

DÉCHIFFRER UNE CIPHERVALUE

RÉCUPÉRATION DU CONTENU RSA BRUT (256 OCTETS)

```
echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRjv3rc5lBZaESmw
CxAS4Y8aRS732CoP/M0EBB/xynTXbaV1dae0DKPBdpX4stjS4D5o01D8P7/e4iTt19YyEJ7sph
mJMVNBjCmKnuEpWkGzZYkX901MIDixYs0GC5VHZMwRQg3BcxZuj8XoF/VGGhb9u0Kw3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0octS+SUT0hUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZeSVI5WwYoqoiBR2ueN5AkongQKNB0TWRQyuL8xwnVw==' \
| openssl base64 -d \
| xxd      # sert uniquement pour l'affichage ci-dessous
```

Résultat :

offset	données au format hexadécimal	format string
00000000	51 0e 45 1c 14 a7 09 13 35 e5 db ea da f5 c6 4d	Q.E..0..5000000M
00000010	cd 43 9c 02 14 16 ff 3a f6 a8 74 8d 12 6f de b7	0C...0:t..o[]
00000020	39 94 16 5a 11 29 b0 0b 10 39 e1 8f 1a 45 2e f7	9..Z.)0..90..E.0
00000030	d8 2a 0f fc c3 84 6c 1f f1 ca 74 d7 6d a5 75 75	0*.00.l.00t0m0uu
00000040	a7 8e 0c a3 c1 a5 da 57 e2 cb 63 4b 80 f9 a0 ed	0..0000w00cK.00
00000050	43 f0 fe ff 7b 88 93 8a dd 7d 63 21 09 ee ca 61	C000{...0}c!.00a
00000060	98 93 15 35 b2 42 98 a9 ee 12 95 a4 1b 36 58 91	...50B.00..0.6X.
00000070	7f 74 d4 c2 03 8b 16 2c 38 60 b9 54 76 4c c1 14	.t00...,8`0TvL0.
00000080	20 dc 17 31 66 e8 fc 5e 81 7f 54 61 a1 6f db 8e	0.lf00^..Ta000.
00000090	91 6d ef 8a c6 e2 fc aa 45 e4 b4 5c 7e 36 ec e4	.m0.000E0\~600
000000a0	56 c3 42 2a 5b 56 7b ba 4d c6 d1 f2 2d 0c fa f1	V0B*[V{0M000-.00
000000b0	d2 f8 a7 fc e1 6b f4 33 4a 1c 4d 2f 92 51 33 a1	0000k03J.M/.Q30
000000c0	53 3c 62 ff 77 00 ab 1b 77 1a 4a a7 6a fd 87 86	S<b0w..0.w.J0j0..
000000d0	d5 08 81 08 4a 26 49 99 cb 1d 75 29 49 17 14 27	0...JGI.0.u)I..'
000000e0	19 7b 95 48 e5 6c 18 a2 aa 22 05 1d ae 78 de 40	.{.H0l.00"..0x0@
000000f0	92 89 a0 40 a3 41 d1 35 91 43 2b 8b f3 1c 27 57	..000A05.C+.0.'W

Ceci est notre contenu cryptographique **RSA binaire brute**. Sortie de 256 octets (lié au certificat RSA 2048 bits / 256 octets)

DÉCHIFFRER UNE CIPHERVALUE

DÉCHIFFREMENT COMPLET (RSA+OAEP+MGF1+SHA1)

Ce script nécessite la **clef privée** RSA du player (Recipient)

