre output sera parfaitement déchiffré et sauvegardé, il est comme l'original :

```
# shasum -a 256   output_56844.j2c   frame.j2c
b469a8333a8ad708becdfc7544f180c1198b12722a2051b90c66b5ba58ded825    output_56844.j2c
b469a8333a8ad708becdfc7544f180c1198b12722a2051b90c66b5ba58ded825    frame.j2c
```

Les checksums sont identiques, preuve que le processus de déchiffrement est complet.

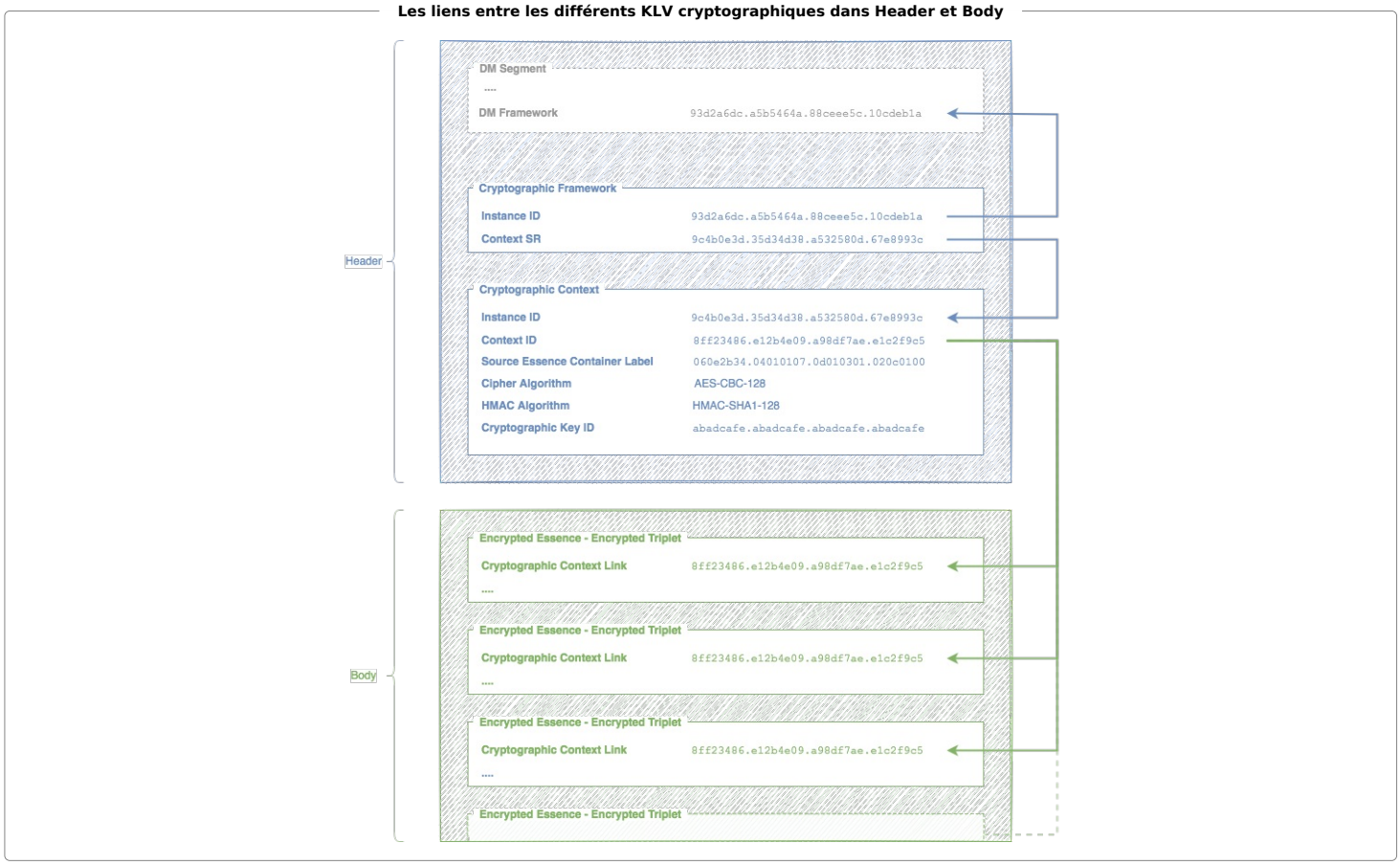
Nous pourrions optimiser ce code, par exemple, placer l'écriture au moment du déchiffrement, cela permet d'éviter de construire un énorme buffer et d'utiliser un petit de 16 octets - au risque d'avoir plus appels systèmes pour les entrées/sorties. Et bien d'autres optimisations encore, amusez-vous en utilisant son code source :)

LES KLV HEADERS EN DÉTAIL

Comme nous avons vu précédemment, nous avons deux KLV spécifiques dans les Headers : **Cryptographic Framework** & **Cryptographic Context**

Avant de partir en détail sur nos deux KLV supplémentaires, voici un schéma expliquant les liaisons entre eux :

Les liens entre les différents KLV cryptographiques dans Header et Body



KLV HEADER : CRYPTOGRAPHIC FRAMEWORK

Universal Label	`060e2b34.02530101.0d010401.02010000`
KLV Type	Local Sets (Baby KLV)
Référence	SMPTE 429-6-2006 - MXF Track File Essence Encryption, Cryptographic Framework Set

```
Contenu du KLV brut :

   Key    : 060e2b34.02530101.0d010401.02010000
   Length : 83xxxxxx
   Value  : 3C0A 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFF 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Contenu du KLV interprété (exemple) :
   ┌──────────────────────────────────────────────────┐
   │  Instance ID   :  6dfa3d83.8a8045fd.adbe8a65.dbd2d1a5
   │  Context SR    :  cbc0f87d.d1a147ca.824bb4d0.6e9dd565
   └──────────────────────────────────────────────────┘
```

Cryptographic Framework fait lien entre le KLV commun **Descriptive Metadata Segment** et notre autre KLV cryptographique **Cryptographic Context**

Nom de l'item	Type	Taille	Local Tag	Universal Label associé	Infos
Instance ID	UUID	16 octets	`3C0A` statique	`060e2b34.01010101.01011502.00000000`	Lien vers **DM Framework -> DM Segment**
Context SR	UUID	16 octets	`FFFF` dynamique	`060e2b34.01010109.06010104.020d0000`	Lien vers **Cryptographic Context -> Instance ID**
(**GenerationUID**)	UUID	16 octets	`0102` statique	`060e2b34.0101010a.05200701.08000000`	Optionnel : un identifiant de création

GenerationUID est une valeur optionnelle, je ne l'ai que rarement constaté sur un MXF.

