

# Cryptographie pratique

Sébastien Millet

Version 0.5, February 29th, 2016

# Table of Contents

1. Introduction .....	1
2. Le format x509 .....	2
2.1. Visualisation d'un certificat x509 .....	2
2.2. Structure d'un certificat x509 .....	4
2.3. Hiérarchie des certificats .....	5
3. Vérification de signature RSA .....	8
3.1. La signature RSA .....	8
3.1.1. Calcul de la signature .....	8
3.1.2. Vérification de la signature .....	8
3.2. Vérification du certificat de letsencrypt.org .....	8
3.2.1. Choix d'un programme de calcul .....	8
3.2.2. Enregistrement de la signature sous forme d'entier .....	10
3.2.3. Enregistrement de la clé publique sous forme d'entier .....	12
3.2.4. La fonction powmod .....	13
3.2.5. Calcul de $M$ .....	14
3.2.6. Analyse de $M$ .....	15
3.2.7. Calcul de $M'$ .....	19
3.3. Conclusion .....	28
4. Bibliography .....	30

# 1. Introduction

Cet article décrit les calculs à réaliser pour vérifier des signatures RSA et ECDSA, dans le cas des certificats x509. Ces certificats sont couramment utilisés et servent notamment à la sécurisation https.

- *RSA* : ce cas sera traité avec le certificat d'un serveur https.
- *ECDSA* : nous commencerons par travailler sur un certificat auto-signé que nous créerons pour l'occasion, afin de se familiariser avec les calculs qu'implique la cryptographie à courbes elliptiques. Pour finir nous récupérerons un certificat CEV (Certificat Électronique Visible). Ce dernier cas sort du cadre x509 qui est l'objet principal de cet article, le format pour un tel certificat étant 2D-Doc.

Au fil du document nous ferons appel aux outils suivants :

- `openssl` pour travailler sur les certificats x509 en ligne de commande
- `python`, `bc` ou `zsh` pour faire des calculs avec des nombres entiers de grande taille
- Conversion entre encodage *PEM* et encodage *binaire* (Linux : `base64`, Windows : `notepad++`)
- Édition du contenu de fichier binaire (Linux : `gvim/xxd`, Windows : `notepad++`)

## *Windows versus Linux*

Ce document s'adresse aux utilisateurs de Windows et Linux.

### NOTE

Il peut arriver qu'entre les deux environnements l'outil ou la commande à employer soit différente. Dans ce cas la solution pour chaque système est présentée.

## 2. Le format x509

### 2.1. Visualisation d'un certificat x509

A l'aide d'un navigateur, ouvrir une page en https et afficher le certificat. Les exemples de ce document sont réalisés avec le certificat https du site <https://letsencrypt.org/>.

*Exemple avec Firefox 44*

1. Cliquer sur l'icône de cadenas à gauche de la barre d'adresse et cliquer sur la flèche droite (Figure 1)
2. Cliquer sur *Plus d'informations* (Figure 2)
3. Cliquer sur *Afficher le certificat* (Figure 3)
4. Afficher l'onglet *Détails* et parcourir les différents champs du certificat (Figure 4, 5 et 6)

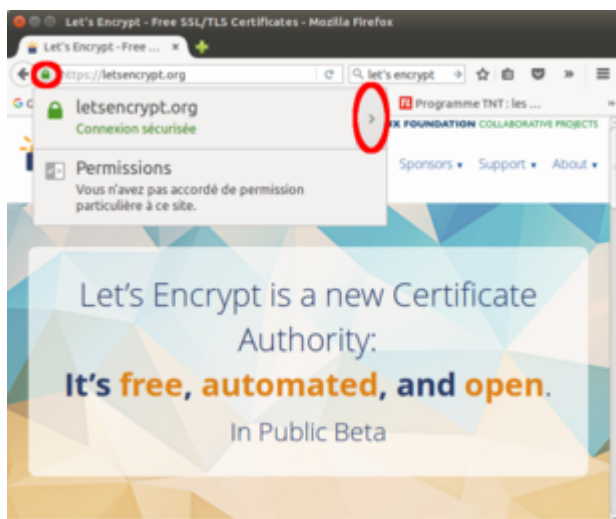


Fig. 1

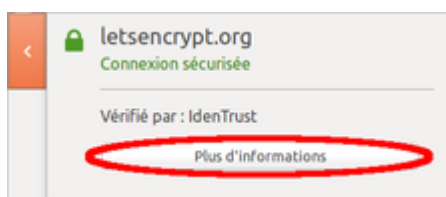


Fig. 2

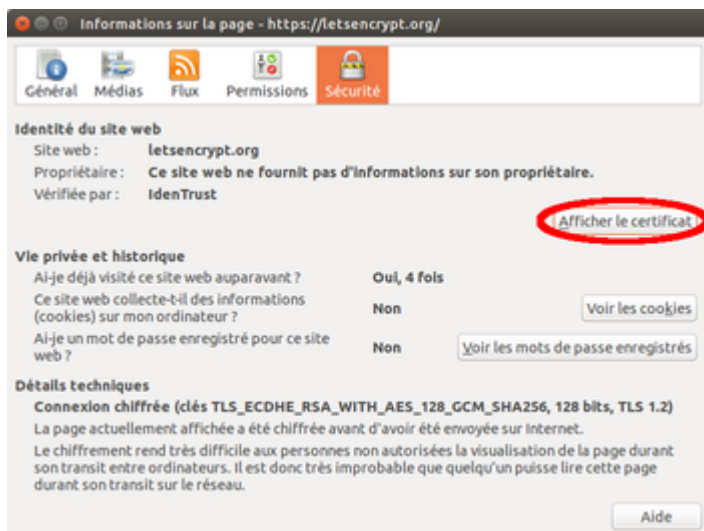


Fig. 3

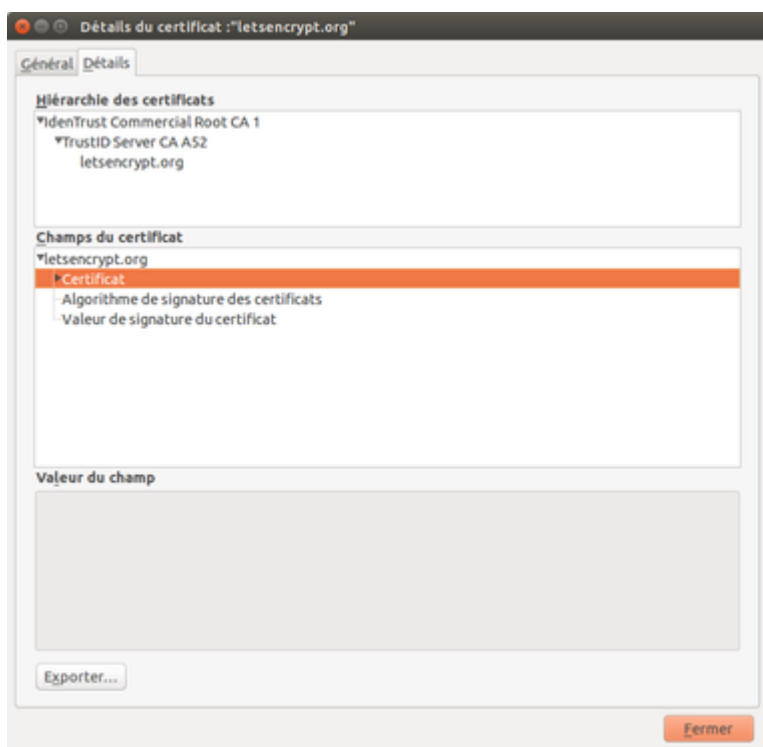


Fig. 4

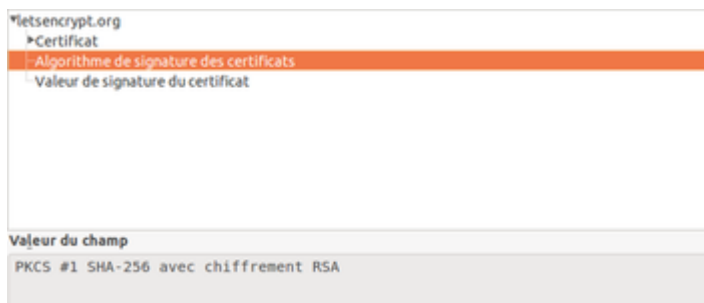


Fig. 5

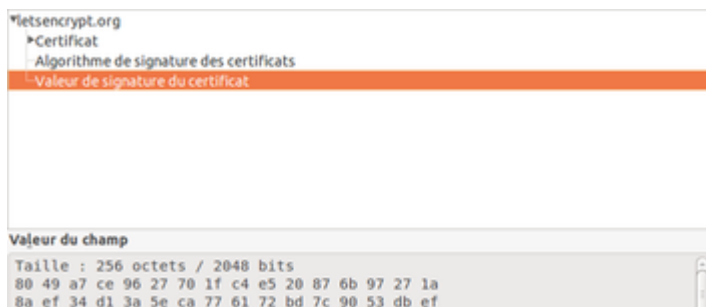


Fig. 6

Nous nous intéresserons à la partie supérieure (*Hiérarchie des certificats*) plus tard.

Pour le moment examinons le certificat. L'affichage de Firefox en dessous de *Champs du certificat* liste trois parties :

1. Le certificat proprement dit, qui contient beaucoup d'informations structurées sur plusieurs niveaux hiérarchiques
2. L'algorithme de signature du certificat, dans notre exemple, *PKCS #1 SHA-256 avec chiffrement RSA*
3. La signature du certificat, ici, une suite de 256 octets

Cette structure en trois parties est toujours respectée pour un certificat x509. A noter qu'Internet Explorer et Chrome affichent les mêmes informations mais sans faire ressortir la structure trois parties.