Utilisation de `openssl pkeyutl -decrypt` et des options adéquates pour l'utilisation d'OAEP, MGF1 et SHA1 :

```
$ echo -ne 'UQ5FHBSnCRM15dvq2vXGTc1DnAIUFv869qh0jRjv3rc5lBZaESmw
CxAS4Y8aRS732CoP/M0EBB/xynTXbaV1dae0DKPBdpX4stjS4D5o01D8P7/e4iTt19YyEJ7sph
mJMVNBjCmKnuEpWkGzZYkX901MIDixYs0GC5VHZMwRQg3BcxZuj8XoF/VGGhb9u0Kw3visbi/KpF
5LRcfjbs5FbDQipbVnu6TcbR8i0M+vHS+Kf84Wv0M0octS+SUT0hUzxi/3cAqxt3Gkqnav2HhtUI
gQhKJkmZyx11KUkXFCcZeSVI5WwYoqoiBR2ueN5AkongQKNB0TWRQyuL8xwnVw==' \
| openssl base64 -d \
| openssl pkeyutl \
  -decrypt \
  -inkey private_key.pem \
  -pkeyopt rsa_padding_mode:oaep \
  -pkeyopt rsa_oaep_md:sha1 \
  -pkeyopt rsa_mgf1_md:sha1 \
| xxd      # sert uniquement pour l'affichage ci-dessous
```

Résultat :

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

Structure des données :

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

OUVRIR TOUS LES CIPHERVALUE D'UN KDM :

Ce script nécessite la **clef privée** RSA du player (Recipient)

Ce simple [script](#) va lire un KDM, analyser chaque **CipherValue**, les déchiffrer - à l'aide de la clef privée [RSA](#) - et générer une sortie des différentes métadonnées et des clefs [AES](#) :

```

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
pad = 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 = pad
    )

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

```

CRÉER UNE CIPHERVALUE (SANS STRUCTURE)

Ce script nécessite le **certificat public** RSA du player (Recipient)

Attention, cet exemple ne génère pas la structure **CipherValue** normée SMPTE pour un KDM, vous devrez compléter le PLAINTTEXT pour intégrer les différents éléments manquants nécessaires et définis dans la norme SMPTE

Ce [script](#) va créer un **CipherValue** respectant les différents algorithmes demandés par la norme KDM du SMPTE (sans sa structure), ainsi nous aurons notre suite OAEP + MGF1 + SHA1 + Base64 :

```

from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
from cryptography.x509 import load_pem_x509_certificate
import base64

PUBLIC_CERTIFICATE = "public_certificate.pem"
PLAINTEXT = b"\xab\xad\xca\xfe\xab\xad\xca\xfe\xab\xad\xca\xfe\xab\xad\xca\xfe"

# read public certificate
with open(PUBLIC_CERTIFICATE, "rb") as file:
    public_certificate_content = file.read()

# load public certificate and extract public key
certificate = load_pem_x509_certificate(
    data = public_certificate_content,
    backend = default_backend()
)
public_key = certificate.public_key();

# create padding OAEP and SHA1
pad = padding.OAEP(
    mgf = padding.MGF1(algorithm=hashes.SHA1()),
    algorithm = hashes.SHA1(),
    label = None
)

# RSA encryption
ciphertext = public_key.encrypt(
    plaintext = PLAINTEXT,
    padding = pad
)

# Base64 encryption
ciphertext64 = base64.b64encode(ciphertext)

print("Ciphertext - RSA =", ciphertext.hex())
print("Ciphertext - Base64 =", ciphertext64)

```

CRÉER UNE CIPHERVALUE (VIA OPENSSL, SANS STRUCTURE)

```

printf "HELLOWORLD" \
| openssl pkeyutl \
  -encrypt \
  -pubin \
  -inkey public_key.pem \
  -pkeyopt rsa_padding_mode:oaep \
  -pkeyopt rsa_oaep_md:sha1 \
  -pkeyopt rsa_mgf1_md:sha1 \
| openssl base64 -e

```

XMLSEC: LA BASE DES SIGNATURES NUMÉRIQUES DU KDM

Voici [exemple de XML valide](#) simplifié (non SMPTE/DCI) et respectant les normes suivantes:

- [XML Signature \(xmldsig\)](#)
- [XML Encryption \(xmldcore-sec\)](#)
- [Canonical XML \(C14N\)](#)

Voici le template XML avant signature :

```

<?xml version="1.0" encoding="UTF-8"?>
<Test>

  <Foo Id="Bar">
    <!-- metadatas -->
  </Foo>

  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
      <ds:Reference URI="#Bar">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#sha256"/>
        <ds:DigestValue/>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue/>
  </ds:Signature>

</Test>

```

Nous allons créer le **DigestValue** pour **Foo** et créer la **SignatureValue** en même temps avec l'outil [xmlsec](#) :

```
# Passage à xmlsec avec notre clef privée :
xmlsec1 \
  --sign \
  --id-attr:Id Foo \
  --privkey-pem private-key.pem \
  xmlsec.sample.xml
```

N'oubliez pas d'avoir la clef RSA privée sous la main (ou d'en [générer une](#)).