KLV HEADER : CRYPTOGRAPHIC CONTEXT

Universal Label	`060e2b34.02530101.0d010401.02020000`
KLV Type	Local Sets (Baby KLV)
Référence	SMPTE 429-6-2006 - MXF Track File Essence Encryption, Cryptographic Context Set

```
Contenu du KLV brut :

   Key    : 060e2b34.02530101.0d010401.02020000
   Length : 83xxxxxx
   Value  : 3C0A 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFE 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFD 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFC 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFB 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            FFFA 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Contenu du KLV interprété (exemple) :
   ┌──────────────────────────────────────────────────────────────────────────────────────────────────────────┐
   │  Instance ID                      :  cbc0f87d.d1a147ca.824bb4d0.6e9dd565
   │  Context ID                       :  67bec4fc.40de4996.aac7fa42.a6b0ed5e
   │  Source Essence Container Label   :  JPEG2000 Picture Element - Frame Wrapped   (060e2b34.04010107.0d010301.020c0100)
   │  Cipher Algorithm                 :  AES-CBC-128                               (060e2b34.04010107.02090201.01000000)
   │  Message Integrity Code Algorithm :  HMAC-SHA1-128                             (060e2b34.04010107.02090202.01000000)
   │  Cryptographic Key ID             :  deadbeef.deadbeef.deadbeef.deadbeef
   └──────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Nom de l'item	Type	Taille	Local Tag	Universal Label associé	Infos
Instance ID	UUID	16 octets	`3C0A` statique	`060e2b34.01010101.01011502.00000000`	Lien vers **Cryptographic Framework -> Context SR**
Context ID	UUID	16 octets	`FFFE` dynamique	`060e2b34.01010109.01011511.00000000`	Lien vers **Encrypted Essence -> Cryptographic Context Link**
Source Essence Container Label	UUID	16 octets	`FFFD` dynamique	`060e2b34.01010109.06010102.02000000`	Identifiant du type de source
Cipher Algorithm	UUID	16 octets	`FFFC` dynamique	`060e2b34.01010109.02090301.01000000`	Identifiant du type de cryptographie
MIC Algorithm	UUID	16 octets	`FFFB` dynamique	`060e2b34.01010109.02090302.01000000`	Identifiant du type de cryptographie
Cryptographic Key ID	UUID	16 octets	`FFFA` dynamique	`060e2b34.01010109.02090301.02000000`	Identifiant pour la clef
(**GenerationUID**)	UUID	16 octets	`0102` statique	`060e2b34.0101010a.05200701.08000000`	Optionnel : un identifiant de création

GenerationUID est une valeur optionnelle, je ne l'ai que rarement constaté sur un MXF.

Cryptographic Context est le KLV donnant toutes les informations essentielles sur le contexte cryptographique du chiffrement utilisée sur les **Encrypted Essence Container** et la somme de contrôle (checksum) calculée pour le **Message Integrity Code (MIC)**.

Source Essence Container Label est l'identifiant permettant de définir quel type d'essence est stocké. Ici, nous aurons l'UUID `060e2b34.04010107.0d010301.020c0100` indiquant que c'est un **JPEG2000 Picture Element - Frame Wrapped**

Cipher Algorithm est l'identifiant qui va définir l'algorithme utilisé pour le chiffrement des essences. Ici, nous aurons l'UUID `060e2b34.04010107.02090201.01000000` qui correspond à AES-CBC-128 (page 8). Vous avez un descriptif de l'algorithme AES-CBC dans un chapitre spécifique.

Message Integrity Code Algorithm est l'identifiant qui va définir l'algorithme utilisé pour la somme de contrôle (checksum) des essences. Ici, nous aurons l'UUID `060e2b34.04010107.02090202.01000000` qui correspond à HMAC-SHA1-128 (page 8).

Cryptographic Key ID est un identifiant important car il sera utilisée pour faire lien entre différentes parties de métadonnées. **Cryptographic Key ID n'est pas la clef AES**, c'est un simple identifiant défini par avance qui servira de "point commun" à plusieurs éléments dans l'univers magique du cinéma numérique. Cet identifiant doit être généré à chaque génération d'une nouvelle clef AES (ou alors définir manuellement cet identifiant si la clef AES existe déjà qui a servi à chiffrer le MXF en question).

La **Cryptographic Key ID** est utilisée dans la CompositionPlaylist et son KDM :

Par exemple, avec la CPL de notre DCP `DCP-INSIDE-CRYPTE_TST-2D-24_C_FR-XX_51_4K_20220102_SMPTE_OV`

```
$ grep "KeyId" "CPL.xml"
<KeyId>urn:uuid:cf2ab7c6-c00f-4d52-aae2-3c3396a89b93</KeyId>
<KeyId>urn:uuid:36205699-4079-4140-a93a-6bd716750348</KeyId>
```

Si nous regardons dans nos deux MXF avec `mxf-analyzer` :

```
$ mxf-analyzer -f "jp2k_video.mxf" -v | grep "Cryptographic Key ID"
Cryptographic Key ID : cf2ab7c6.c00f4d52.aae23c33.96a89b93

$ mxf-analyzer -f "wav_audio.mxf" -v | grep "Cryptographic Key ID"
Cryptographic Key ID : 36205699.40794140.a93a6bd7.16750348
```

ou avec asdcplib :

```
$ asdcp-info -i "jp2k_video.mxf" | grep "CryptographicKeyID"
CryptographicKeyID: cf2ab7c6-c00f-4d52-aae2-3c3396a89b93
```

```
$ asdcp-info -i "wav_audio.mxf" | grep "CryptographicKeyID"
CryptographicKeyID: 36205699-4079-4140-a93a-6bd716750348
```

La **Cryptographic Key ID** sera aussi présent dans chaque KDM créé pour cette CPL :

```
$ grep "<KeyId>" "KDM.xml"
<KeyId>urn:uuid:cf2ab7c6-c00f-4d52-aae2-3c3396a89b93</KeyId>
<KeyId>urn:uuid:36205699-4079-4140-a93a-6bd716750348</KeyId>
```

Notez que nous retrouverons **KeyId** également dans les **CipherValue** dans notre KDM. Mais cela est en dehors du scope de ce paragraphe, reportez-vous à la page KDM pour plus d'informations.

Les **KeyId sont les Cryptographic Key ID** de chaque MXF.

Un exemple d'utilisation avec le Doremi DMS-2000

Le DMS-2000 est encodeur et décodeur de DCP et générateur de KDM : Lorsque qu'il génère un KDM, il n'a pas besoin de manipuler les MXF d'un DCP.

Il lui suffit simplement de lire sa CPL et de trouver tous les tags **KeyId** dans le XML. Puis il va chercher dans son répertoire `/usr/share/AESkeys/` s'il possède un fichier nommé avec la **KeyId** (par exemple `/usr/share/AESkeys/deadbeefdeadbeefdeadbeefdeadbeef`).

Si c'est le cas, il lui suffit d'utiliser le contenu du fichier, d'en extraire la clef AES stockée dedans et de faire son travail... :-)

En résumé: Cryptographic Key ID (MXF) == KeyId (CPL) == KeyId (KDM)

MESSAGE INTEGRITY CODE (MIC)

Le **Message Integrity Code (MIC)** est une somme de contrôle (checksum) permettant de déterminer si nos données sont correctement là et n'ont pas été corrompues ou altérées : elle permet de vérifier l'intégrité des données.

Conceptuellement, un checksum est le résultat d'un calcul effectué sur un ensemble d'octet.

Il existe plusieurs types de calcul, plusieurs algorithmes, des plus simples aux plus complexes, des plus fiables aux plus vulnérables. Elles sont appelées **fonctions de hashage**. Par exemple, CRC32, MD5, SHA, BLAKE, ...

Pour le cas d'un MXF DCP, le **Message Integrity Code (MIC)** doit être généré via l'algorithme **HMAC-SHA1-128** :

HMAC

HMAC est l'acronyme de **Keyed-Hash Message Authentication Code** *(code d'authentification de message de hachage à clé)* est un mécanisme qui va utiliser une fonction de hashage (par exemple, SHA) et appliquer une clef secrète afin d'obtenir un hash particulier de 20 octets.

SHA-1

SHA-1 est l'acronyme de **Secure Hash Algorithm**, c'est notre fonction de hashage qui va effectuer des calculs sur les différents octets de nos données. Le résultat sera représenté par un nombre hexadécimal de 20 octets (ou 160 bits)

HMAC va utiliser la fonction de hashage **SHA-1** afin de générer un hash sur nos données avec un "enrobage" supplémentaire en utilisant la clef secrète pour chiffrer.

Comme se déroule le procédé HMAC ? (en résumé)

Initialement, notre clef va être dupliquée : nous aurons deux clefs auxquelles nous allons appliquer des transformations - notamment avec du XOR et les valeurs `0x36` (pour la 1ère clef) et `0x5c` (pour la 2nd clef) - afin d'obtenir deux clefs uniques (nommées **I_KEY_PAD** et **O_KEY_PAD**) de 64 octets chacune.

Ces deux clefs uniques serviront chacune aux deux passes cryptographiques SHA-1 :

- La première clef **I_KEY_PAD** sera concaténée avec les données **DATA** pour une première passe cryptographique SHA-1 afin de produire notre premier hash **HASH_1** :

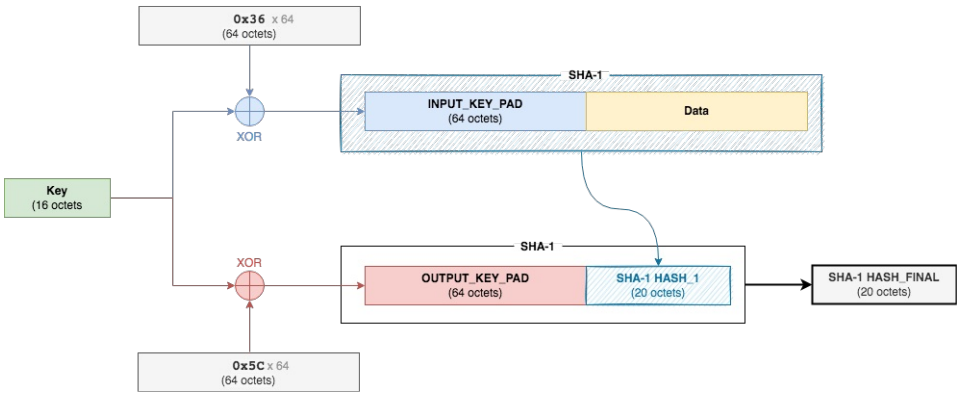
```
SHA1( I_KEY_PAD + DATA   ) => HASH_1 (20 octets)
```

- La seconde clef **O_KEY_PAD** sera concaténée avec le hash obtenu **HASH_1** pour une seconde passe cryptographique SHA-1 afin de produire notre second hash **HASH_2** :