## 2.2. Structure d'un certificat x509

Où la structure d'un certificat est-elle définie, et quelle est cette définition ?

Une recherche sur un moteur de recherche avec les mots-clés *RFC* et *x509* produit l'URL suivante dans les premières réponses :

<https://tools.ietf.org/html/rfc5280>

Et effectivement la [1: Nous utiliserons le féminin dans ce document. RFC étant un acronyme anglais, il n'y a pas d'argument définitif pour l'emploi du masculin ou du féminin.] **RFC 5280** définit le format x509 version 3.

Affichons-la. Dans la section 4.1 se trouve la définition suivante.

...

#### 4.1. Basic Certificate Fields

The X.509 v3 certificate basic syntax is as follows. For signature calculation, the data that is to be signed is encoded using the ASN.1 distinguished encoding rules (DER) [X.690]. ASN.1 DER encoding is a tag, length, value encoding system for each element.

```
Certificate ::= SEQUENCE {  
    tbsCertificate      TBSCertificate,  
    signatureAlgorithm  AlgorithmIdentifier,  
    signatureValue      BIT STRING }
```

...

La suite définit les différents éléments du certificat, à savoir ce que sont les structures *TBSCertificate* et *AlgorithmIdentifier*.

### Grammaire, ASN.1 et DER

La structure du certificat est décrite par une *grammaire*, d'après les règles de syntaxe **ASN.1** (Abstrat Syntax Notation number 1) [1].

Les différents encodages possibles du standard *ASN.1* sont eux-mêmes des standards et le **X.690** est l'un d'entre eux [2].

La RFC précise que la partie du certificat à signer doit être encodée selon le standard *Distinguished Encoding Rules* ou **DER**. Entre autres encodages, le document X.690 définit le DER.

Nous verrons plus loin l'encodage DER. Si vous souhaitez le découvrir, plutôt que d'examiner directement le document X.690, je vous recommande de commencer par l'article Wikipédia [3].

En ASN.1 le mot-clé *SEQUENCE* sans autre précision indique que la valeur est constituée d'une suite de valeurs elles-mêmes spécifiées en ASN.1. La valeur *Certificate* contient donc, à la suite :

1. La valeur *tbsCertificate*, soit le certificat à signer (**to be signed Certificate**)
2. La valeur *signatureAlgorithm*, soit l'identification de l'algorithme de signature
3. La valeur *signatureValue*, soit la signature elle-même

## 2.3. Hiérarchie des certificats

Dans la partie *tbsCertificate* de *letsencrypt.org*, intéressons-nous à deux éléments en particulier, l'*émetteur* du certificat et le *sujet* du certificat.

- Le *sujet* du certificat a pour **CN (Common Name)** *letsencrypt.org* et c'est le dernier nom qui est affiché dans la hiérarchie des certificats (partie supérieure de la fenêtre).

- L'émetteur du certificat a pour CN *TrustID Server CA A52* et on peut voir ce nom au-dessus de *letsencrypt.org* dans la hiérarchie.

L'émetteur et le sujet ont également le pays (C) et l'organisation (O) définis dans leur nom, ainsi que d'autres éléments. Le "nom simple" ou "nom court" du certificat est son CN. Le standard x509 ne définit pas cette notion de "nom simple" ou "nom court", nous l'employons ici pour préciser que dans la pratique, le CN est le véritable nom du certificat, les autres éléments donnant des informations annexes. Parfois, seul le CN est affiché pour désigner un certificat.

Cela dit, le nom du certificat (au sens du standard x509) est constitué de *la totalité des éléments qui le composent*, et non pas seulement du CN.

Le lien hiérarchique est toujours établi entre un émetteur et un sujet. L'émetteur est celui qui signe le certificat, le sujet est celui qui est signé. Voir figures 7 et 8.

Un certificat peut être à la fois émetteur (d'autres certificats sont signés par lui) et sujet (il est lui-même signé par un autre certificat), et cette chaîne forme une structure hiérarchique. Dans notre exemple, on voit que le certificat *TrustID Server CA A52* est lui-même signé par *IdenTrust Commercial Root CA 1*.

#### NOTE

Dans la pratique la situation se complique souvent avec des certificats croisés. Quoiqu'il en soit, la structure de base des liens qui relient les certificats est hiérarchique.

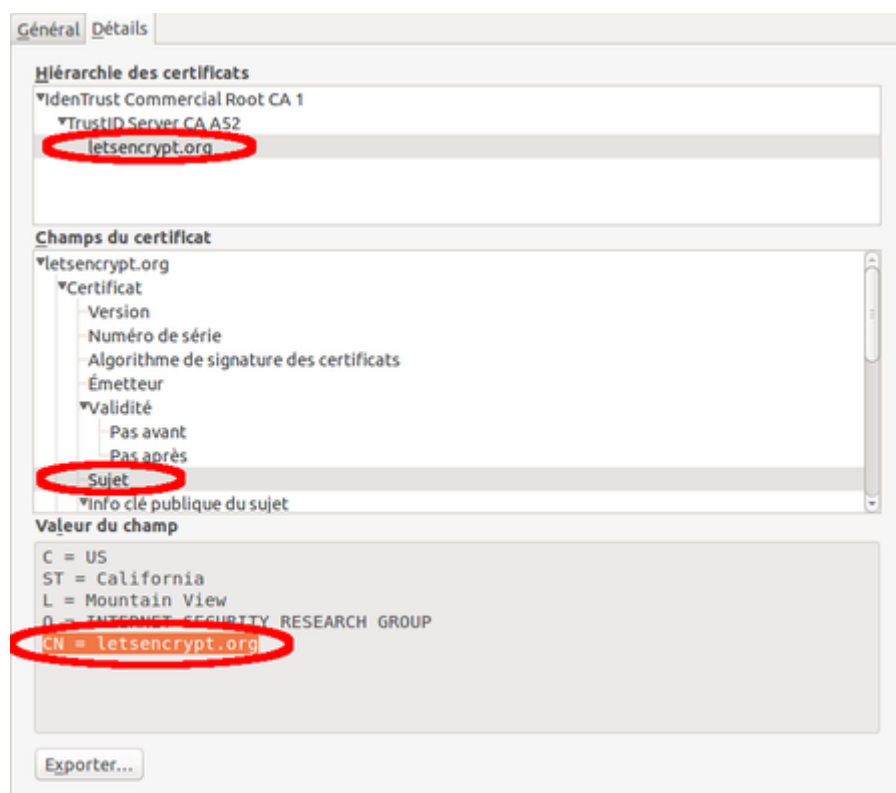


Fig. 7 : Sujet du certificat



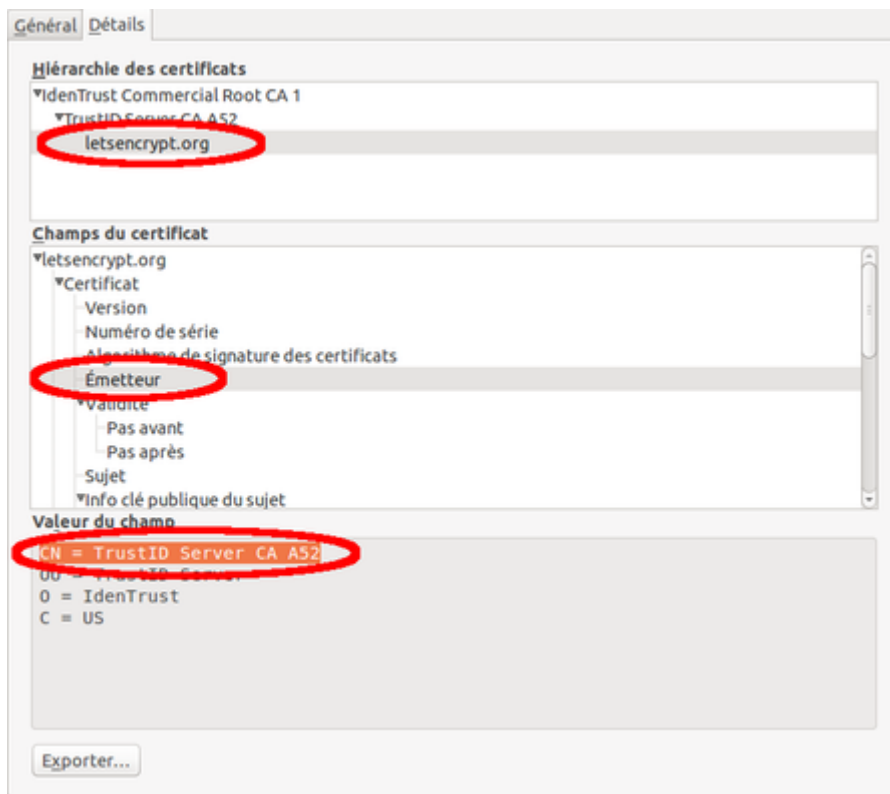


Fig. 8 : Émetteur du certificat

Pour signer, l'émetteur utilise sa clé privée. La vérification de la signature est faite avec sa clé publique. Ainsi pour vérifier l'authenticité du certificat [https de letsencrypt.org](https://letsencrypt.org), nous aurons besoin de la clé publique de son émetteur, *TrustID Server CA A52*.

Ce principe est toujours respecté avec les certificats x509, que ce soit avec RSA ou d'autres mécanismes à clé publique / clé privée.

Nous allons maintenant passer à la vérification de la signature RSA.

## 3. Vérification de signature RSA

### 3.1. La signature RSA

#### Notations

- La valeur à signer (ou si l'on préfère, le *bloc de données* à signer) est **tbsCertificate**, soit le certificat sans les informations de signature. Dans la structure en trois parties de notre certificat x509, c'est la première.
- L'entité qui signe le certificat (l'émetteur du certificat) a pour clé RSA **(n, e)** (**n** est le modulo, **e** est l'exposant) et **d**. Le couple **(n, e)** est la clé publique, **d** est la clé privée.