Et voici la sortie de l'outil [xmlsec](#) avec nos **DigestValue** et **SignatureValue** complétés :

```
<?xml version="1.0" encoding="UTF-8"?>
<Test>

  <Foo Id="Bar">
    <!-- metadata -->
  </Foo>

  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
      <ds:Reference URI="#Bar">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#sha256"/>
        <ds:DigestValue>p35xH4WZ5LEYS6r20H/ewLUjtQn0p8Ce9k7khaP2qfM=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>QoFMZCMwwav9hFP3FcZdq30aGb4Zn/pKirzBAF+Aoj2fhLv89pul/pu/mARcuNRX
+t2WrxF0BjHMKR4dx9j/+GaQ0FRkqmY5gg9QzxBLG7NZSHiEBSZgYtY+P+0vjuYR
/hpulS3EHAvg40aKkN8eAhQgNVaSsvFTmC8NDAY+k7hWukwLlNrIumvCt/bXWiBV
60kiZ+sCglwt/KugPbmF0LiyAakwL+kx67G8LD8BwLLG2ySSTY4BAMp8IL9QSKB9
PFI/ADjKr0mrDbch6N40spqCREIC0SGdMnN1ZPw4S8v+MGuIiVZYxdA8IioerE4Q
WpHqx9rw6VLo0IshPpwP+w=</ds:SignatureValue>
  </ds:Signature>

</Test>
```

A partir de là, on peut faire la même chose en respectant les normes SMPTE/DCI pour générer ou vérifier des signatures d'un KDM.

GÉNÉRER LES SIGNATURES (DIGESTVALUE + SIGNATUREVALUE) D'UN KDM

La clef privée à utiliser ici dans un workflow DCI est celle de l'encodeur KDM, pas celui du player.

Il faut un template KDM avec les différents tag **Reference** et **Signature** conditionnés et préparés pour le passage des outils comme l'outil [xmlsec1](#).

Exemple de XML prêt à l'emploi, les parties **Authenticated** ont été vidés pour une meilleure compréhension :

```
<?xml version="1.0" encoding="UTF-8"?>
<DCinemaSecurityMessage xmlns="http://www.smpte-ra.org/schemas/430-3/2006/ETM">

  <AuthenticatedPublic Id="ID_AuthenticatedPublic">
    <!-- all metadatas -->
  </AuthenticatedPublic>

  <AuthenticatedPrivate Id="ID_AuthenticatedPrivate" xmlns:enc="http://www.w3.org/2001/04/xmenc#">
    <!-- all encrypted Keys -->
  </AuthenticatedPrivate>

  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>

      <!-- for the block AuthenticatedPublic -->
      <ds:Reference URI="#ID_AuthenticatedPublic">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
        <ds:DigestValue/>      <!-- empty -->
      </ds:Reference>

      <!-- for the block AuthenticatedPrivate -->
      <ds:Reference URI="#ID_AuthenticatedPrivate">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
        <ds:DigestValue/>    <!-- empty -->
      </ds:Reference>

    </ds:SignedInfo>

    <!-- Signature (all DigestValue) -->
    <ds:SignatureValue />    <!-- empty -->

  </ds:Signature>

</DCinemaSecurityMessage>
```

Vous retrouverez un template complet prêt à l'emploi ici : [KDM.template-xmlsec.xml](#)

Maintenant, nous passons ce KDM sans empreintes **DigestValue** ni **SignatureValue** à l'étape de la signature :

```
# Signature d'un KDM - avec sa clef privée
xmlsec1 sign \
  --id-attr:Id AuthenticatedPublic \
  --id-attr:Id AuthenticatedPrivate \
  --privkey-pem encoder_private_key.pem \
  KDM.xml
```

N'oubliez pas d'avoir la clef RSA privée sous la main (ou d'en générer une).