```
SHA1( O_KEY_PAD + HASH_1 ) => HASH_2 (20 octets)
```

Le dernier hash **HASH_2** sera notre **Message Integrity Code (MIC)**

En une image, voila ce qu'il se passe :



Et une implémentation prototype rapide en Python et en C

La combinaison de **HMAC** + **SHA-1** permet d'avoir un checksum qui peut être utilisé à la fois pour du contrôle d'intégrité des données et également pour valider son authenticité : sans sa clef secrète, le checksum sera différent.

Des algorithmes ouverts, publics et largement utilisés. Des implémentations disponibles dans tous les langages de programmation et des données disponibles prêt au hashage, tout semble être dans le meilleur des mondes. Sauf ... qu'il existe (encore) une subtilité propre à SMPTE.

Comme nous le voyons depuis tout à l'heure - que ce soit au niveau de la norme SMPTE ou au niveau des outils manipulant des MXF - il est indiqué que l'algorithme utilisé pour calculer le **Message Integrity Code (MIC)** est **HMAC-SHA1-128**. Rien de plus.

Sauf si on s'attarde sur le paragraphe de l'item MIC (SMPTE 429-6 - paragraphe 7.10), il existe une petite subtilité posée comme cela et - si nous n'y prêtons pas forcément attention - elle peut faire toute la différence :

> *« The key used in the MIC algorithm (MICKey) is derived from the key (CipherKey) referred to by Cryptographic Key ID using the combination of algorithms defined in Appendix 3.1 and Appendix 3.3 of FIPS 186-2. Specifically the MICKey shall equal to x1 per Appendix 3.1 using CipherKey as the seed-key XKEY, setting XSEEDj = 0 and constructing the function G(t,c) per Appendix 3.3. In addition, since Appendix 3.1 is being used as a general random number generator, the term "mod q" in step 3.c shall be omitted, per the "General Purpose Random Number Generation" of the Change Notice 1 addendum. x0 shall be discarded. »*
>
> **SMPTE 429-6 - Paragraphe 7.10 - MIC (fair-use)**

Sous tout ce charabia se cache une étape essentielle : On doit appliquer une fonction de dérivation de clef (**Key Derivation**) sur notre clef initiale dont la définition se trouve dans la norme **FIPS 186-2**, Section **« General Purpose Random Number Generation »** (Appendix 3.1).

Sans cette étape, notre calcul HMAC-SHA1-128 ne donnera jamais le bon résultat. Je n'ai aucune idée précise des raisons pour cette étape supplémentaire. Probablement pour empêcher l'utilisation d'une méthode d'attaque permettant de retrouver la clef d'origine qui a servi lors du calcul HMAC.

QU'EST-CE QU'UNE DÉRIVATION DE CLEF (KEY DERIVATION) ?

En quelques mots : on prend une clef, on la triture tellement qu'elle donne une nouvelle clef.

Si vous appliquez votre clef initiale à votre HMAC-SHA1-128 sans passer par la case "dérivation de clef", vous n'obtiendrez qu'un mauvais hash. Il faut donc en amont créer cette clef dérivée puis l'appliquer à notre HMAC-SHA1-128.

Au final et en résumé, le véritable algorithme utilisé est **HMAC-SHA1-128 + FIPS-186-2-GPRNG**. Et là, vous obtiendrez un **checksum compatible SMPTE**.

Faire un schéma explicatif + implementation en Python

ET LA CLEF ?

Pour notre HMAC-SHA1-128, nous avons besoin d'une clef de 128 bits (16 octets).

Et quelle clef avons-nous à disposition et qui serait de 128 bits ? Notre clef AES, bien entendu !

A l'aide de ce programme permettant de générer une dérivation **FIPS-186-2-GPRNG** sur notre clef AES `00000000000000000000000000000000`, elle deviendra alors `55ACAD4D81EF20B346F80F4A2BF74A28` : c'est cette dernière que nous devons utiliser avec notre algorithme HMAC.

Nous reviendrons sur le calcul de cette dernière plus tard.

LES DONNÉES UTILISÉES POUR LE CHECKSUM

Il existe une dernière petite subtilité : le checksum n'est **pas** calculé **sur l'ensemble des données** mais seulement sur une portion.

La somme de contrôle utilisera les éléments suivants pour son calcul :

	Nom	Taille (octets)	Format	Position
Encrypted Source Value	**Initialization Vector (IV)**	16		Offset 68
	Check Value	16	`CHUKCHUKCHUKCHUK`	
	(Plaintext Data)	Variable		
	Encrypted Data	Variable		
	TrackFile ID - Length	4	BER long-format coding - 0x83	
	TrackFile ID - Value	16	UUID	
	Sequence Number - Length	4	BER long-format coding - 0x83	
	Sequence Number - Value	8	Integer	
	MIC - Length	4	BER long-format coding - 0x83	

Il n'y a pas besoin de déchiffrer les données dans **Encrypted Data** ni **Check Value**, il faut lire les données brutes - sans aucun traitement.

Si on veut la faire rapide (et sans respecter les tailles en cas de changement), ce seront les données entre l'octet 68 et l'octet -20 de la fin (on ne lit pas les 20 octets du MIC, bien entendu). (Astuce Python: `value[68:-20]`).

CALCUL D'UN HASH MIC

Pour nos besoins, la **Derivation Key** de notre clef AES sera déjà fixée dans le code.

```
+--------------------------------------------------------------------------------
| import hashlib
| import hmac
+--------------------------------------------------------------------------------

    (...)
    encryptedData = value.read(encryptedDataLength)
    print("Encrypted Source Value - Encrypted Data : %s...%s" % (encryptedData[0:16].hex(), encryptedData[-16:].hex()))

+--------------------------------------------------------------------------------
|   # TrackFile ID
|   trackfile_length = value.read(4)
|   print("TrackFile ID Length                     : %s" % trackfile_length.hex())
|   trackfile_value = value.read(16)
|   print("TrackFile ID Value                      : %s" % trackfile_value.hex())
|
|   # Sequence Number
|   sequencenum_length = value.read(4)
|   print("Sequence Number Length                  : %s" % sequencenum_length.hex())
|   sequencenum_value = value.read(8)
|   print("Sequence Number Value                   : %s" % sequencenum_value.hex())
|
|   # MIC
|   mic_length = value.read(4)
|   print("Message Integrity Code (MIC) Length     : %s" % mic_length.hex())
|   mic_value = value.read(20)
|   print("Message Integrity Code (MIC) Value      : %s" % mic_value.hex())
|
|   # Derivation key :
|   # (FIPS 186-2 - General Purpose Random Number Generation)
|   # La clef AES 00000000000000000000000000000000 devient :
|   derivation_key = b'\x55\xAC\xAD\x4D\x81\xEF\x20\xB3\x46\xF8\x0F\x4A\x2B\xF7\x4A\x28'
|
|   # Calculate HMAC
|   digester = hmac.new(
|       key=derivation_key,
|       msg=None,
|       digestmod=hashlib.sha1
|   )
|   digester.update(msg=IV)
|   digester.update(msg=checkValue)
|   digester.update(msg=plaintextData)
|   digester.update(msg=encryptedData)
|   digester.update(msg=trackfile_length)
|   digester.update(msg=trackfile_value)
|   digester.update(msg=sequencenum_length)
|   digester.update(msg=sequencenum_value)
|   digester.update(msg=mic_length)
|   print("Calculate MIC = %s" % digester.hexdigest())
+--------------------------------------------------------------------------------
```

Si nous lançons notre programme, nous obtenons un MIC calculé :

