
Cryptographie pratique

Sébastien Millet <<milletseb@laposte.net>>

Revision 0.5

Revision History
February 29th, 2016

SM

Table of Contents

Introduction	1
Le format x509	2
Visualisation d'un certificat x509	2
Structure d'un certificat x509	4
Hiérarchie des certificats	5
Vérification de signature RSA	8
La signature RSA	8
Vérification du certificat de letsencrypt.org	8
Conclusion	25
Vérification de signature ECDSA	25
Cryptographie à courbes elliptiques	25
Bibliography	35

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*)¹

Au fil du document nous ferons appel aux outils suivants :

- *openssl* pour travailler sur les certificats x509 en ligne de commande
- *pkfile* pour extraire la partie signée d'un certificat et *dder* pour afficher certains contenus binaires
- *python* ou *bc* pour faire des calculs avec des nombres entiers de grande taille
- Conversion entre encodage *PEM* et encodage *binnaire* (Linux : *base64*, Windows : *notepad++*)
- Édition de contenu de fichier binaire (Linux : *gvim/xxd*, Windows : *notepad++*)

Windows versus Linux

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

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.

¹Ce dernier cas sort du cadre x509.

Le format x509

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)

Figure 1.

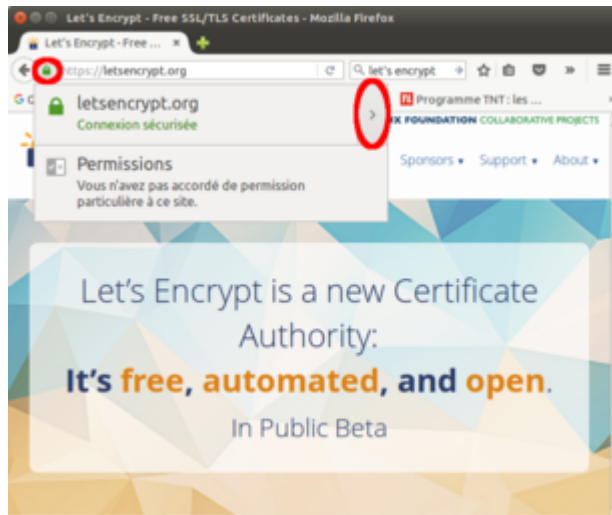


Figure 2.



Figure 3.

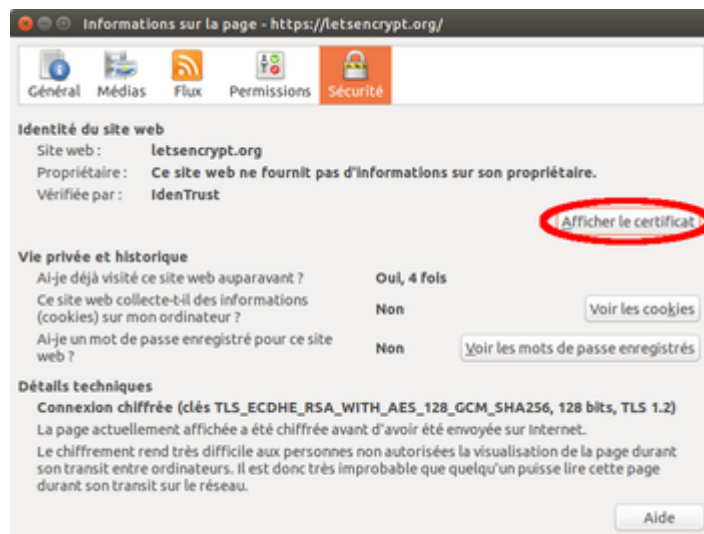


Figure 4.