Résultat (seulement de la partie **Signature**, le reste ne changeant pas) :

```
(...)
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
    <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
    <ds:Reference URI="#ID_AuthenticatedPublic">
      <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
      <ds:DigestValue>qN4dvJpemd94ppazl6ii6nmo9JfLBczdpT9yXb3ltow=</ds:DigestValue>
    </ds:Reference>
    <ds:Reference URI="#ID_AuthenticatedPrivate">
      <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
      <ds:DigestValue>9bowz05/W7f4qTJf04K1VXTYEI14uQgJDYr6Z1uP/Ho=</ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>FylhfvacbwQ8mcxLiGI3B6HL0EcZuISQBYd+Ebkrll4oWs5RUaKnq4GA6o6+LE2m
yNf20dNcJ/7IKPvrDw8NFXR7KvRDWcJa9CGaXd87uxFpsUiBBj3u9Q/EIM4gaBH/
RaaRsy0tKmEenguo6JWMVBBLE20bfLd0rBirpIyTbaIDUCyiUaI4qLrxR0uhuHvJ
gTejDcNbznGpn4esFjcZHT0/C6EDW1U/N3t+AG0cCCjYBf80dIoA0luhVNyglWtV
DNew02sMtYuWc7m1swzjYqiBk+INkHnPrUvRsxZgzWoo3XGfgbXr15e2TY/IFN2C
7bdJ5r6vXpB4dPfHThNxmg==</ds:SignatureValue>
</ds:Signature>
(...)
```

Vous avez un KDM signé :-)

VÉRIFIER LES SIGNATURES (DIGESTVALUE + SIGNATUREVALUE) D'UN KDM

La clef publique à utiliser ici dans un workflow DCI est celle de l'encodeur KDM, pas celui du player.

```
# Vérification d'un KDM - avec une clef publique
xmlsec1 verify \
  --id-attr:Id AuthenticatedPublic \
  --id-attr:Id AuthenticatedPrivate \
  --pubkey-cert-pem public-certificate.pem \
  KDM.xml
```

VÉRIFIER LES SIGNATURES (VIA OPENSSSL)

```
openssl base64 -d -in signature.txt -out signature.txt.sha256
openssl dgst -sha256 -verify public-key.pem -signature signature.txt.sha256 filename
```

```
openssl dgst -sha256 -sign private-key.pem -out filename.sha256 filename
openssl base64 -in filename.sha256 -out signature.sha256
```

CRÉER UN DIGESTVALUE (VIA OPENSSSL)

```
printf '<Foo Id="Bar"></Foo>' | openssl sha256 -binary | openssl base64
r4kvmLusMibh32vkz03cxPDWcX/PUj5g10J146Z80LM=
```

Verification avec xmlsec1 :

```
<?xml version="1.0" encoding="UTF-8"?>
<Test>

  <Foo Id="Bar"></Foo>

  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
      <ds:Reference URI="#Bar">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
        <ds:DigestValue/>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue/>
  </ds:Signature>

</Test>
```

Output :

```
<?xml version="1.0" encoding="UTF-8"?>
<Test>

  <Foo Id="Bar"/>

  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
      <ds:Reference URI="#Bar">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
        <ds:DigestValue>r4kvmLusMibh32vkz03cxPDWcX/PUj5g10J146Z80LM=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>MXDe3khop+BGRsSQBe4A3+F+mYfrskMAN15G20bHg7oCZhzzr+Yx5UZrm3LDPduS
6UUUfBymoE7mcXhSpGrzeYca4xG3cKJvKEBC0TDeQefk9XGTt8fEa/Z2mCs/ZLp
QpnyVz9Ufq2lM9Eo4oeRkV69dBY7u0Y10p+i02ImVpyWKktim8wEvCAVp/sDaLIo
l46zdjpiynhg2pLNjdZhsC+lilIUcD9nDe/6BLF23d4UsjjRT0n4su1pT+VQqoCJ
6xPw4Sm0RkEpKKA4pupwk8t6pXbEs7HbvphqHsxqgkDR2sJg3Dk8d4HMLbnmp70q
3AtBAGruDVjRswxwnX7acQ==</ds:SignatureValue>
  </ds:Signature>

</Test>
```