```
$ ./mxf-encrypted-hmac.py "encrypted-key-00000000000000000000000000000000.mxf"
(...)
TrackFile ID Length                     : 83000010
TrackFile ID Value                      : 89af85f04a1545ec8a769008829b2029
Sequence Number Length                  : 83000008
Sequence Number Value                   : 0000000000000001
Message Integrity Code (MIC) Length     : 83000014
Message Integrity Code (MIC) Value      : 5b594d66d09cf6ddfda8f6e691e4291ea7097bc8
Calculate MIC = 5b594d66d09cf6ddfda8f6e691e4291ea7097bc8
```

Nous voyons de suite que notre calcul est le même que le MIC Value inscrit dans le KLV.

En version réduite, vous pouvez même vous permettre de faire cela en Python :

```
    (...)
    # read Value
    # value = io.BytesIO(value)  # on ne va pas utiliser io.BytesIO sur value
    (...)
+--------------------------------------------------------------------------------
|   derivation_key = b'\x55\xAC\xAD\x4D\x81\xEF\x20\xB3\x46\xF8\x0F\x4A\x2B\xF7\x4A\x28'
|
|   # Calculate HMAC
|   digester = hmac.new(
|       key=derivation_key,
|       msg=None,
|       digestmod=hashlib.sha1
|   )
|
|   digester.update(value[68:-20])
|   print("Calculate MIC = %s" % digester.hexdigest())
+--------------------------------------------------------------------------------
```

ECRIRE NOS PROPRES KLV CRYPTOGRAPHIQUES

Alors que nous avons vu précédemment comment lire des KLV cryptographiques, nous allons voir maintenant comment en écrire !

Pour cela, nous aurons deux approches : soit via une bibliothèque, soit par nous mêmes.

Dans le milieu, il existe deux bibliothèques reconnues :

- MXFLib, créée par l'un des créateurs du format MXF et gérant l'ensemble des spécifications MXF mais qui semble un peu abandonnée depuis des années.
- ASDCPlib, une bibliothèque plus orientée pour la gestion des MXF DCP/IMF et encore maintenue.

La dernière étant la plus utile pour nos besoins, nous nous tournons vers cette dernière.

AVEC ASDCPLIB

ASDCPlib intègre de base des outils en ligne de commande qui permettent de faire par exemple de la création de MXF, du unwrapping, de la vérification, etc...

Mais ici notre but n'est pas d'utiliser les outils tout-fait mais de coder notre propre programme - en C++ - afin de créer des KLV cryptographiques. Pour le coup, avec asdcplib, il serait plus compliqué de faire QUE des KLV au lieu de faire directement un MXF complet et chiffré, alors autant en profiter :)

Ce code est un proof-of-concept, il va générer un MXF qu'avec une seule frame chiffrée :

```
#include <AS_DCP.h>
#include <KM_prng.h>  /* FortunaRNG */
#include <Metadata.h> /* MXF:: */

using namespace ASDCP;

int main(void) {

    WriterInfo Info;
    JP2K::MXFWriter Writer;
    JP2K::FrameBuffer FrameBuffer(1024 * 1024);
    JP2K::PictureDescriptor PDesc;
    JP2K::SequenceParser Parser;
    // AES
    AESEncContext* Context = 0;
    const byte_t aes_key[16] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    Kumu::FortunaRNG RNG;
    byte_t iv[CBC_BLOCK_SIZE];
    // HMAC
    HMACContext* HMAC = 0;

    // Set Header
    Info.LabelSetType = LS_MXF_SMPTE;
    Kumu::GenRandomUUID(Info.ContextID);
    // AssetUUID == TrackFile ID (MIC)
    Kumu::GenRandomUUID(Info.AssetUUID);

    // Set Cryptographic
    Kumu::GenRandomUUID(Info.CryptographicKeyID);
    Info.EncryptedEssence = true;
    Context = new AESEncContext;
    Context->InitKey(aes_key);
    Context->SetIVec(RNG.FillRandom(iv, CBC_BLOCK_SIZE));

    // Set HMAC
    Info.UsesHMAC = true;
    HMAC = new HMACContext;
    HMAC->InitKey(aes_key, Info.LabelSetType);

    // Set Parser from files
    Parser.OpenRead("essences/JPEG2000/");
    Parser.FillPictureDescriptor(PDesc);

    // Go to the first file
    Parser.Reset();

    // Open MXF
    Writer.OpenWrite("dump.mxf", Info, PDesc);

    // --- foreach frame -----------------------------------
    // Read each frame (only one here)
    // Each call of ReadFrame() shift to the next frame
    // ReadFrame() returns a zero if no new frame
    Parser.ReadFrame(FrameBuffer);
    FrameBuffer.PlaintextOffset(0);  // force no plaintext
    // Write each frame into MXF (only one here)
    Writer.WriteFrame(FrameBuffer, Context, HMAC);
    // -----------------------------------------------------

    // Close MXF
    Writer.Finalize();

    // Show 256 bytes from JPEG2000
    FrameBuffer.Dump(stderr, 256);
    // Show all metadatas from JPEG2000
    JP2K::PictureDescriptorDump(PDesc);

    return 0;
}
```

Note: Ce code se veut ultra-simplifié. Par exemple, nous ne vérifions pas les retours des fonctions (à l'aide d'un `Result_t`). Principalement pour avoir une vision rapide des différentes étapes entre les fonctions et éviter du code inutile à la compréhension.

Pour expliquer quelques principes de la librairie asdcplib :

`Parser.OpenRead` va lire un répertoire et indexer chaque image JPEG2000 dans un index interne. A chaque appel de `Parser.ReadFrame`, ce dernier va lire le fichier suivant dans sa liste, puis remplir le `FrameBuffer` qui sera utilisé juste après par `Writer.WriteFrame` pour créer un KLV Essence. Dans notre code, on ne fait que lire la première image. Mais on devrait boucler sur `Parser.ReadFrame` et `Writer.WriteFrame` pour ajouter chaque fichier dans le MXF.

Pour compiler :

```
# Vous devrez d'abord compiler asdcplib
# afin d'avoir les modules libasdcp.(so,dylid) et libkumu.(so,dylid)
ASDCPLIB="/chemin/asdcplib/"

g++ -g -O2 \
    -lpthread \
    -Wl,-bind_at_load \
    -DHAVE_OPENSSL=1 \
    -I$(ASDCPLIB)/src/ \
    $(ASDCPLIB)/src/.libs/libasdcp.so \    # Linux
    $(ASDCPLIB)/src/.libs/libkumu.so \     # Linux
    $(ASDCPLIB)/src/.libs/libasdcp.dylib \ # MacOS
    $(ASDCPLIB)/src/.libs/libkumu.dylib \  # MacOS
    `pkg-config openssl --cflags` \
    `pkg-config openssl --libs` \
    asdcplib-create-encrypted-mxf.cpp \
    -o asdcplib-create-encrypted-mxf
```

Pour démarrer :

```
# Vous devrez d'abord compiler asdcplib
# afin d'avoir les modules libasdcp.(so,dylid) et libkumu.(so,dylid)
ASDCPLIB="/chemin/asdcplib/"