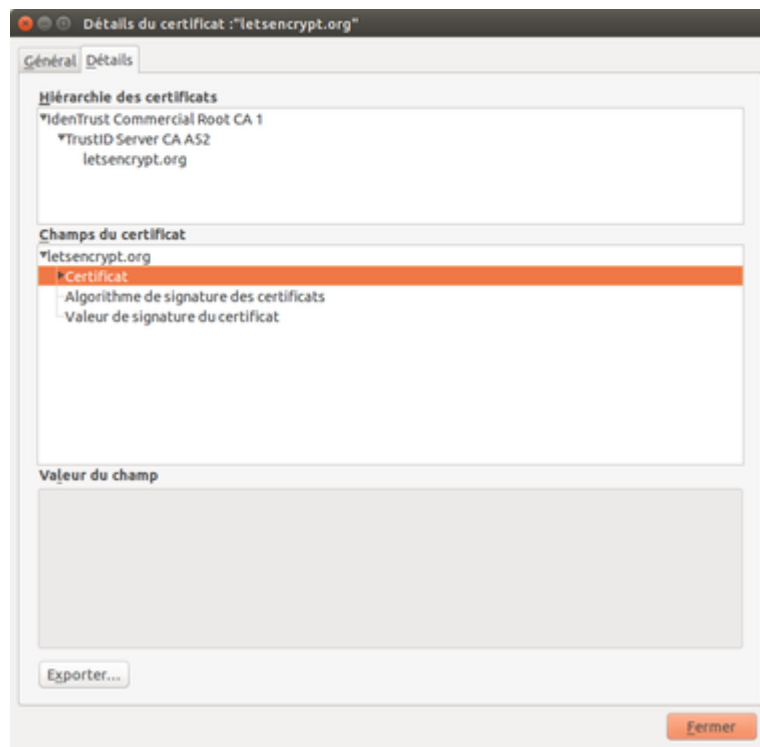


Figure 5.

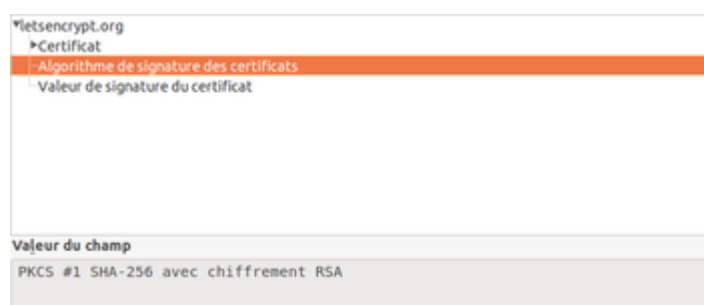
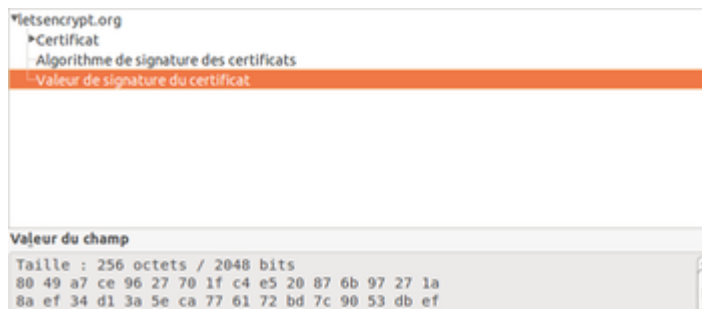


Figure 6.

Nous nous intéresserons à la partie supérieure (*Hierarchie 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.

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 la² **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*.

²Nous 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.

Grammaire, ASN.1 et DER

La structure du certificat est décrite par une *grammaire*, d'après les règles de syntaxe **ASN.1** (Abstract 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

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.

Cela dit, le nom (au sens du standard x509) est constitué du DN (*Distinguished Name*), il s'agit de la *totalité des éléments qui le composent* (et non pas seulement le 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 peut se compliquer avec des certificats croisés. Quoi qu'il en soit, la structure de base des liens qui relient les certificats est hiérarchique.