La page Wikipédia consacrée au système RSA [4] explique le lien entre  $(n, e)$  et  $d$ , et nous indique le calcul à effectuer pour *chiffrer*. Pour *signer*, le calcul inverse le rôle de l'exposant privé et public, et pour *vérifier* la signature, le rôle des exposants privé et public est encore inversé (par rapport à la signature).

Dans notre exemple le tbsCertificate est celui de *letsencrypt.org*, tandis que la clé RSA (clé publique  $(n, e)$  et clé privée  $d$ ) est celle de *TrustID Server CA A52*.

#### 3.1.1. Calcul de la signature

1. L'émetteur calcule le hash (noté  $M$ ) de la valeur *tbsCertificate* du sujet, soit  $M = \text{hash}(\text{"tbsCertificate"})$
2. Il calcule la signature [2: On simplifie ici pour se concentrer sur les étapes importantes. En fait, le hash  $M$  calculé n'est pas signé tel quel, il subit auparavant quelques transformations comme nous le verrons plus loin.] [3: Ce calcul est le même que pour déchiffrer un message destiné au propriétaire de la clé publique (qui est ici l'émetteur du certificat). Signer en RSA revient à chiffrer en inversant le rôle de l'exposant public et privé.] (notée  $S$ ) avec la formule  $S = M^d \bmod n$

#### 3.1.2. Vérification de la signature

1. Le vérificateur calcule  $M = S^e \bmod n$
2. Il calcule  $M' = \text{hash}(\text{"tbsCertificate"})$
3. Si on a l'égalité  $M = M'$ , la signature est vérifiée

## 3.2. Vérification du certificat de letsencrypt.org

### 3.2.1. Choix d'un programme de calcul

Nous avons besoin d'une "calculatrice" qui calcule sur des nombres entiers arbitrairement grands, sans perte de précision. Dans la suite de ce document, c'est *bc* qui sera utilisé [5].

- Linux : *bc* est disponible par défaut sur la plupart des distributions.
- Windows : les binaires sont accessibles à l'URL [6].

#### NOTE

##### Le choix de *bc*

- *bc* est installé par défaut sur la plupart des distributions Linux, et facile et rapide à installer sous Windows.
- *bc* contient peu de fonctions mathématiques intégrées mais sur Internet on trouve de nombreux scripts qui permettent de l'enrichir considérablement.
  - Dans *bc* la variable *scale* définit le nombre de décimales des nombres manipulés. Comme nous ne ferons que des calculs sur des entiers, nous laisserons *scale* à zéro (pas de partie décimale). Zéro est la valeur par défaut de *scale* au lancement de *bc* [5: Lorsque *bc* est lancé avec l'option *-l*, des fonctions mathématiques sont chargées au démarrage et *scale* vaut 20. Nous ne nous servirons pas de cette option dans ce document et *scale* sera toujours égal à zéro.] .

#### NOTE

##### Alternatives à *bc*

- **sagemath**, logiciel mathématique en licence GPL.
- **python**, langage de programmation en licence GPL, calcule par défaut sur des entiers de taille arbitrairement grande et convient donc aux calculs que nous allons faire.
- Logiciels mathématiques propriétaires bien connus.

Table 1. Comparaison entre *bc* et *python*

<b>bc</b>	<b>python</b>
Saisie d'un entier en hexadécimal	
Exécuter au préalable <code>ibase = 2 * 8</code> [6: <code>2 * 8</code> produit toujours 16 (décimal). Si 16 est lu alors qu' <i>ibase</i> vaut déjà 16, le résultat sera lu en hexadécimal et vaudra 22.] Exemple : <code>ibase = 2 * 8 var = ABEF0E0</code> (Attention les caractères hexadécimaux doivent être en majuscule.)	Saisir l'entier précédé de 0x Exemple : <code>&gt;&gt;&gt; var = 0xabef0e0</code> [7: <i>python</i> peut lire les nombres hexadécimaux indifféremment en majuscule et minuscule. <i>bc</i> exige des majuscules.]
Affichage d'entier en hexadécimal	

bc	python
<p>Exécuter au préalable <code>ibase = 2 * 8</code> [6: 2 * 8 produit toujours 16 (décimal). Si 16 est lu alors qu'ibase vaut déjà 16, le résultat sera lu en hexadécimal et vaudra 22.] Exemple : <code>ibase = 2 * 8 2 ^ (2 ^ 4) + 1</code> [8: Sous Windows, <i>bc</i> a des difficultés à lire le caractère <i>circonflexe</i> au clavier (en-dessous de la touche 9). Pour contourner ce problème, il est possible d'utiliser (sur un clavier français) l'accent circonflexe à droite de la touche P, en appuyant dessus à deux reprises. Cela affiche deux <i>circonflexes</i> et vous devez alors en supprimer un.] 10001</p>	<p>Interpoler avec %x Exemple : <code>&gt;&gt;&gt; '%x' % (2 ** (2 ** 4) + 1) '10001'</code></p>

### 3.2.2. Enregistrement de la signature sous forme d'entier

1. Depuis le navigateur, afficher la signature du certificat de *letsencrypt.org*.
2. Sélectionner la signature et la copier-coller dans un éditeur de texte.
3. Supprimer les caractères surnuméraires (enlever les ':' et les sauts de ligne).
4. Passer les caractères hexadécimaux en majuscule [9: Si vous utilisez *python*, il est inutile de passer les caractères hexadécimaux en majuscule.].
5. Ajouter `s=` devant le nombre, et ajouter une première ligne `ibase = 2 * 8`.
6. Enregistrer dans **val.b**.

Voir figures 9 et 10.

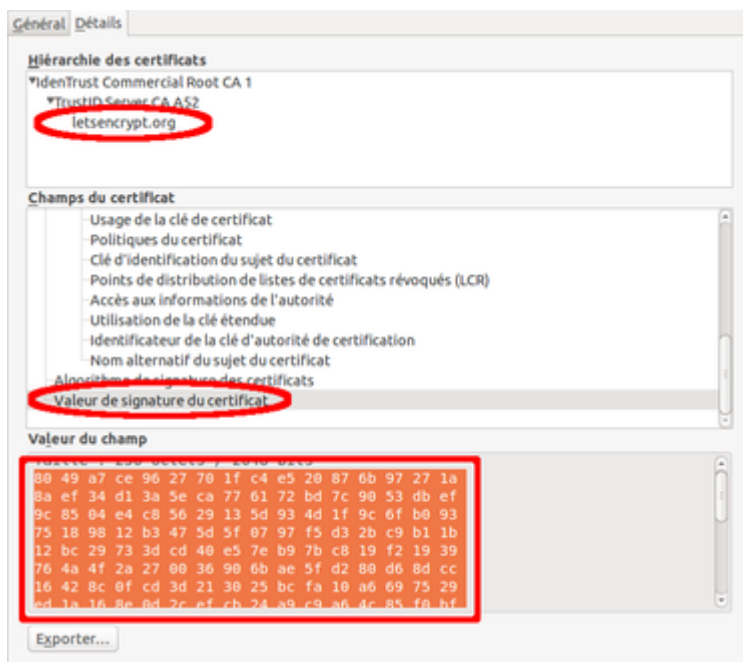


Fig. 9 : Signature dans le navigateur

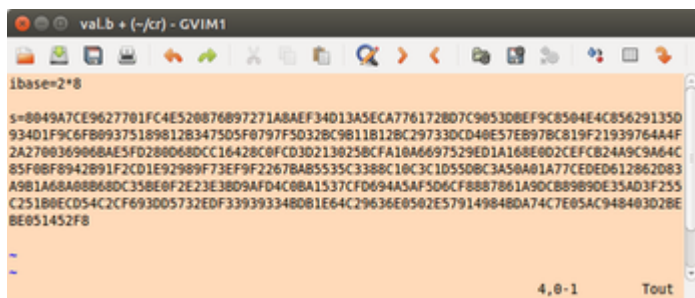


Fig. 10 : Fichier val.b

L'instruction `ibase = 2 * 8` ordonne à `bc` de lire les nombres en hexadécimal. `ibase = 16` fonctionne aussi, à condition qu'`ibase` soit égal à 10 (valeur par défaut) au moment d'exécuter `ibase = 16`. Si `ibase` est déjà égal à 16 et que l'on exécute `ibase = 16`, une erreur se produit car `bc` lit 16 en hexadécimal (soit 22) et cette valeur est interdite.

#### NOTE

1. Avec `bc` il est impératif de passer les nombres hexadécimaux en majuscule.
2. `bc` n'accepte pas les noms de variable qui contiennent des majuscules.

Le navigateur affiche l'exposant de la clé RSA (le nombre noté *e* tout à l'heure) sous forme décimale, alors que dans le script `bc` nous l'entrons en hexadécimal. Pour convertir un nombre décimal en hexadécimal, exécuter dans un terminal :

#### Linux

```
$ echo "obase=16; 65537" | bc
10001
```

#### Windows

```
$ echo obase=16; 65537 | bc.exe
10001
```

## Passer du texte en majuscule

### Linux

Sous Linux, on peut utiliser la ligne de commande, par exemple (nombreuses autres solutions) :

```
$ tr '[:lower:]' '[:upper:]' < fichier_entrée > fichier_sortie
```

Un éditeur suffisamment avancé comme *vim* ou *emacs* le permet aussi.

### Windows

Windows ne dispose pas par défaut d'outil pour passer du texte en majuscule. Il faut installer *GNUWin32* ou *cygwin*.

L'éditeur de texte (natif) adéquat pour ce type de transformation est **notepad++**, disponible à cette URL [\[7\]](#).