# Linux
LD_LIBRARY_PATH="$(ASDCPLIB)/src/.libs:${LD_LIBRARY_PATH}" ./asdcplib-create-encrypted-mxf

# MacOS
DYLD_LIBRARY_PATH="$(ASDCPLIB)/src/.libs:${DYLD_LIBRARY_PATH}" ./asdcplib-create-encrypted-mxf
```

Nous venons de créer un MXF chiffrer avec la librairie asdcplib. Vous retrouverez le code source ici : asdcplib-create-encrypted-mxf.cpp + Makefile.

Et si nous faisions nos propres KLV cryptographiques nous-mêmes ? :)

ECRIRE NOS PROPRES KLV CRYPTOGRAPHIQUES EN PYTHON

Disclaimer

Parce que créer un MXF entier serait trop complexe pour ce simple paragraphe, nous allons nous cantonner à la création des trois KLV nécessaires : Les deux headers **Cryptographic Framework** & **Cryptographic Context**, et un KLV **Encrypted Essence Container**.

Effectuons un petit rappel sur les paramètres de chaque KLV :

- KLV **Cryptographic Framework** :
  - **Type** : Local Sets
  - **Universal Label** : `060e2b34.02530101.0d010401.02010000`
  - **Référence** : SMPTE 429-6-2006 - MXF Track File Essence Encryption, Cryptographic Framework Set
- KLV **Cryptographic Context** :
  - **Type** : Local Sets
  - **Universal Label** : `060e2b34.02530101.0d010401.02020000`
  - **Référence** : SMPTE 429-6-2006 - MXF Track File Essence Encryption, Cryptographic Context Set
- KLV **Encrypted Essence** :
  - **Type** : Variable-Length Pack
  - **Universal Label** : `060e2b34.02040101.0d010301.027e0100`
  - **Référence** : SMPTE 429-6-2006 - MXF Track File Essence Encryption, Encrypted Essence

Commençons par la création du KLV **Cryptographic Framework**.

1/3 - KLV CRYPTOGRAPHIC FRAMEWORK

Ce KLV est relativement simple, il n'a que deux items :

- **Instance ID** est une référence à un précédent KLV **DM Segment** et son item **DM Framework**.
- **Context SR** est une référence à notre futur KLV **Cryptographic Context** et son item **Instance ID**.

Voici un schéma rapide de comment doit être ce KLV :

```
Key    : 060e2b34.02530101.0d010401.02010000
Length : 83xxxxxx
Value  : 3C0A 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFF 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Comme cela, rien de bien compliqué. Le plus "difficile" étant de calculer le **Length** au format BER.

Nous avons deux approches possibles :

- Soit on la fait simple et on considère que la taille de **Value** est de 40 octets : **Deux items** à 20 octets :
  - 2 octets pour son **Local Tag**,
  - 2 octets pour sa taille de valeur
  - 16 octets pour stocker l'UUID
- Soit nous allons créé notre **Value** en entier puis nous calculerons sa taille (qui nous donnera 40 octets).

Nous allons partir sur la dernière approche car plus les KLV au format **Local Sets** vont devenir complexe, plus il sera difficile de faire comme avec la première approche.

Nous allons commencer par créer les deux items : **Instance ID** et **Context SR**.

Leurs **Locals Tags** sont respectivement `3C0A` (**Instance ID**) et `FFFF` (**Context SR**). Les deux valeurs de ces deux items seront des UUID, donc leurs tailles seront de 16 octets chacunes. En hexadécimal, `16` donne `0x10` . Et sur deux octets: `0x0010` : cela sera la taille de notre item.

La valeur de **Instance ID** est liée à un UUID déjà définie dans un précédent KLV `DM Framework -> DM Segment` - que nous n'aurons pas dans notre exemple. Nous définirons alors notre propre valeur arbitrairement.

La valeur de **Context SR** est liée à l'UUID de notre prochain KLV `Cryptographic Context -> Instance ID` - qui n'existe pas encore. Donc pour celui-ci nous pourrons créer un UUID aléatoire.

On prépare nos UUID :

```
import uuid
instance_id_uuid = b"\x6D\xFA\x3D\x83\x8A\x80\x45\xFD\xAD\xBE\x8A\x65\xDB\xD2\xD1\xA5"
context_sr_uuid  = uuid.uuid4().bytes
```

Nous allons calculer les tailles de nos items sous un format de 2 octets (on convertit le `int` en `bytes` 2 octets) :

```
instance_id_length = len(instance_id_uuid).to_bytes(2, byteorder='big')
context_sr_length  = len(context_sr_uuid).to_bytes(2, byteorder='big')
```

Par exemple, la taille de `instance_id_uuid` est de 16 octets. A l'aide de `to_bytes` , nous demandons explicitement une conversion de ce nombre (un integer) en une valeur en byte (donc `0x10`) et dans une représentation de 2 octets (donc `0x0010`).

La taille de `instance_id_length` et `context_sr_length` seront `0x0010` .

Nous allons créer nos deux items avec nos différents **LocalTags** (`3C0A` et `FFFF`) en concaténant **LocalTags** + **Taille** + **UUID** :

```
item_instance_id = b'\x3C\x0A' + instance_id_length + instance_id_uuid
item_context_sr  = b'\xFF\xFF' + context_sr_length + context_sr_uuid
```

Cela donne par exemple pour `item_instance_id` : `3C0A00106DFA3D838A8045FDADBE8A65DBD2D1A5`

Nous allons créer enfin la **Value** de notre KLV en fusionnant nos deux items **Instance ID** et **Context SR** :

```
value = item_instance_id + item_context_sr
```

Et maintenant, on va calculer le **Length** de notre KLV au format BER long-form-coding en 4 bytes (donc un entête à `0x83`) en récupérant la taille de notre **Value** :

```
length = b'\x83' + len(value).to_bytes(3, byteorder='big')
```

`length` nous donnera `0x83000028` car `len(value)` sera à 40 octets, en hexadécimal, c'est `28` et on demande une valeur en byte de 3 octets, donc `000028` .

Un petit affichage pour la beauté du geste :-)

```
print("Instance ID - Value   = %s" % instance_id_uuid.hex())
print("Instance ID - Length  = %s" % instance_id_length.hex())
print("Instance ID - Item   => %s" % item_instance_id.hex())

print("Context SR - Value    = %s" % context_sr_uuid.hex())
print("Context SR - Length   = %s" % instance_id_length.hex())
print("Context SR - Item    => %s" % item_context_sr.hex())

print("Length                = %s" % length.hex())
print("Value (items)         = %s" % value.hex())
```

Le résultat donne :

```
Instance ID - LocalTag  = 3C0A
Instance ID - Length    = 0010
Instance ID - Value     = 6DFA3D838A8045FDADBE8A65DBD2D1A5
Instance ID - Item     => 3C0A00106DFA3D838A8045FDADBE8A65DBD2D1A5

Context SR - LocalTag   = FFFF
Context SR - Length     = 0010
Context SR - Value      = 060BBAEC103D4ACDAD10B09261607FB7
Context SR - Item      => FFFF0010060BBAEC103D4ACDAD10B09261607FB7

KLV Length              = 83000028
KLV Value (items)       = 3C0A00106DFA3D838A8045FDADBE8A65DBD2D1A5FFFF0010060BBAEC103D4ACDAD10B09261607FB7
```

Nous avons tout ce qu'il nous faut pour créer un KLV complet en fusionnant l'**Universal Label** de **Cryptographic Framework**, son **Length** et sa **Value** :

```
cryptographic_framework_ul = b"\x06\x0E\x2B\x34\x02\x53\x01\x01\x0D\x01\x04\x01\x02\x01\x00\x00"
cryptographic_framework = cryptographic_framework_ul + length + value
print("KLV = %s" % cryptographic_framework.hex())
```

Ce qui donne :

```
KLV = 060E2B34025301010D01040102010000830000283C0A00106DFA3D838A8045FDADBE8A65DBD2D1A5FFFF00108D68A9E6CC764FD49A2FAC7838F24BAE
```

Nous voila avec notre premier KLV ! :) Passons maintenant au KLV du **Cryptographic Context**

2/3 - KLV CRYPTOGRAPHIC CONTEXT

Ce KLV est aussi simple que le précédent, il a juste plus d'items.

- **Instance ID** (`3C0A`) est une référence à notre précédent KLV **Cryptographic Context** et son item **Context SR** que nous avons généré aléatoirement.
- **Context ID** (`FFFE`) est une référence à nos futurs **Encrypted Essence** et à son item **Cryptographic Context Link**, on aura donc à générer un UUID aléatoire qu'on réutilisera plus tard.
- **Source Essence Container Label** (`FFFD`) sera pour notre exemple `060e2b34.04010107.0d010301.020c0100` car mentionnant que c'est un **JPEG2000 Picture Element - Frame Wrapped**
- **Cipher Algorithm** (`FFFC`) et **MIC Algorithm** (`FFFB`) ont aussi leur propres labels :
  - `060e2b34.04010107.02090201.01000000` pour **AES-CBC-128**
  - `060e2b34.04010107.02090202.01000000` pour **HMAC-SHA1-128**
- **Cryptographic Key ID** (`FFFA`) va être généré aléatoirement.

