KDM : AUTHENTICATED PRIVATE

PREFACE

Authenticated Private is the block where the data can only be read by the holder of the RSA private key.

There is the **main element of the KDM**: Here are **stored the AES keys** that allow us to decrypt the encrypted MXFs of the DCP.

The **AuthenticatedPrivate** section follows the **XML Encryption** (xmlenc) archive standard.

INSIDE AUTHENTICATEDPRIVATE

Let's go back to the **AuthenticatedPrivate** section of our own KDM:

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

As you can see, we have a list of several **EncryptedKey** elements each containing the most important element here: the **CipherValue** section.

These include our four AES keys along with their associated metadata.

THE ENCRYPTEDKEYS AND THEIR CIPHERVALUE

Here is an example with a single **EncryptedKey** :

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

We will focus on the data within **CipherValue**, which contains the AES key and its associated metadata.

The **EncryptedKey** element is defined in the XML Encryption - EncryptedKey section.

Within the EncryptedKey element, we have two main child elements :

- **EncryptionMethod** with its algorithm attribute set to **RSA-OAEP-MGF1** (w3c-xmlenc)
- **DigestMethod** with its algorithm attribute set to **SHA1**

The main encryption method is the RSA cryptographic algorithm, meaning we are working with an **asymmetric** encryption.

> **Asymmetric encryption ? a brief reminder**
>
> Previously, we already saw what symmetric encryption is in the chapter AES.
>
> As a reminder, a symmetric encryption needs a single key to both encrypt and decrypt. It's like to open and close a door with a key.
>
> By contrast, the **A**symmetric encryption needs two keys :
>
> - One key for the encryption - also known as public key and can be distributed to everyone without any restriction
> - Another key for the decryption - also known as private key and must be kept secret by its owner and never distributed.
>
> It's not possible to decrypt using the public key. And it's not possible to encrypt using the private key.
>
> (However, it's possible to sign a document using the private key and to verify the signature using the public key. Concept that we will explore in the chapter Signature. For now, it's outside our scope for CipherValue encryption

The other parameters included during the **RSA** encryption are :

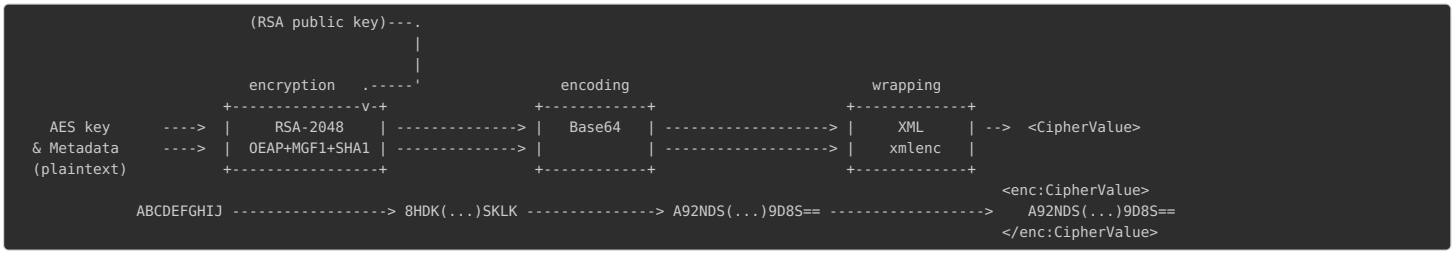
- **OAEP** : Optimal Asymmetric Encryption Padding
- **MGF1** : Mask Generation Function
- **SHA1** : Secure Hash Algorithm

These three algorithms will be used within the RSA encryption process (By contrast with a base64 encoding which it used before or after an encryption)

In our case, the encryption will be processed as follows :

- **Encryption** of the AES keys (and metadata) using the **public key** with the RSA algorithm, along with the Optimal Asymmetric Encryption Padding (OEAP) + Mask Generation Function (MGF1) [1] algorithms, and SHA1 algorithm.
- Encoding the RSA binary output in Base64.
- The base64 output is our **CipherValue**

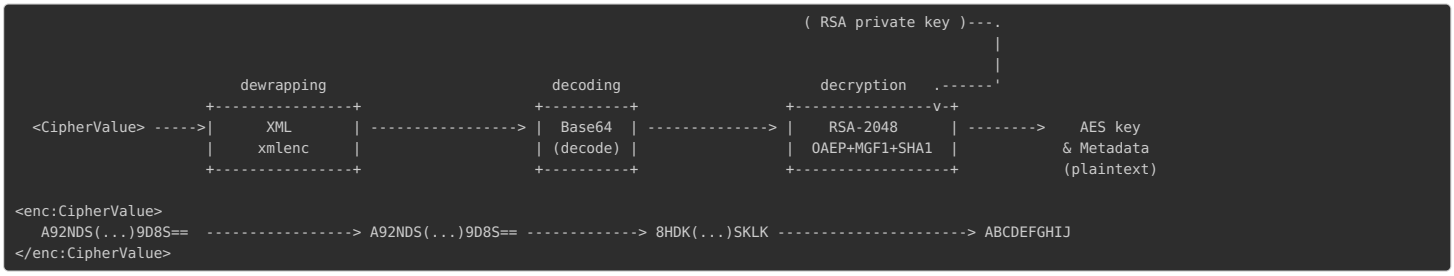
A diagram with the differents steps :



And the decryption will be processed as follows :

- Decoding the **CipherValue** encoded in Base64.
- **Decryption** of the output binary, using the **private key**, with the RSA algorithm, along with the Optimal Asymmetric Encryption Padding (OEAP) + Mask Generation Function (MGF1) [2] algorithms, and SHA1 algorithm.
- The output is a binary form that contains our AES key and its associated metadata.