**NOTE** *vim* et *emacs* sont disponibles sous Windows.

### 3.2.3. Enregistrement de la clé publique sous forme d'entier

Depuis le navigateur :

1. Sélectionner le certificat de *TrustID Server CA A52* et afficher sa clé publique.
2. Sélectionner la valeur de la clé publique et la copier-coller dans **val.b**. Il faut le faire en deux fois, une fois pour le modulo de 256 octets (variable *n*) et une fois pour l'exposant (variable *e*).

Ne pas oublier de passer les caractères hexadécimaux en majuscule.

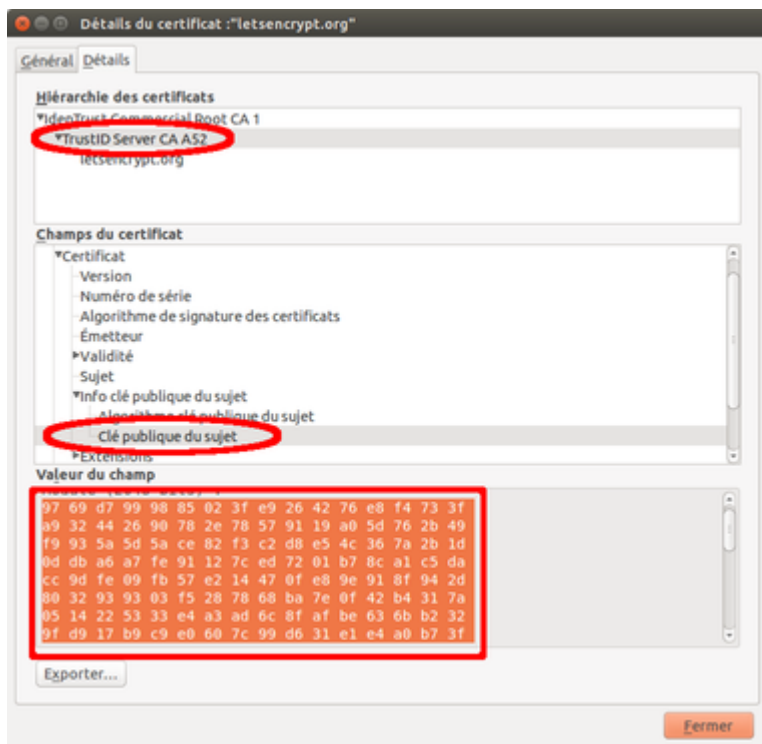


Fig. 11 : Clé publique dans le navigateur

A l'arrivée, **val.b** contient trois variables, s, n et e.

**val.b**

ibase=2\*8

s=8049A7CE9627701FC4E520876B97271A8AEF34D13A5ECA776172BD7C9053DBEF9C8504E4C85629135D93  
4D1F9C6FB09375189812B3475D5F0797F5D32BC9B11B12BC29733DCD40E57EB97BC819F21939764A4F2A27  
0036906BAE5FD280D68DCC16428C0FCD3D213025BCFA10A6697529ED1A168E0D2CEFCB24A9C9A64C85F0BF  
8942B91F2CD1E92989F73EF9F2267BAB5535C3388C10C3C1D55DBC3A50A01A77CEDED612862D83A9B1A68A  
08B68DC35BE0F2E23E3BD9AFD4C0BA1537CFD694A5AF5D6CF8887861A9DCB89B9DE35AD3F255C251B0ECD5  
4C2CF693DD5732EDF33939334BDB1E64C29636E0502E57914984BDA74C7E05AC948403D2BEBE051452F8

n=9769D7999885023FE9264276E8F4733FA932442690782E78579119A05D762B49F9935A5D5ACE82F3C2D8  
E54C367A2B1D0DDBA6A7FE91127CED7201B78CA1C5DACC9DFE09FB57E214470FE89E918F942D8032939303  
F5287868BA7E0F42B4317A0514225333E4A3AD6C8FAFBE636BB2329FD917B9C9E0607C99D631E1E4A0B73F  
AFB232AC7E8C9CDC02EBE1BC1F149CBC91F7B2FB42F3E1202BCBBF8FF3B37063FAF7752802ABC5D4B0EDEA  
257F87CD371496833C40021BA09E19477FF3B0CCC52560B83512F151EB17DCFC5BA5D99BEF404CD77771E9  
FB458B7EF2E369B042661746903ACD463DF1B0096FDCFFEE3361CAFCC72E3CED5E0AD1BF221269804B23

e=10001

### 3.2.4. La fonction powmod

Les amateurs de *python* ont encore un avantage à ce stade. L'équivalent de la fonction *powmod* y est disponible sous forme d'un troisième paramètre (facultatif) à la fonction *pow*.

Pour ceux qui utilisent bc comme moi, il faut écrire la fonction.

La fonction *powmod* met en oeuvre l'algorithme d'exponentiation rapide, décrit à cette URL [8]. En fait nous sommes dans un contexte modulaire et d'après Wikipédia le nom exact de l'algorithme est *exponentiation modulaire*. Un article y est consacrée [9]. Les deux algorithmes font appel au même principe, mais le second exploite le contexte modulaire pour que les nombres manipulés n'atteignent pas une taille démesurée. C'est le second algorithme (*exponentiation modulaire*) dont nous avons besoin pour la suite.

Sur Internet, on peut trouver la fonction *powmod* dans de nombreux scripts *bc* à télécharger. À noter qu'elle porte parfois d'autres noms, *mpower* par exemple.

*powmod.b*

```
define powmod(a, b, c) {
    auto p, r
    p = a
    r = 1
    while (b > 0) {
        if (b % 2) r = (r * p) % c
        p = (p * p) % c
        b /= 2
    }
    return r
}
```

### 3.2.5. Calcul de $M$

Nous voilà prêts pour calculer  $M$ .

1. Lancer la commande suivante [10: Par défaut, *bc* ajoute un anti-slash après le 68e caractère et passe à la ligne suivante. On peut modifier ce comportement avec la variable d'environnement *BC\_LINE\_LENGTH*. Quand cette variable est égale à zéro, les nombres ne sont pas coupés.] :

```
$ BC_LINE_LENGTH=0 bc powmod.b val.b
```

2. Dans le shell *bc*, exécuter [6:  $2 * 8$  produit toujours 16 (décimal). Si 16 est lu alors qu'*ibase* vaut déjà 16, le résultat sera lu en hexadécimal et vaudra 22.] (pour que les nombres soient affichés en hexadécimal)

```
obase=2*8
```

3. Toujours dans le shell *bc*, exécuter

```
powmod(s, e, n)
```

Rappelons que c'est  $S$  (variable *s* dans *val.b*) qui doit être élevé à la puissance  $e$ , modulo  $n$ . Lors de la définition d'*obase*, *ibase* vaut 16 (résultat du script *val.b* chargé au démarrage) donc *obase*=16 ne



fonctionnerait pas.

```
BC_LINE_LENGTH=0 bc powmod.b val.b
sebastien@maison-pclin in ~/cr
$ BC_LINE_LENGTH=0 bc powmod.b val.b
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
obase=2*8
powmod(s, e, n)
1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF003031300D0609608648016503040201050004208364DA78F1FD8DCC6812E568268BF2DAF87
91BE383109745388879C496A8C3DD
```

Fig. 12 : Calcul pour vérifier la signature RSA

#### IMPORTANT

Au démarrage de bc, ibase est égal à 16 du fait du chargement de *val.b*. Si vous exécutez *ibase = 16*, *ibase* sera égal à... 22 ! C'est logique, puisque 16 est saisi en hexadécimal. Le plus simple est d'utiliser *2 \* 8* comme indiqué. Vous pouvez aussi exécuter *ibase = 10* (*ibase* étant égal à 16), mais c'est moins clair, et vous devez être certain qu'*ibase* vaut bien 16 au moment d'exécuter *ibase = 10*, pour passer l'affichage des nombres en hexadécimal (*2 \* 8* fonctionne toujours quelle que soit la valeur d'*ibase*, y compris pour les valeurs inférieures ou égales à 7).

Le résultat (figure 12) avec tous ces *F* prouve avec une quasi certitude que le calcul s'est bien passé. Les *F* correspondent au 'padding' standard effectué pour une signature RSA, la valeur qui suit (à partir de **303130**) est le hash de *tbsCertificate* "emballé".

"Emballé", c'est-à-dire ? La valeur est spécifiée en ASN.1 et codée en DER, et elle contient d'autres informations que le seul hash de *tbsCertificate*.

C'est ce que nous allons voir dans le chapitre suivant.

### 3.2.6. Analyse de *M*

Nous allons procéder en trois étapes :

1. Enregistrement du contenu hexadécimal
2. Conversion du contenu hexadécimal en binaire
3. Examen du contenu binaire avec la commande *openssl asn1parse*

#### 1 Enregistrement du contenu hexadécimal

Faisons un copier-coller de *M* (en hexadécimal) à partir de l'octet qui suit l'octet nul, et enregistrons le résultat dans le fichier *m.hex*.

```
sebastien@maison-pclin: ~/cr
└─sebastien@maison-pclin in ~/cr
$ cat m.hex
3031300D0609608648016503040201050004208364DA78F1FD8DCC6812E568268BF2DAF8791BE38310974
5388879C496A8C3DD
└─sebastien@maison-pclin in ~/cr
$
```

Fig. 13 : *m.hex*

Contenu de *m.hex*

```
3031300D0609608648016503040201050004208364DA78F1FD8DCC6812E568268BF2DAF8791BE383109745
388879C496A8C3DD
```

## 2 Conversion du contenu hexadécimal en binaire

Maintenant nous allons convertir *m.hex* en binaire, puisque le contenu actuel est le *codage des octets en hexadécimal* de la signature, ce n'est pas la signature elle-même.