Notons que l'encodage utilise la version longue (`<Foo Id="Bar"></Foo>`) qui va être utilisé lors du sha256+base64 pour générer `r4kvmLusMibh32vkz03cxPDWcX/PUj5g10J146Z80LM=` mais que l'output xml de xmlsec1 va être `<Foo Id="Bar"/>`

CRÉER UNE DIGESTVALUE (VIA OPENSSSL)

```
openssl dgst -sha256 \
  -binary KDM-DigestValue-Foo.xml \
  | openssl base64

"r4kvmLusMibh32vkz03cxPDWcX/PUj5g10J146Z80LM="
```

```
<Foo Id="Bar"></Foo>
```

CRÉER UNE SIGNATUREVALUE (VIA OPENSSL)

```
openssl dgst -sha256 \  
-sign sample_private_key_non-smpte.pem \  
-binary KDM-SignatureValue-SignedInfo.xml \  
| openssl base64  
  
"NaAMJplu6vSiLFsHZ050Rr9adFJrhTZ33hnT5XS/elIAQtt0H30izLbNoFTYK/K  
vbyxZfxpTsMoTxr1gz7v/01dNryZuxe19APRv0BMPCdoZlwzuXTb8X6udLbAkFk  
0GSwmhLu0vNgg3tpdPqH8nT/bxAPgx2gEJ0o/4r7VX3JDlZMQiL5+Yt+PJIMtbZ1  
0HZGEZvAWLfkIeYV5IonFaRijWCqNl8P7x3UzlnSGKNfd7N71RZJ5HLI8bI57Cj0  
9uRzmDh7z3ndPXRUCe6B5x2efS46/lUzk13Xzgjsbq1Qg9hi/NzgbT0wpZc6oq  
8GjNuWstqwXVC4fM8SkjKA=="
```

Fichier [KDM-SignatureValue-SignedInfo.xml](#)

```
<ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">  
  <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"></ds:CanonicalizationMethod>  
  <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"></ds:SignatureMethod>  
  <ds:Reference URI="#Bar">  
    <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#sha256"></ds:DigestMethod>  
    <ds:DigestValue>r4kvmLusMibh32vkz03cxPDWcX/PUj5g10J146Z80LM=</ds:DigestValue>  
  </ds:Reference>  
</ds:SignedInfo>
```

(attention aux conversions `\s` -> `\t` et aux CRLF)

CANONICALIZE XML C14N (PYTHON)

```
#!/usr/bin/env python3  
from lxml import etree  
  
#  
# SignedInfo need Comments on canonicalized output  
# All <!-- --> are conserved  
#  
"""  
Snippet : C14N from file to file (withComments)  
with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:  
    etree.canonicalize(  
        from_file = "inputfile.xml",  
        out = out_file,  
        with_comments = True  
    )  
"""  
  
# Quick XML opening  
with open("KDM-SignatureValue-SignedInfo-withComments.xml", "rb") as xml:  
    tree = etree.fromstring(  
        text = xml.read()  
    )  
  
# Read SignedInfo block  
ap = tree.xpath("//*[@local-name()='SignedInfo']")  
  
# Go to canonicalization with Comments  
ap_c4n = etree.canonicalize(etree.tostring(ap[0]).decode('utf-8'), with_comments=True) # with_comments !  
  
# Display output  
print(ap_c4n)  
  
# Write output  
print("writing c14n.xml")  
with open("c14n.xml", "w") as file:  
    file.write(ap_c4n)
```

Références Python :

- **withComments:** `lxml.tostring()` : <https://lxml.de/api/lxml.etree-module.html#tostring>
- **withComments:** `lxml.write_c14n()` : https://lxml.de/api/lxml.etree.ElementTree-class.html#write_c14n

SOUS CHAPITRES

- **KDM - Authenticated Public** : notre partie lisible par tous

- **KDM - Authenticated Private** : notre partie lisible uniquement par le récepteur
- **KDM - Signature** : notre partie pour authentifier et valider le KDM
- **KDM - Codes** : Codes sources et techniques sur les différentes parties du KDM

CHAPITRES CONNEXES

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