Voici un schéma rapide de comment doit être ce KLV :

```
Key    : 060e2b34.02530101.0d010401.02020000
Length : 83xxxxxx
Value  : 3C0A 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFE 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFD 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFC 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFB 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
         FFFA 0010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Vu le nombre d'item, cette fois, on va créer une fonction :)

```
def create_item(localtag : bytes, item_value : bytes):
    item_length = len(item_value).to_bytes(2, byteorder='big')
    return localtag + item_length + item_value
```

Ici, nous ne faisons que ce que nous avons fait auparavant.

Maintenant, on va créer les deux UUIDs qu'y n'ont aucune référence :

```
context_id_uuid = uuid.uuid4().bytes
```

Et maintenant, on va créer tous les items (notez que `context_sr_uuid` a été défini dans notre précédent KLV) :

```
item_instance_id   = create_item(b'\x3C\x0A', context_sr_uuid)  # variable définie dans notre précédent KLV
item_context_id    = create_item(b'\xFF\xFE', context_id_uuid)
item_essence_label = create_item(b'\xFF\xFD', b'\x06\x0E\x2B\x34\x04\x01\x01\x07\x0D\x01\x03\x01\x02\x0C\x01\x00')
item_cipher_algo   = create_item(b'\xFF\xFC', b'\x06\x0E\x2B\x34\x04\x01\x01\x07\x02\x09\x02\x01\x01\x00\x00\x00')
item_mic_algo      = create_item(b'\xFF\xFB', b'\x06\x0e\x2b\x34\x04\x01\x01\x07\x02\x09\x02\x02\x01\x00\x00\x00')
item_key_id        = create_item(uuid.uuid4().bytes)
```

Et maintenant, nous concaténons tous nos items pour créer la **Value** de notre KLV et on calcule notre **Length** :

```
value = item_instance_id \
        + item_context_id \
        + item_essence_label \
        + item_cipher_algo \
        + item_mic_algo \
        + item_key_id

# BER format (long-form-coding)
length = b'\x83' + len(value).to_bytes(3, byteorder='big')
```

On rajoute notre **Universal Label** de **Cryptographic Context**, son **Length** et sa **Value** :

```
cryptographic_context_ul = b'\x06\x0E\x2B\x34\x02\x53\x01\x01\x0D\x01\x04\x01\x02\x02\x00\x00'
cryptographic_context = cryptographic_context_ul + length + value
```

Et voila ! Nous avons notre **Cryptographic Context** Passons maintenant à notre plat de résistance : **Encrypted Essence Container**

3/3 - KLV ENCRYPTED ESSENCE CONTAINER

Souvenez-vous, notre **Encrypted Essence Container** est d'un type différent que nos précédents KLV : nous étions sur des KLV **Local Sets** et nous passons à un KLV **Variable-Length**

Pack : nous n'avons plus d'**item.LocalTag**, seulement des **item.Length** et **item.Value**.

Nous avons aussi beaucoup d'items présent : **Cryptographic Context Link**, **Plaintext Offset**, **Source Key**, **Source Length** et **Encrypted Source Value** qui sont la base de notre **Encrypted Essence Container** (nous mettons de côté la partie MIC pour l'instant).

Commençons par **Cryptographic Context Link** avec notre `context_id_uuid` défini dans notre précédent KLV :

```
cryptographic_context_link_value = context_id_uuid
cryptographic_context_link_length = len(context_id_uuid).to_bytes(3, byteorder='big')
item_cryptographic_context_link = b'\x83' \
    + cryptographic_context_link_length \
    + cryptographic_context_link_value
```

En premier, on calcule la taille de notre item : ne pas oublier que contrairement à nos précédents **item.Length**, ceux-là respectent le format BER, donc nous aurons un petit entête BER (`0x83`). Notre item sera donc `0x83000010` et notre UUID.

Pour **Plaintext Offset**, c'est un peu plus compliqué :

```
plaintext_offset_value = int(0).to_bytes(8, byteorder='big')
plaintext_offset_length = len(plaintext_offset_value).to_bytes(3, byteorder='big')
item_plaintext_offset = b'\x83'
    + plaintext_offset_length \
    + plaintext_offset_value
```

Nous ne voulons pas utiliser **Plaintext Offset**, sa valeur sera donc à 0. Par contre, il doit être sur 8 octets. Nous prenons donc notre petit 0 tout seul et on va l'intégrer dans une suite de 8 octets. On aura donc simplement `0x0000000000000000` . Et notre item sera donc `0x830000080000000000000000`

Passons maintenant à Source Key qui est le type de l'essence et comment elle est stockée. Notre valeur sera le label pour un **Picture Essence - Line Wrapped Data, Not Clip Wrapped, J2C Picture** qui est `060e2b34.01020101.0d010301.15010801`

```
source_key_value = b'\x06\x0E\x2B\x34\x01\x02\x01\x01\x0D\x01\x03\x01\x15\x01\x08\x01'
source_key_length = len(source_key_value).to_bytes(3, byteorder='big')
item_source_key = b'\x83' \
    + source_key_length \
    + source_key_value
```

Rien à dire de plus, on a déjà vu cela précédemment.

On va passer à **Source Length** : cet item est la taille de la source, donc la taille de notre frame. Pour notre exemple, on va simplement récupérer la taille de notre fichier JPEG2000. Notez que si vous avez déjà le contenu en mémoire, `os.path.getsize(...)` est inutile.

```
source_size = os.path.getsize("frame.j2c")
source_length_value = source_size.to_bytes(8, byteorder='big')
source_length_length = len(source_length_value).to_bytes(3, byteorder='big')
item_source_length = b'\x83' \
    + source_length_length \
    + source_length_value
```

Comme vous voyez, c'est identique à avant, la seule subtilité étant de récupérer la taille de notre frame JPEG2000.

Nous passons maintenant à notre (gros) item **Encrypted Source Value**. C'est à partir de ce moment que nous allons monter un peu en difficulté. On va jouer avec la cryptographie.

Pour des raisons de simplicité, nous allons faire des raccourcis, comme lire le fichier en entier d'un coup et notre **clef de chiffrement AES** sera simplement que des zéros.

Puis nous définissons notre coeur cryptographique :

```
aes_key = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
iv = os.urandom(16)

cipher = Cipher(
    algorithms.AES(aes_key),
    modes.CBC(iv),
    backend=default_backend()
)
encryptor = cipher.encryptor()
```

Notre **Initialization Vector (IV)** de 16 octets sera généré aléatoirement à chaque appel. Avec notre handler `encryptor` nous pouvons maintenant chiffrer !

On déjà chiffrer notre **CheckValue** : obligatoire, souvenez-vous, sinon notre chiffrement final ne sera pas correct.