Quelques précisions

- Pour qu'un certificat joue le rôle émetteur, il doit être une autorité de certification ou CA (*Certificate Authority*), il s'agit de l'une des nombreuses options que l'on peut trouver dans un certificat.
- En plus des éléments permettant d'identifier le sujet du certificat, la partie `tbsCertificate` contient sa *clé publique*. C'est précisément le rôle du certificat : *certifier* la correspondance entre une entité (un DN) et une clé publique.
- L'émetteur (qui est une autorité de certification) a toujours un numéro d'identification de clé (*Subject Key Identifier*). Cet identifiant, qui doit être par construction un entier, doit être enregistré dans les certificats émis comme numéro de clé d'autorité de certification (*Authority Key Identifier*). Cette logique permet de gérer facilement les cas où une autorité de certification possède plusieurs clés.
- En résumé, quand un émetteur signe un sujet, on doit avoir égalité entre ces différents éléments

Dans le certificat émetteur	Lien	Dans le certificat sujet
Subject DN ^a	=	Issuer DN
Subject Key Identifier	=	Authority Key Identifier

^aIci on a un peu simplifié les choses pour ne pas alourdir le texte. En fait l'émetteur est identifié par son DN mais aussi par un *Serial Number*, nécessaire dans les cas où une même entité (DN) est associée à différentes clés. Donc le lien hiérarchique est établi avec le couple (DN, Serial Number), qui correspond à un couple (DN, key) de manière unique. Le DN seul ne serait pas suffisant. Les identifiants de clé correspondent (de manière bi-univoque) à un couple (DN, key).

- La RFC 5280 ([4]) impose l'emploi d'identifiants de clé lors de l'émission de certificats mais la section 4.2.1.2 indique aussi que les applications ne sont pas obligées de vérifier les identifiants de clé lors de la validation d'une chaîne de certification. Conclusion ? Il n'y en a pas.

Figure 7.