*Linux*

Exécuter la commande

```
$ xxd -r -p m.hex > m.der
```

Le plus simple est d'utiliser *notepad++* et d'enregistrer le fichier transformé avec le nom *m.der*.

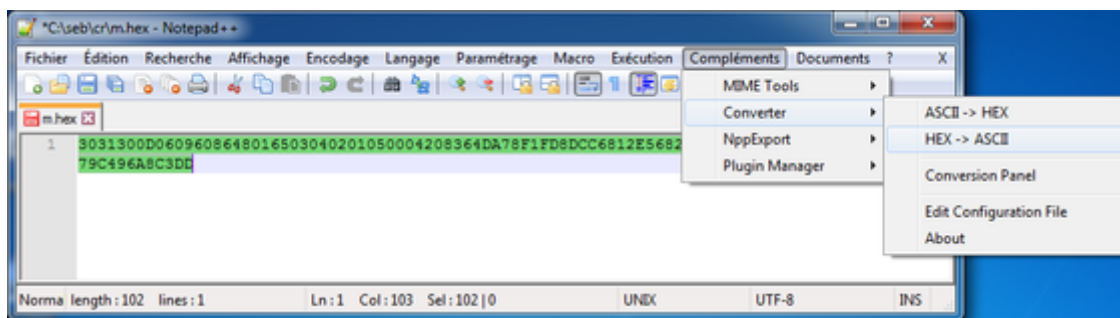


Fig. 14 : Avant la conversion hexadécimal -> binaire

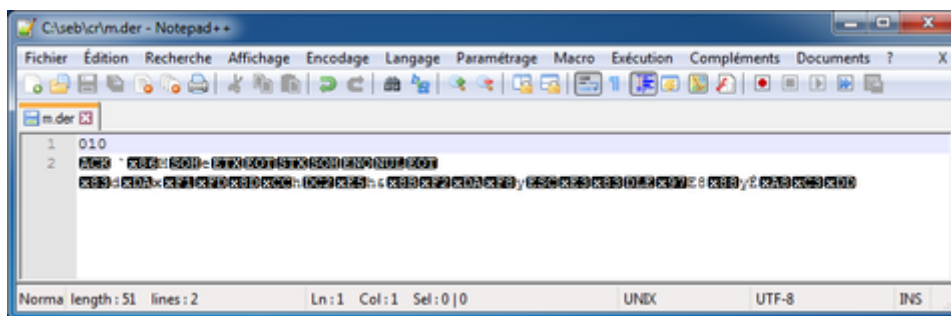


Fig. 15 : Après la conversion hexadécimal -> binaire

#### ATTENTION

Ne pas laisser un saut de ligne à la fin du fichier qui s'ajouterait aux données binaires de la signature.

Le fichier *m.der* contient les octets que l'on avait dans la signature, ce que l'on peut vérifier facilement...

- ... sous Linux avec la commande *hd* (on peut aussi utiliser *hexdump* ou *xxd*).
- ... sous Windows avec *notepad++*, en reconvertissant en hexadécimal et en vérifiant que l'on retombe sur ses pieds.

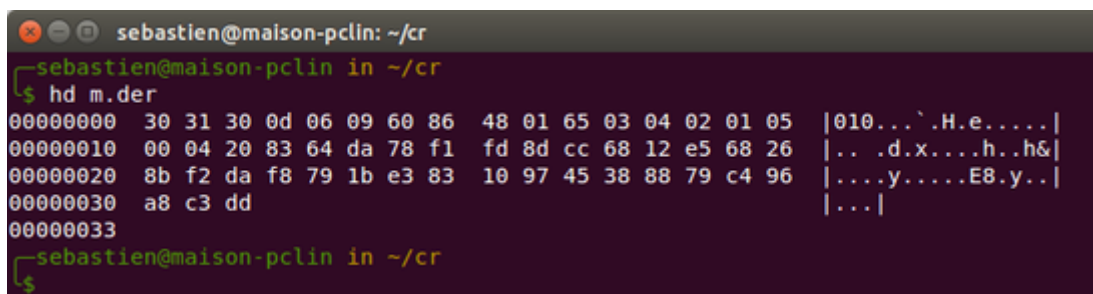


Fig. 16 : Contenu du fichier binaire *m.der*

### 3 Examen du contenu DER du fichier *m.der*

## openssl

Pour afficher le contenu binaire (qui se trouve être encodé en DER) nous allons utiliser l'exécutable en ligne de commande fourni avec la librairie openssl. Cet outil s'appelle *openssl*.

Par la suite nous utiliserons également *dder* et *pkfile*, deux utilitaires créés par l'auteur de cet article pour afficher et extraire plus facilement ce type de données.

*dder* est disponible à l'URL [\[10\]](#).

*pkfile* est disponible à l'URL [\[11\]](#).

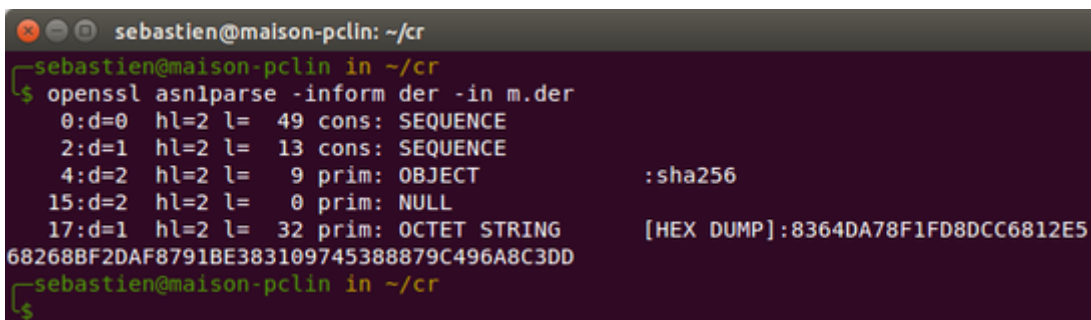
- Sous Linux ce programme est disponible par défaut.
- Sous Windows il faut trouver un binaire à télécharger. Le plus simple est de télécharger la version GNUWin32, disponible à cette URL [\[12\]](#).

La version proposée au téléchargement (en février 2016) date de 2008. Des binaires plus récents sont disponibles sur Internet. Pour faire les manipulations décrites dans ce document, la version de GNUWin32 est suffisante.

- À noter que Sous Windows les environnements "de taille importante" (*cygwin*, *perl*, ...) sont souvent installés avec leur librairie openssl, qui contient l'outil en ligne de commande *openssl*.

Exécuter la commande

```
$ openssl asn1parse -inform der -in m.der
```



```
sebastien@maison-pclin: ~/cr
sebastien@maison-pclin in ~/cr
$ openssl asn1parse -inform der -in m.der
0:d=0 hl=2 l= 49 cons: SEQUENCE
2:d=1 hl=2 l= 13 cons: SEQUENCE
4:d=2 hl=2 l= 9 prim: OBJECT          :sha256
15:d=2 hl=2 l= 0 prim: NULL
17:d=1 hl=2 l= 32 prim: OCTET STRING  [HEX DUMP]:8364DA78F1FD8DCC6812E5
68268BF2DAF8791BE383109745388879C496A8C3DD
sebastien@maison-pclin in ~/cr
$
```

Fig. 17

Nous verrons dans le chapitre suivant la syntaxe ASN.1 plus en détail.

La commande *openssl* (figure 17) nous donne deux informations :

- L'algorithme de hash est SHA-256, ce que l'on savait déjà d'après le contenu de *signatureAlgorithm* (deuxième partie de la structure en trois parties du certificat).
- Nous voyons la valeur du hash (le bloc *prim: OCTET STRING*) sous forme hexadécimale. Sa longueur correspond bien au SHA-256 (256 bits de longueur soit 32 octets).

Nous savons désormais que la signature SHA-256 de la valeur `tbsCertificate` du certificat de *letsencrypt.org* devrait être :

```
8364DA78F1FD8DCC6812E568268BF2DAF8791BE383109745388879C496A8C3DD
```

C'est ce que nous allons vérifier en calculant maintenant  $M'$ .

### 3.2.7. Calcul de $M'$

Nous allons calculer  $M'$  en deux étapes :

1. Extraction de `tbsCertificate` du certificat *letsencrypt.org*.
2. Calcul du hash SHA-256 de `tbsCertificate`.

Mais avant d'extraire `tbsCertificate`, nous devons comprendre comment le certificat est structuré et codé.

#### ASN.1 et DER

Au début de ce document, nous avons observé que la section 4.1 de la RFC 5280, qui définit la structure des certificats x509 v3, contient cette définition :

```
Certificate ::= SEQUENCE {  
    tbsCertificate      TBSCertificate,  
    signatureAlgorithm  AlgorithmIdentifier,  
    signatureValue      BIT STRING }
```

Un certificat x509 est défini selon la syntaxe ASN.1. L'encodage peut être BER [11: BER signifie **B**asic **E**ncoding **R**ules] , CER [12: CER signifie **C**anonical **E**ncoding **R**ules] ou DER [13: DER signifie **D**istinguished **E**ncoding **R**ules] .

Ces trois standards sont très proches, l'intérêt de DER étant son unicité : une structure de données spécifiée en ASN.1 ne peut être encodée en DER que d'une manière, et une seule. BER et CER permettent certaines variations dans la manière d'encoder.

Pour ne pas alourdir la rédaction, nous parlerons toujours d'encodage DER ou simplement DER, même lorsque les données manipulées pourraient ne pas être DER [14: La RFC 5280 indique que les *données signées* d'un certificat (la partie `tbsCertificate`) doivent être encodées en DER, mais ne donne pas d'indication pour le reste.] .