```
checkvalue = encryptor.update(b'CHUKCHUKCHUKCHUK')
```

Nous avons déjà notre **CheckValue chiffré**, passons maintenant aux données de l'image, une frame JPEG2000 4K d'une taille de 40136 octets :

```
with open("frame.j2c", "rb") as file:
    source_content = file.read()
```

Nous n'avons fait pas dans la subtilité ni dans l'optimisation, nous avons lu entièrement notre fichier JPEG2000. Et nous chiffrons directement `source_content` avec cependant une petite surprise :

```
pad = b'\x00' * (16 - len(source_content) % 16)
encrypted_data = encryptor.update(source_content + pad)
```

Vous remarquez que nous créons un padding : nous utilisons un **modulo 16** qui va nous donner combien il y a d'octets en trop et nous allons retrancher ce résultat à une taille de bloc de 16, qui nous donnera **le nombre d'octets manquant**, et cela rapidement.

Un bref passage sur le principe de notre calcul (modulo et minus), avec des exemples et une taille de `source_content` égale à 40136 (octets) :

```
#-----------------------
#     Taille 40136
#-----------------------

# On fait un module 16 sur notre taille pour avoir le nombre d'octet dans le dernier bloc :
>>> 40136 % 16
8
# 8 octets dans le dernier bloc,
>>> 16 - 8
8
# on doit donc rajouter 8 octets pour faire un dernier bloc de 16 octets

#-----------------------
#     Taille 40140
#-----------------------

# Prenons une taille de 40140 octets :
>>> 40140 % 16
12
# 12 octets dans le dernier bloc,
>>> 16 - 12
4
# on doit donc rajouter 4 octets pour faire un 16 octets

#-----------------------
#     Taille 40149
#-----------------------

# Prenons une taille de 40149 octets :
>>> 40149 % 16
5
# 5 octets dans le dernier bloc,
>>> 16 - 5
11
# on doit rajouter 11 octets pour faire 16 octets

#-----------------------
#     Exception :)
#-----------------------

# Petite exception, prenons une taille multiple de 16 :
>>> 40144 % 16
0
# On constate que nous n'avons pas d'octet en trop, tout est parfaitement normal
# Sauf que si nous faisons :
>>> 16 - 0
16
# Cela rajoute quand même un bloc de 16 octets :)
# Ce n'est pas très grave dans notre cas car rajouter des octets en plus
# n'a pas beaucoup d'importance tant que nous arrivons à un multiple de 16.
# Le padding sera supprimé lors du déchiffrement.
```

Notez que c'est une approche pour faire rapidement du padding, il en existe d'autres.

Maintenant que nous avons nos principaux items pour **Encrypted Source Value** : **IV**, **CheckValue** et **EncryptedData**, nous pouvons créer notre **Encrypted Source Value** en les concaténant :

```
encrypted_source_value = iv + checkvalue + encrypted_data
```

Il nous reste plus qu'à créer le **item.Length** de **Encrypted Source Value** :

```
encrypted_source_length = len(encrypted_source_value).to_bytes(3, byteorder='big')
item_encrypted_source_value = b'\x83' \
    + encrypted_source_length \
    + encrypted_source_value
```

Nous avons maintenant tous nos items majeurs pour enfin créer la **Value** et le **Length** de notre KLV :

```
# Création de Value
value = item_cryptographic_context_link \
        + item_plaintext_offset \
        + item_source_key \
        + item_source_length \
        + item_encrypted_source_value
# Création du Length
length = b'\x83' + len(value).to_bytes(3, byteorder='big')
```

Il nous reste plus qu'à créer notre KLV avec son **Universal Label** pour **Encrypted Essence Container**, le **Length** et la **Value** :

```
encrypted_essence_container_ul = b'\x06\x0E\x2B\x34\x02\x04\x01\x01\x0D\x01\x03\x01\x02\x7E\x01\x00'
encrypted_essence_container = encrypted_essence_container_ul + length + value
```

Et voila ! Nous avons notre KLV **Encrypted Essence Container** avec notre frame JPEG2000 chiffré à l'intérieur.

Maintenant, nous allons simplement écrire ces trois KLV dans un fichier. Bien entendu, ce fichier ne marchera jamais dans un lecteur MXF lambda, il faudrait ajouter les quelques 20 KLV supplémentaires et nécessaires pour être parfaitement compatible MXF SMPTE.

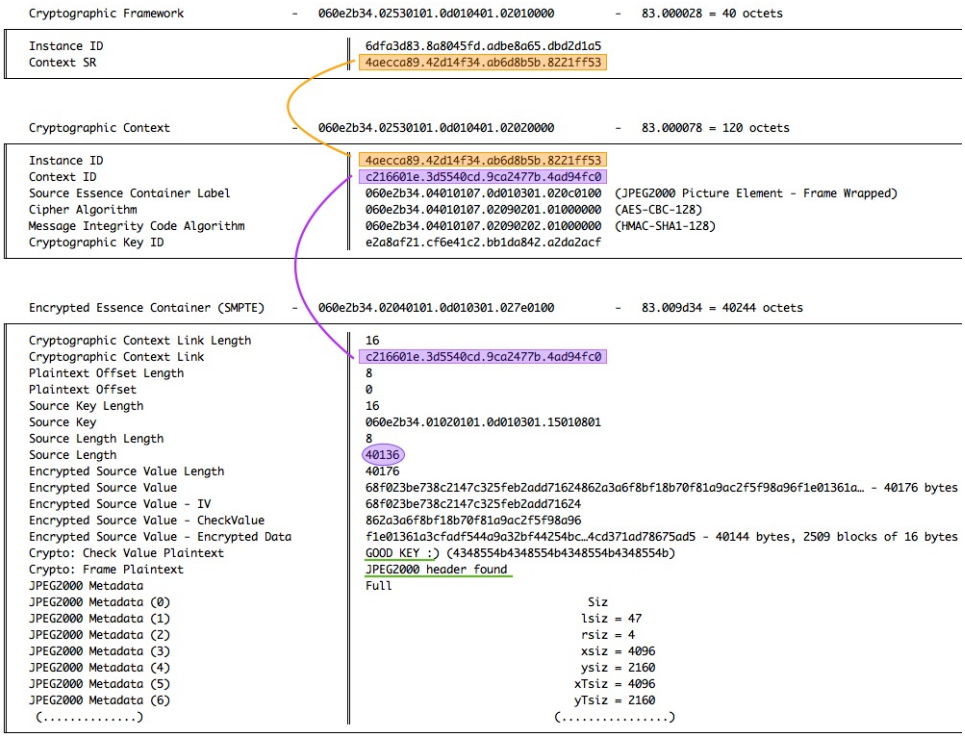
Nous allons créer un fichier de sortir juste pour voir si notre lecteur de MXF `mxf-reader` lit correctement les différents KLV :

```
with open("encrypted-klvs.bin", "wb") as file:
    file.write(cryptographic_framework)
    file.write(cryptographic_context)
    file.write(encrypted_essence_container)
```

Lançons notre programme pour générer ce faux MXF :

```
$ ./mxf-create-klv-encrypted.py
```

On se retrouve avec un fichier `encrypted-klvs.bin` de 40.448 octets que nous allons analyser avec sa clef de déchiffrement :



L'analyse indique que tout s'est déroulé correctement. Nos trois KLV sont correctement placés et avec la clef de déchiffrement, nous avons pu déchiffrer la frame JPEG2000 et en extraire des métadonnées provenant de l'image directement. Nous constatons également que nos différents liens d'UUID sont correctes. Et enfin, nos checksums entre notre JPEG2000 d'origine et celui qui a été extrait sont parfaitement égaux :

```
$ shasum -a 256 "essences/JPEG2000/frame.j2c" "extract.j2c"
b469a8333a8ad708becdfc7544f180c1198b12722a2051b90c66b5ba58ded825  "essences/JPEG2000/frame.j2c"
b469a8333a8ad708becdfc7544f180c1198b12722a2051b90c66b5ba58ded825  "extract.j2c"
```

Nous pouvons réutiliser ce code pour insérer nos KLV cryptographiques dans un MXF complet.

MIC : CRÉATION

A terminer

LE PADDING CRYPTOGRAPHIQUE ET LA LIMITE DE LA BANDE PASSANTE

La spécification DCI mentionne une limite de bitrate, elle est de 250 Mb/s (500 Mb/s pour le HFR et le HDR).

Afin d'avoir la plus grande qualité possible dans l'image, certains laboratoires mettaient la compression du JPEG2000 à son plus bas pour faire correspondre à un bitrate à la limite des 250 Mb/s (ou des 500 Mb/s pour le HFR).

En faisant cela, il arrive parfois un effet de bord : comme nous l'avons vu, la cryptographie AES nécessite un multiple de 16 octets.

Que se passe-t-il si un fichier JPEG2000 n'est pas un multiple de 16 octets ? Un padding va être créée. Dans l'absolu, ce n'est pas grave, ce sont quelques octets en plus dans le KLV.

Mais que se passe-t-il si **beaucoup** de JPEG2000 ne sont pas des multiples de 16 ? Il y aura donc énormément de padding, donc un surplus d'octets dans chaque KLV. Et donc un dépassement de la limite du bitrate.

Cela ne sera quasiment rien, au lieu d'être à 250 Mb/s, vous serez à 250.001 Mb/s par exemple. Souci : certains players refusent catégoriquement de lire ce type de MXF. Votre DCP sera donc rejeté.

C'est pour cela qu'il est conseillé de demander une compression avec une valeur correspondante de bitrate légèrement en dessous de la limite (par exemple ~245 Mb/s - ou 495 Mb/s en HFR) afin de laisser une légère marge si des paddings cryptographiques sont ajoutés lors de la phase de chiffrement des MXF.

CONCLUSION

Et voila, vous savez maintenant (quasiment) tout sur un MXF chiffré :-)

Ce paragraphe est probablement perfectible et des éléments peuvent manquer.

ANNEXE : CODES ET TECHNIQUES

Retrouvez les codes sources et techniques sont disponibles dans une page spécifique : MXF-Codes

ANNEXE : IDENTIFIANTS UL & LABEL

Voici des résumés et explications rapides des différents labels pour la partie cryptographie.

UNIVERSAL LABEL : ENCRYPTED ESSENCE CONTAINER

Universal Label utilisé comme clé pour identifier les KLV chiffrés.