A diagram with the differents steps :



HOW TO DECRYPT A CIPHERVALUE

We will now focus on the content within the **CipherValue** element and attempt to decrypt it to understand what is behind it.

If we follow the different steps, the first is the **Base64** decoding.

In the example below, we will use OpenSSL, which allows us to view the decryption process step by step (we will look at a Python version afterward)

If we remove the Base64 layer, this is what we get :

```
echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUTOhUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \
    | xxd     # only for the display below
```

And, here's the `xxd` output :

```
 offset        data in hexadecimal format             string format
------------------------------------------------------------------------
00000000:  510e 451c 14a7 0913 35e5 dbea daf5 c64d  | Q.E..▯..5▯▯▯▯▯M
00000010:  cd43 9c02 1416 ff3a f6a8 748d 126f deb7  | ▯C....▯:▯▯t..o▯
00000020:  3994 165a 1129 b00b 1039 e18f 1a45 2ef7  | 9..Z.)▯..9▯..E.▯
00000030:  d82a 0ffc c384 6c1f f1ca 74d7 6da5 7575  | ▯*.▯▯.l.▯▯t▯m▯uu
00000040:  a78e 0ca3 c1a5 da57 e2cb 634b 80f9 a0ed  | ▯..▯▯▯▯W▯▯cK.▯▯
00000050:  43f0 feff 7b88 938a dd7d 6321 09ee ca61  | C▯▯▯{...▯}c!.▯▯a
00000060:  9893 1535 b242 98a9 ee12 95a4 1b36 5891  | ...5▯B.▯▯..▯.6X.
00000070:  7f74 d4c2 038b 162c 3860 b954 764c c114  | .t▯▯...,8`▯TvL▯.
00000080:  20dc 1731 66e8 fc5e 817f 5461 a16f db8e  | ▯.1f▯▯^..Ta▯o▯.
00000090:  916d ef8a c6e2 fcaa 45e4 b45c 7e36 ece4  | .m▯.▯▯▯E▯\~6▯▯
000000a0:  56c3 422a 5b56 7bba 4dc6 d1f2 2d0c faf1  | V▯B*[V{▯M▯▯▯-.▯▯
000000b0:  d2f8 a7fc e16b f433 4a1c 4d2f 9251 33a1  | ▯▯▯▯k▯3J.M/.Q3▯
000000c0:  533c 62ff 7700 ab1b 771a 4aa7 6afd 8786  | S<b▯w.▯.w.J▯j▯..
000000d0:  d508 8108 4a26 4999 cb1d 7529 4917 1427  | ▯...J&I.▯.u)I..'
000000e0:  197b 9548 e56c 18a2 aa22 051d ae78 de40  | .{.H▯l.▯▯"..▯x▯@
000000f0:  9289 a040 a341 d135 9143 2b8b f31c 2757  | ..▯@▯A▯5.C+.▯.'W
```

This output isn't understandable at all, it's normal. This is our cryptographic content - our **raw RSA binary data**

We will jump to step 2 :

RSA decryption with the **OAEP**, **MGF1**, and **SHA1** parameters along with **our RSA private key**.

For this, we'll reuse the previous output and we add our RSA decryption step using `openssl pkeyutl -decrypt` and the appropriate parameters (`-pkeyopt`) :

```
$ echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUTOhUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \
    | openssl pkeyutl \
        -decrypt \
        -inkey private_key.pem \
        -pkeyopt rsa_padding_mode:oaep \
        -pkeyopt rsa_oaep_md:sha1 \
        -pkeyopt rsa_mgf1_md:sha1 \
    | xxd     # only for the display below
```

Here's the `xxd` output :

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

Now, you'll notice some clearly readable elements in the last column, such as a datetime or this 'MDEK'.

As mentioned earlier, **CipherValue** contains not only the AES key but also its associated metadata : it's a binary structure with several fixed-size data fields that needs to be parsed and analyzed in order to extract the relevant information.

Here is the definition of the **Plaintext CipherValue internal structure** with the fields and size of each one :

Name of the field	Position	Size	Format	
Structure ID	0	16 octets	UUID	Fixed-size Identifier which always is `f1 dc 12 44 60 16 9a 0e 85 bc 30 06 42 f8 66 ab`
Certificate ThumbPrint	16	20 octets	Hash	The thumbprint of public certificate of our encoder (Signer) (don't confuse with Public Key Thumbprint)
CPL Id	36	16 octets	UUID	The CPL identifier (Id) that is ued
Key Type	52	4 octets	String	The type of the key (eg. MDEK)
Key Id	56	16 octets	UUID	The AES key identifier
Date Not Valid Before	72	25 octets	Datetime	The date validity of the KDM (just for your information)
Date Not Valid After	97	25 octets	Datetime	The date validity of the KDM (just for your information)
AES key	122	16 octets	AES	The precious decryption key of our MXF
Total		**138 octets**		

In total, the data structure is 138 octets long.