ASN.1 structure chaque valeur (*data value* dans le document X.690) selon la typologie **T - L - V** ou **Tag - Length - Value**. La forme "longueur indéfinie" apporte un quatrième élément, "end-of-contents" (marqueur de fin de valeur), que nous ignorerons car DER ne permet pas cette forme.

#### [T]ag

Définit toutes les caractéristiques de la valeur, notamment son *type*, par exemple une date (*UTCTime*), une séquence (*SEQUENCE* ou *SEQUENCE OF*), un entier (*INTEGER*), et bien d'autres.

Il définit également si l'encodage de la valeur est *primitive* ou *constructed*.

### Encodage primitive

La donnée ne contient pas de sous-structure

### Encodage constructed

La donnée est elle-même une structure qui suit la typologie **T-L-V**, et ici, cette structure est elle-même définie en ASN.1 [15: Certains types de données, notamment les *\*string* (OCTET STRING, UTF8String, etc.), peuvent être *primitive* ou *constructed*, ce qui signifie que leur encodage peut suivre une structure hiérarchique "propre à l'encodage", non explicitée dans la définition de la valeur en ASN.1.].

### Structure hiérarchique

La différence *primitive* - *constructed* est à la base de la structure hiérarchique d'une spécification ASN.1, les éléments *constructed* étant les branches, les éléments *primitive* étant les feuilles.

Dans la définition d'un *Certificate* ci-dessus, le type *SEQUENCE* est, par définition du type *SEQUENCE* en ASN.1, *constructed*. Cela signifie que la valeur *SEQUENCE* est elle-même une structure ASN.1, ce qu'on peut voir dans la liste des trois composants d'un certificat définie entre accolades.

### [L]ength

Donne la longueur de la valeur en octets.

### [V]alue

Est la donnée elle-même. Dans ce document nous employons le terme *valeur* (dans le document X.690 le terme est *data value*).

Affichons le contenu DER avec l'utilitaire en ligne de commande *dder* en exécutant la commande suivante :

```
$ dder -width 8 -recursive "| " m.der
```

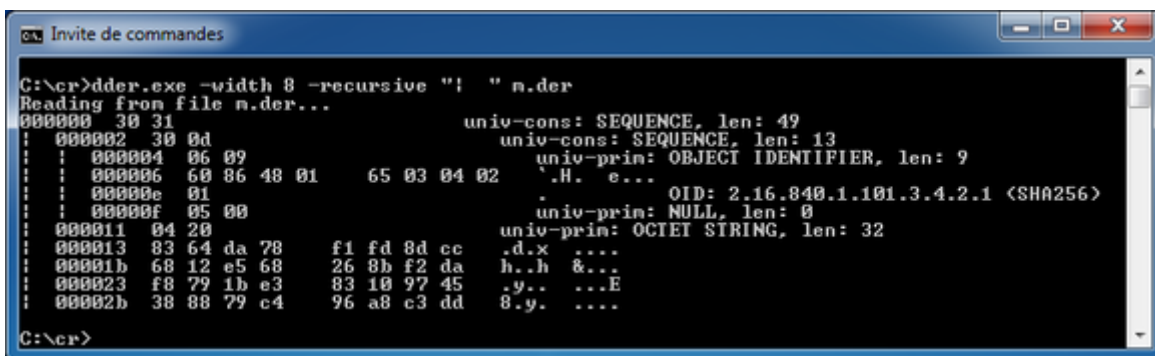


Fig. 18

La copie d'écran de la figure 18 a été faite sous Windows pour changer un peu.

La description de la valeur indique *-cons* ou *-prim*, pour *constructed* ou *primitive*. Chaque fois que le

tag indique une valeur *constructed* (ici, il s'agit à deux reprises du type *SEQUENCE*, qui est obligatoirement *constructed*), on descend d'un niveau dans la structure hiérarchique, que l'on a fait ressortir ici avec l'option *-recursive* "|".

## Extraction de `tbsCertificate` du certificat *letsencrypt.org*

### Étape 1 de l'extraction

Commençons par enregistrer le certificat depuis le navigateur, dans le fichier *letsencryptorg.der*.

*Firefox*

Afficher le certificat comme vu précédemment, puis afficher l'onglet *Détails* et cliquer sur le bouton *Exporter*. Ensuite sélectionner *Certificat X.509 (DER)*.

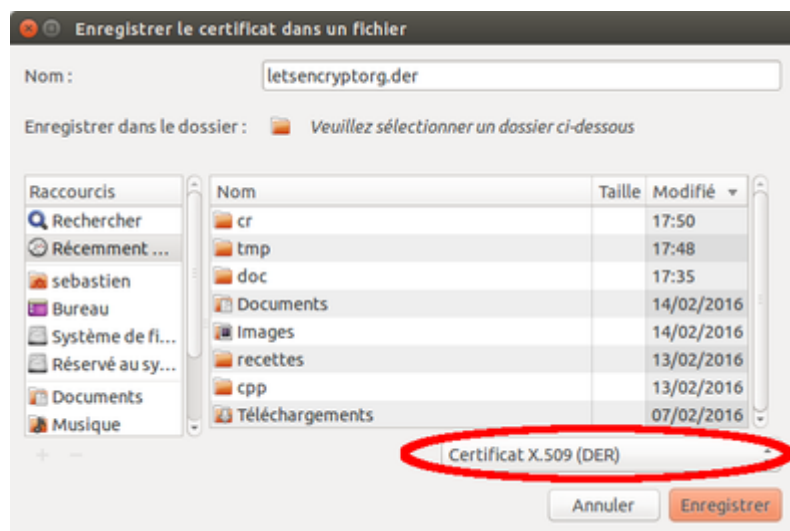


Fig. 19

Afficher le certificat, puis cliquer sur le bouton *Copier dans un fichier...*.... Ensuite sélectionner *X.509 binaire encodé DER*.

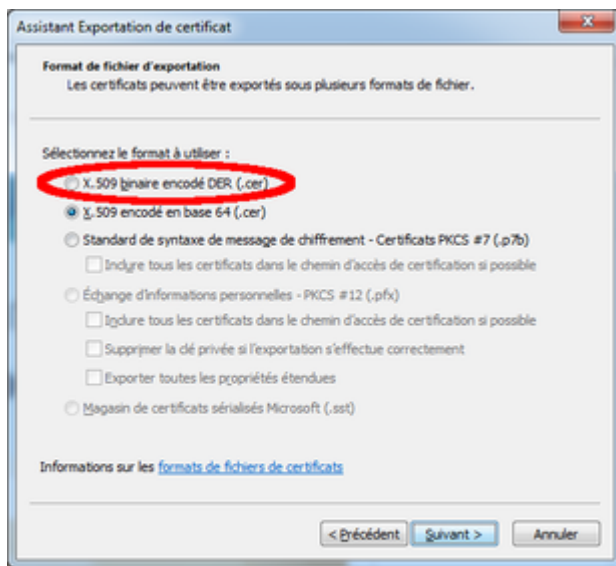


Fig. 20



## PEM versus DER

Avec les certificats x509, l'encodage PEM est une surcouche de DER qui consiste à :

- Encoder le contenu binaire (après chiffrement éventuel) en BASE64
- Ajouter au début et à la fin un texte standardisé qui définit et délimite la nature du contenu, ainsi que les informations de chiffrement s'il y a lieu
  - La ligne ajoutée au début commence par -----BEGIN
  - La ligne ajoutée à la fin commence par -----END

Si vous avez enregistré le certificat avec un encodage PEM, vous pouvez le convertir en DER avec *openssl*. Exemple pour *letsencrypt.org* (PEM) que l'on convertit en *letsencrypt.org* (DER) - comme on peut voir, l'encodage par défaut avec *openssl* est PEM.

```
$ openssl x509 -in letsencryptorg.cer -outform der -out  
letsencryptorg.der
```

### NOTE

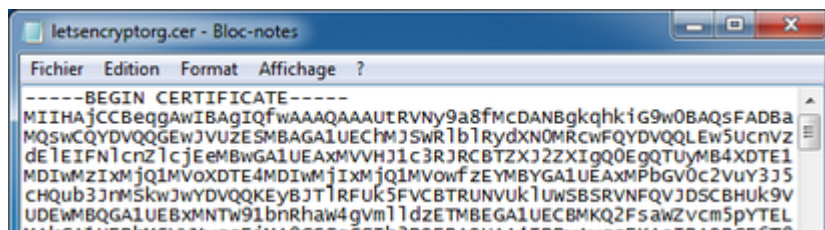


Fig. 21

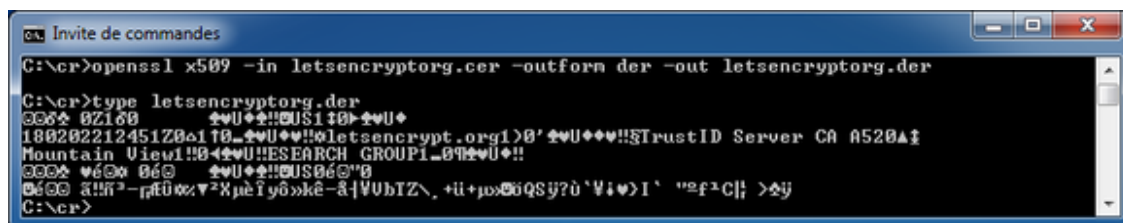


Fig. 22

Vous pouvez faire cette conversion "à la main" en enlevant la première et la dernière ligne et en faisant une conversion BASE64 → binaire avec des outils comme *base64* (Linux) ou *notepad++* (Windows).

## Étape 2 de l'extraction

Nous disposons maintenant d'un certificat enregistré en binaire (encodage DER) dans le fichier *letsencryptorg.der*, dont nous allons à présent extraire la partie *tbsCertificate*.