```
06.0E.2B.34.02.04.01.01.0D.01.03.01.02.7E.01.00
---------- || || || || || || || || || || || ++- Reserved
           || || || || || || || || || ||  `---- Encrypted Triplet
           || || || || || || || || || ||  `------- Encrypted Essence
           || || || || || || || || ||  `---------- MXF Generic Container
           || || || || || || ||  `-------------- Version 1
           || || || || || ||  `---------------- Essence Containers
           || || || || ||  `------------------ AAF Association
           || || || ||  `---------------------- Org Registered
           || || ||  `------------------------ Version 1
           || ||  `-------------------------- Set/Pack Dictionary
           ||  `---------------------------- Variable Length Pack
            `------------------------------- Sets/Packs
```

La seule différence entre Interop et SMPTE est l'octet **Version** :

```
06 0e 2b 34 02 04 01 07 0d 01 03 01 02 7e 01 00 (Interop)
06 0e 2b 34 02 04 01 01 0d 01 03 01 02 7e 01 00 (SMPTE)
                  ^^---------------------------------- Version
```

LABEL : ENCRYPTED ESSENCE CONTAINER LABEL

Label utilisé pour le tag "Essence Container" dans les KLV "Partition Pack Header", "Partition Pack Footer" et "Preface". Il ne sert que comme identifiant (label) dans les metadatas.

```
06.0E.2B.34.04.01.01.07.0D.01.03.01.02.0B.01.00
---------- || || || || || || || || || || || ++- Reserved
           || || || || || || || || || || ||  `---- Frame Wrapped
           || || || || || || || || || ||  `------- Encrypted Essence Container
           || || || || || || || || ||  `---------- MXF Generic Container
           || || || || || || || ||  `------------- Version 1
           || || || || || || ||  `---------------- Essence Containers
           || || || || ||  `------------------ AAF Association
           || || || ||  `---------------------- Org Registered
           || || ||  `------------------------ Version 7
           || ||  `-------------------------- Labels Structure
           ||  `---------------------------- Labels Registry
            `------------------------------- Labels
```

LABEL : AES-128-CBC

Label utilisé pour le tag "Cipher Algorithm" dans les KLV "Cryptographic Context"

```
06.0E.2B.34.04.01.01.07.02.09.02.01.01.00.00.00
---------- || || || || || || || || || || || ++- Reserved (16)
           || || || || || || || || || ||  `---- Reserved (15)
           || || || || || || || || || ||  `------- Reserved (14)
           || || || || || || || || ||  `---------- Algorithm Designator AES-128-CBC
           || || || || || || || ||  `------------- Data Encryption Algo
           || || || || || || ||  `---------------- Data Encryption
           || || || || ||  `------------------ Encryption
           || || || ||  `---------------------- Org Registered
           || || ||  `------------------------ Version 7
           || ||  `-------------------------- Labels Structure
           ||  `---------------------------- Labels Registry
            `------------------------------- Labels
```

LABEL : HMAC-SHA1-128

Label utilisé pour le tag "MIC Algorithm" dans les KLV "Cryptographic Context"

```
06.0E.2B.34.04.01.01.07.02.09.02.01.01.00.00.00
---------- || || || || || || || || || || || ++- Reserved (16)
           || || || || || || || || || ||  `---- Reserved (15)
           || || || || || || || || || ||  `------- Reserved (14)
           || || || || || || || || ||  `---------- Algorithm Designator HMAC-SHA1-128
           || || || || || || || ||  `------------- Data Hashing Algo
           || || || || || || ||  `---------------- Data Encryption
           || || || || || ||  `------------------ Encryption
           || || || ||  `---------------------- Org Registered
           || || ||  `------------------------ Version 7
           || ||  `-------------------------- Labels Structure
           ||  `---------------------------- Labels Registry
            `------------------------------- Labels
```

ANNEXE : SAMPLES

Mettre à jour cette liste des assets

Des samples de **MXF chiffrés** :

- MXF encrypted.mxf (clef AES: `00000000000000000000000000000000`)
- MXF encrypted-plaintextoffset.mxf (clef AES: `00000000000000000000000000000000`)
- MXF DCP `DCP-INSIDE-CRYPTE` jp2k_video.mxf (clef AES: `6e256ec2308835ea1d46d8a359296f38`)
- MXF DCP `DCP-INSIDE-CRYPTE` wav_audio.mxf (clef AES: `f5a3d36ab03412984de4aa313199437a`)

Des samples de **KLV** provenant de MXF chiffrés :

- KLVs Encrypted MXF (dir)
- KLVs Encrypted PlaintextOffset MXF (dir)

ANNEXE : DES TAILLES FIXES DANS UN VARIABLE-LENGTH PACK ?

Alors que le KLV est de type Variable-Length Pack et donc que chaque item a son propre **item.length** pour définir la taille variable de sa **item.value**, vous remarquerez que - pour tous les items évoqués ci-dessus - nous avons déjà indiqué leurs tailles.

C'est tout simplement parce que les tailles des **item.value** des items sont déjà fixées dans la norme ! Théoriquement, nous n'aurions pas besoin des **item.length** dans chaque item (sauf si on a pas la documentation, mais dans ce cas, nous ne saurions pas également à quoi correspond tel ou tels items).

Vous remarquerez aussi que l'item **Encrypted Source Value** n'est qu'un containeur pour quatre élements (**IV**, **CheckValue**, **Plaintext Data** et **Encrypted Data** et son padding) mais sans aucun **item.length** propre à eux : IV est obligatoirement à 16 octets donc cela aurait été inutile, cependant Check Value est défini comme un bloc de 16 octets mais aurait pu être différent tout en restant un multiple de 16 octets, par exemple.

Et enfin, vous remarquerez que l'item **Plaintext Offset** est un ersatz de **Length** pour **Plaintext Data** mais qui ne se trouve pas à côté de lui (ce qui aurait pu être le cas, pour respecter la structure d'un **Variable-Length Pack**).

On a une partie de **Variable-Length Pack** mais avec des définitions de tailles fixes comme pour le **Fixed-Length Pack** et une partie qui semble ressembler à du **Fixed-Length Pack** (donc sans **Length**) mais avec une partie variable comme **Encrypted Data**.

C'est pour tout cela que j'ai surnommé ce type de KLV, un **Fucked Pack**.

RÉFÉRENCES

- Digital Cinema System Specification v1.4.3 - Chapitre 9.7.2 - Image and Sound Encryption (AES Encryption, 128 bits, CBC)
- Advanced Encryption Standard (AES) - November 26, 2001. FIPS-197
- Digital Cinema System Specification, 9.7.5. Integrity Check Codes : *« Cryptographic data integrity checksums shall be ensured according to the HMAC-SHA-1 algorithm, as specified in FIPS PUB 198a "The Keyed-Hash Message Authentication Code." (..) The requirements of this section shall be superseded by the FIPS 140-2 or FIPS 140-3 »*

NOTES

1. Si vous tombez sur un **Universal Label** `060e2b34.02040107.0d010301.027e0100` (octet n°8 - Version - à `0x07`), c'est également un **Encrypted Essence Container** mais pour **Interop**. ↩