But why is it 138 octets long, and not 256 octets ?

Given that we use 2048-bit RSA keys (256 bytes long), the **CipherText** (our **CipherValue** without the Base64 layer - i.e. the raw RSA binary data) will be 256 octets long.

```
printf 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRJv3rc5lBZaESmw
CxA54Y8aRS732CoP/MOEbB/xynTXbaV1daeODKPBpdpX4stjS4D5oO1D8P7/e4iTit19YyEJ7sph
mJMVNbJCmKnuEpWkGzZYkX901MIDixYsOGC5VHZMwRQg3BcxZuj8XoF/VGGhb9uOkW3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0ocTS+SUTOhUzxi/3cAqxt3GkqnavZHhtUI
gQhKJkmZyx11KUkXFCcZe5VI5WwYoqoiBR2ueN5AkomgQKNB0TWRQyuL8xwnVw==' \
    | openssl base64 -d \        # removing the base64 layer to have the encrypted RSA content
    | wc -c                      # calculating the number of octets in output
256 # octets
```

We have a 42-octets OAEP headers, leaving 214 octets available for our structure (whether encrypted or not) [3].

Here, we can see that only 128 octets out of 214 octets available are used.
This means that 76 octets of additional metadata can be added if necessary (don't do that :)

Note that RSA-2048 performs encryption in 256-octets blocks, but the CipherText could be larger, multiple of 2048. But we are out-of-scope of SMPTE standard of the KDM (for now, maybe in the future ?)

If we'll reuse the previous output and we highlight the different elements using few colors :

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

So, we have the data structured like this :

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

We have just successfully decrypted our first **EncryptedKey** !

All that's left is to perform exactly the same procedure on all the **EncryptedKey** elements.

Note that among all the previous values in our metadata, the only truly useful data is our AES key, the rest being metadata that we can ignore (if we do not intend to verify and validate a KDM or if we do not comply with the verification requirements standardized by SMPTE and the DCI-CTP [4]. Otherwise, this data will be useful to validate our KDM)

WHAT IS THE LIMIT ON THE NUMBER OF KEY (ENCRYPTEDKEY) IN A KDM ?

According to the **SMPTE 429-2 - DCP Operational Constraints**, they don't have more than 256 distinct cryptographic keys, as uniquely identified by their Key ID [5]. However, this rule is in the CPL Constraints, not KDM Constraints.

In the **SMPTE 430-1 - KDM** and **SMPTE 430-3 - ETM** standards, there are no rules regarding this [6]. We can therefore assume that there are no limitations. However, it's possible that the player and encoder impose a limit of 256 keys, as specified by the CPL Constraints. But technically, you can generate a KDM with an unlimited number of keys.

WHY IS THE STRUCTURE ID FIXED ?

The standard impose a value for the StructureID field et must be `f1dc124460169a0e85bc300642f866ab`

According to the standard, this choice is explained by the need to prevent a certain type of attack on the KDM :

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

This allows also to quickly check if the data has been correctly decrypted; It is enough to read the 16 first octets and verify it if it matches our identifier.

WHAT IS THE CERTIFICATE THUMBPRINT ?

The **Certificate Thumbprint** is often confused with the **Public Key Thumbprint**.

Even though both use the public certificate, both don't take the same information from the certificate to create their thumbprints.

For further information, take a look at the chapter Certificats paragraph Certificate Thumbprint and Public Key Thumbprint.

READING CIPHERVALUE

Here is a very simple code example that reads all the **CipherValues** in a KDM and decrypts them :

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

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

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

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

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

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

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

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

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

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

# read each private cipherValue
for cipher_value in cipher_values:

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

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

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

CREATE A CIPHERVALUE

To create a CipherValue, you just need to do the exact opposite work :)

Some useful codes :

- An example of a Python program that encrypts a CipherValue : ciphervalue-write.py (excluding its structure)
- An example of code to create a CipherValue (excluding its structure) and its commandline version with OpenSSL, in the chapter KDM Codes

SUBCHAPTERS

- **KDM - Authenticated Public** : the non-encrypted part and readable by anyone
- **KDM - Authenticated Private** : the encrypted part and readable only by the receiver
- **KDM - Signature** : our part in authenticating and validating the KDM

- **KDM - Codes** : Source code and technical tips on each part of the KDM

RELATED CHAPTERS

- **Cryptography** : To understanding the cryptography used in digital cinema.
- **Certificate** : The certificates used in digital cinema.
- **AES Cryptography** : The symmetric cryptography used to encrypt the MXF.
- **RSA Cryptography** : The asymmetric cryptography used in KDM to encrypt AES keys (among others things).

NOTES

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