### NOTE

Les outils *dder* et *pkfile* (*pkfile* est présenté plus loin) peuvent lire indifféremment des fichiers codés en PEM ou en DER. Si le fichier est PEM, il est décodé à la volée.

Il y a deux solutions pour ce faire

1. À la main
2. À l'aide de l'utilitaire *pkfile*. *pkfile* est disponible à cette URL [\[11\]](#).

*Extraction de tbsCertificate : alternative 1 (à la main)*

Un éditeur de fichier binaire ferait l'affaire, mais la manipulation est plus claire si l'on affiche le contenu hexadécimal avec *dder*.

Exécuter la commande

```
$ dder -recursive "| " -hex letsencryptorg.der > d
```

*Encodage des fichiers affichés par dder*

*dder* lit indifféremment des fichiers PEM et DER. Si le fichier est PEM il décode (et décrypte si nécessaire) les données en mémoire avant d'afficher le contenu DER.

**NOTE**

Donc la commande aurait pu être (avec le fichier *letsencryptorg.cer* encodé en PEM) :

```
$ dder -recursive "| " -hex letsencryptorg.cer > d
```

Ouvrir le fichier *d* dans un éditeur de texte. Comme on a affiché le contenu avec un décalage à chaque niveau hiérarchique (option *-recursive "| "*), la structure en trois parties des certificats x509 ressort bien.

Supprimer les deux premières lignes et un certain nombre de lignes à la fin pour que seul demeure le contenu de *tbsCertificate*. Ci-dessous, le texte à conserver est le texte sélectionné (les lignes au milieu ont été supprimées pour condenser l'image).

```
C:\cr>dder -recursive "| " -hex letsencryptorg.cer > d
C:\cr>_
```

*Fig. 23*

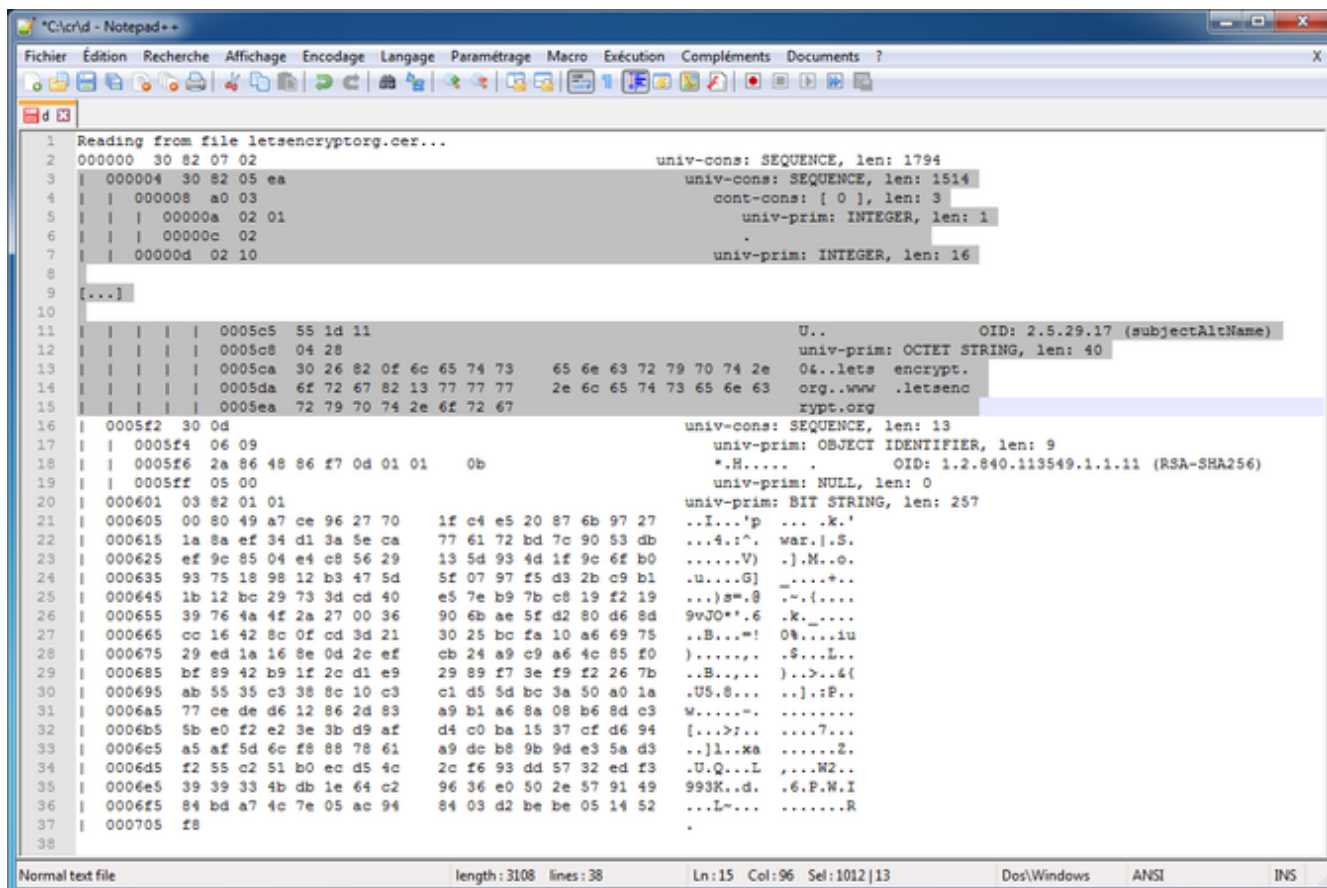


Fig. 24

Ensuite :

- Supprimer les "|" en début de ligne
- Supprimer l'offset (numéro sur six caractères)
- Supprimer le texte après les codes hexadécimaux (une fois supprimé les "|" et l'offset en début de ligne, ce sont tous les caractères au-delà de la 55e position qu'il faut supprimer).
- Supprimer tous les espaces

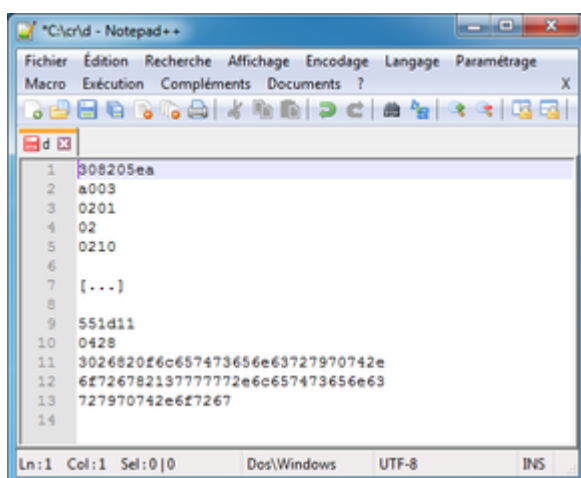


Fig. 25

On retrouve (figure 25) une sous-partie du fichier *letsencryptorg.der*, codée en hexadécimal, qui commence par **308205ea** et se termine par **2e6f7267**.

## NOTE

Cette manipulation peut être faite avec un programme d'édition de fichier binaire, en ne gardant que les données à partir de l'offset 4, d'une longueur de 1518 octets.

Il faut ensuite convertir le contenu "codé hexadécimal" en binaire comme cela a été vu précédemment (commande `xxd -r -p` sous Linux, avec `notepad++` sous Windows).

Au final on obtient le fichier `tbs.der` comme montré sur la figure 26.

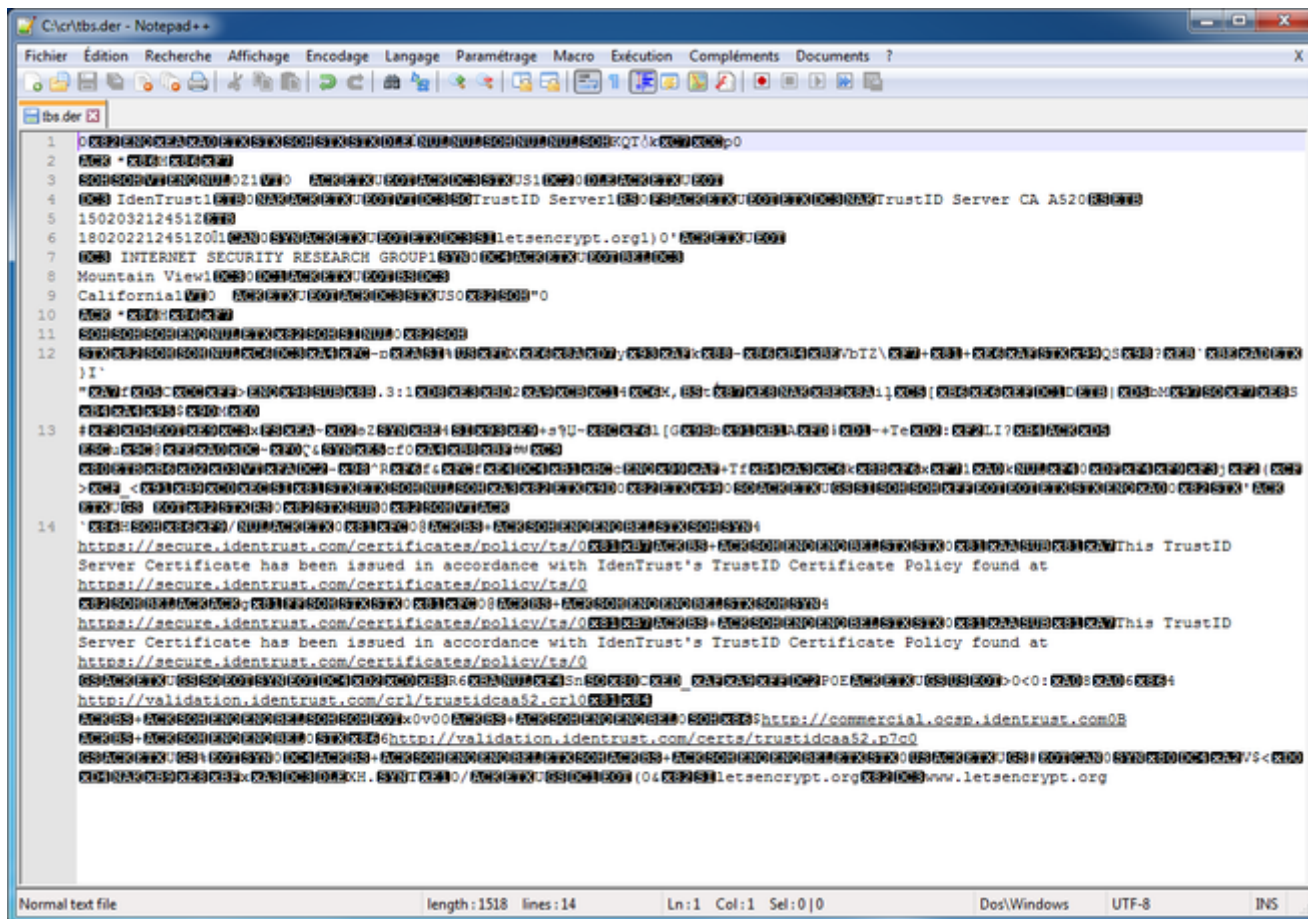


Fig. 26

Extraction de `tbsCertificate` : alternative 2 (à l'aide de `pkfile`)

Exécuter la commande

```
$ pkfile letsencryptorg.der -l 2
```

```

C:\cr>pkfile.exe letsencryptorg.cer -l 2
1      cons: SEQUENCE, len: 1798 <4+1794>
1.1    ┌─ cons: SEQUENCE, len: 1518 <4+1514>
1.2    │ ┌─ cons: SEQUENCE, len: 15 <2+13>
1.3    │ └─ prim: BIT STRING, len: 261 <4+1+256>
1.3.1  │ 8049A7CE 9627701F C4E52087 6B97271A
        │ 8AEF34D1 3A5ECA77 6172BD7C 9053DBEF
        │ 9C8504E4 C8562913 5D934D1F 9C6FB093
        │ 75189812 B3475D5F 0797F5D3 2BC9B11B
        │ 12BC2973 3DCD40E5 7EB97BC8 19F21939
        │ 764A4F2A 27003690 6BAE5FD2 80D68DCC
        │ 16428C0F CD3D2130 25BCFA10 A6697529
        │ ED1A168E 0D2CEFCB 24A9C9A6 4C85F0BF
        │ 8942B91F 2CD1E929 89F73EF9 F2267BAB
        │ 5535C338 8C10C3C1 D55DBC3A 50A01A77
        │ CEDED612 862D83A9 B1A68A08 B68DC35B
        │ E0F2E23E 3BD9AFD4 C0BA1537 CFD694A5
        │ AF5D6CF8 887861A9 DCB89B9D E35AD3F2
        │ 55C251B0 ECD54C2C F693DD57 32EDF339
        │ 39334BDB 1E64C296 36E0502E 57914984
        │ BDA74C7E 05AC9484 03D2BEBE 051452F8
C:\cr>

```

Fig. 27

Le résultat visible sur la figure 27 nous montre que le `tbsCertificate` correspond à la valeur `SEQUENCE` de 1518 octets (4 octets d'en-tête, 1514 octets de données). `pkfile` donne à cette valeur les coordonnées 1.1.

Nous enregistrons cette valeur en binaire dans le fichier `tbs2.der` en exécutant la commande

```
$ pkfile letsencryptorg.der -n 1.1 -x -o tbs2.der
```

```

C:\cr>pkfile.exe letsencryptorg.cer -n 1.1 -x -o tbs2.der
C:\cr>

```

Fig. 28

*Encodage des fichiers analysés par pkfile*

#### NOTE

`pkfile` lit indifféremment des fichiers PEM et DER. Si le fichier est PEM il décode (et déchiffre si nécessaire) les données en mémoire avant de traiter le contenu DER.

## Calcul du hash de `tbsCertificate`

### Linux

Le calcul des différents algorithmes de hash est disponible en ligne de commande. Pour SHA-256 le programme est `sha256sum`.

Exécuter

```
$ sha256sum tbs.der
```

```

sebastien@maison-pclin:~/cr$ sha256sum tbs.der
8364da78f1fd8dcc6812e568268bf2daf8791be383109745388879c496a8c3dd  tbs.der
sebastien@maison-pclin:~/cr$ sha256sum tbs2.der
8364da78f1fd8dcc6812e568268bf2daf8791be383109745388879c496a8c3dd  tbs2.der
sebastien@maison-pclin:~/cr$

```

Fig. 29

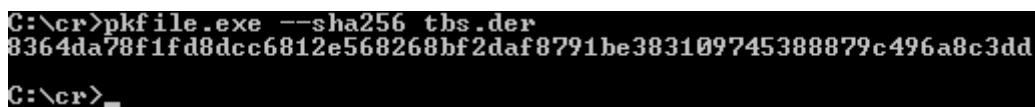


Les outils de calcul de hash ne sont pas disponibles par défaut. Voici quelques solutions (d'autres existent) :

- Utiliser *pkfile*, qui permet de calculer différents hash [17: *pkfile* peut le faire aussi sous Linux, mais dans ce système il est plus logique d'utiliser les programmes déjà installés. La fonctionnalité de calcul de hash a été ajoutée à *pkfile* pour simplifier les manipulations sous Windows.] .
- Installer les outils *GNUWin32*, disponibles à cette URL [12].
- 7-ZIP ajoute des menus contextuels dans l'explorateur pour calculer différents hashes, dont SHA-256. 7-ZIP est disponible à cette URL [13].

Exemple avec *pkfile*, exécuter

```
$ pkfile --sha256 tbs.der
```



```
C:\cr>pkfile.exe --sha256 tbs.der
8364da78f1fd8dcc6812e568268bf2daf8791be383109745388879c496a8c3dd
C:\cr>
```

Fig. 30



```
C:\cr>pkfile.exe --sha256 tbs2.der
8364da78f1fd8dcc6812e568268bf2daf8791be383109745388879c496a8c3dd
C:\cr>
```

Fig. 31

Les captures d'écran montrent également le calcul sur le fichier *tbs2.der* (créé avec *pkfile*), qui est bien identique à *tbs.der*, ce qui confirme au passage l'équivalence des deux termes de l'alternative pour extraire *tbsCertificate*.

**Le hash trouvé (*M'*) concorde avec la signature (*M*), ce qui valide la signature du certificat *letsencrypt.org***

### 3.3. Conclusion

Nous arrivons au terme des manipulations à effectuer pour vérifier la signature RSA d'un certificat x509. Les calculs étaient élémentaires, mais cela nous a permis de voir :

- Les différents formats et encodages et les outils pour passer de l'un à l'autre
  - PEM versus DER
  - BASE64 versus binaire
  - Hexadécimal versus binaire
  - ASN.1
  - La structure de certificat x509

- Les solutions pour calculer sur des entiers de grande taille
  - *python* (évoqué ici mais non détaillé) et *bc*
  - Les transformations éventuelles et précautions à prendre pour calculer en saisie et affichage hexadécimal
  - La création de la fonction *powmod* dans *bc*
- Outils divers
  - Calcul du hash d'un fichier
  - Extraction d'une valeur d'un fichier encodé en DER

La *signature* a été vérifiée mais le certificat lui-même n'a pas été validé en totalité. Il y a deux raisons à cela.

1. Une fois un certificat vérifié, il faut vérifier son parent dans la hiérarchie (le certificat qui l'a signé), et ainsi de suite, jusqu'à vérifier un certificat connu dans la base des certificats du navigateur (les certificats *racine*).
2. Le navigateur vérifie si les certificats rencontrés ont été révoqués à l'aide des Listes de Révocation des Certificats ou *CRL* (**C**ertificate **R**evocation **L**ist).

## 4. Bibliography

- [1] Présentation de l'ASN.1 : <http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx>
- [2] Document X.690 (format PDF) : <https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>
- [3] Article Wikipédia consacré au X.690 : <https://en.wikipedia.org/wiki/X.690>
- [4] Article Wikipédia consacré au RSA : [https://fr.wikipedia.org/wiki/Chiffrement\\_RSA](https://fr.wikipedia.org/wiki/Chiffrement_RSA)
- [5] Page d'accueil de *bc* : <https://www.gnu.org/software/bc/>
- [6] *bc* pour Windows : <http://gnuwin32.sourceforge.net/packages/bc.htm>  
  
*bc* a besoin du fichier *readline5.dll* pour s'exécuter. Vous pouvez le trouver à cette URL : <http://gnuwin32.sourceforge.net/packages/readline.htm>
- [7] Le site de notepad++ : <https://notepad-plus-plus.org/fr>
- [8] Article Wikipédia sur l'algorithme d'exponentiation rapide : [https://fr.wikipedia.org/wiki/Exponentiation\\_rapide](https://fr.wikipedia.org/wiki/Exponentiation_rapide)
- [9] Article Wikipédia sur l'exponentiation modulaire : [https://fr.wikipedia.org/wiki/Exponentiation\\_modulaire](https://fr.wikipedia.org/wiki/Exponentiation_modulaire)
- [10] Obtenir *dder* : <http://dder.sourceforge.net>
- [11] Obtenir *pkfile* : <http://pkfile.sourceforge.net>
- [12] GNUWin32 : <http://gnuwin32.sourceforge.net/packages/openssl.htm>
- [13] Télécharger 7-ZIP : <http://www.7-zip.org/download.html>