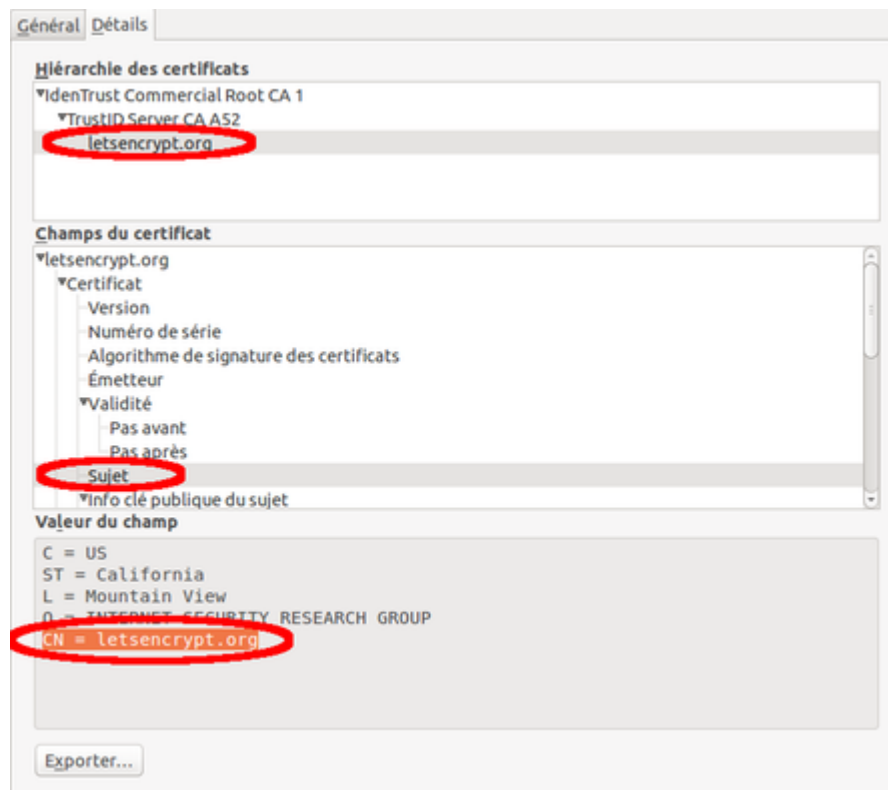
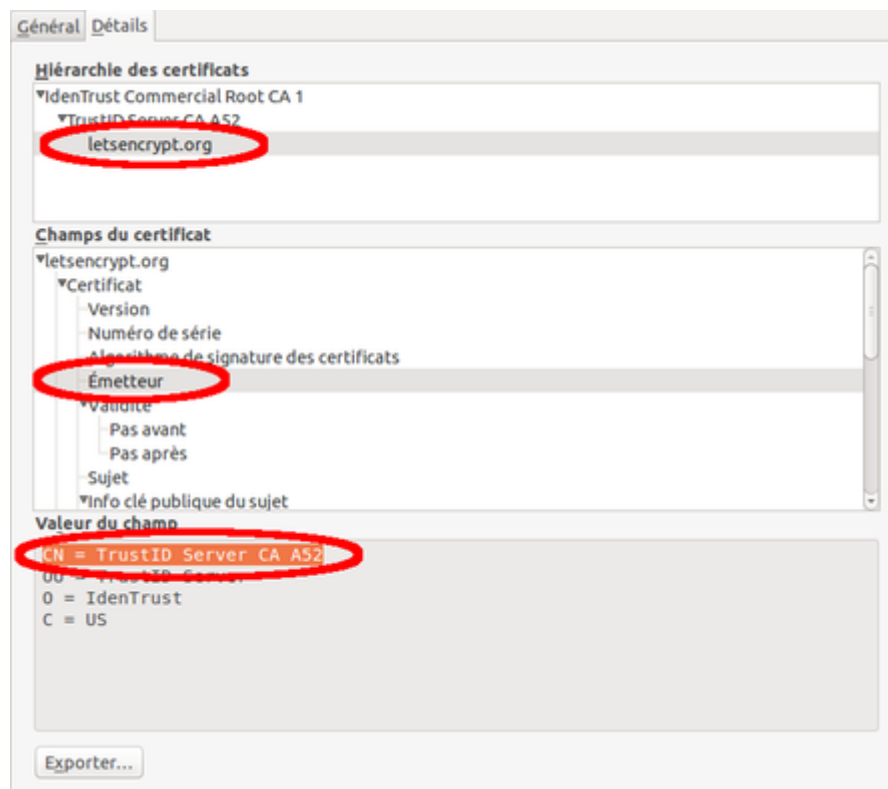


Figure 8.



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](https://letsencrypt.org) de *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.

Vérification de signature RSA

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 [5] 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*.

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 ^{3 4} (notée S) avec la formule $S = M^d \bmod n$

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

Vérification du certificat de letsencrypt.org

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 [6].

- Linux : *bc* est disponible par défaut sur la plupart des distributions.

³On 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.

⁴Ce 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.

- Windows : les binaires sont accessibles à l'URL [7].

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.

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*

bc	python
Saisie d'un entier en hexadécimal	
Exécuter au préalable <code>ibase = 2 * 8^a</code> Exemple : <code>ibase = 2 * 8</code> <code>var = ABEF0E0</code> <i>(Attention les caractères hexadécimaux doivent être en majuscule.)</i>	Saisir l'entier précédé de <i>0x</i> Exemple : <code>>>> var = 0xabef0e0^b</code>
Affichage d'entier en hexadécimal	
Exécuter au préalable <code>obase = 2 * 8^a</code> Exemple : <code>obase = 2 * 8</code> <code>2 ^ (2 ^ 4) + 1^c</code> <code>10001</code>	Interpoler avec <i>%x</i> Exemple : <code>>>> '%x' % (2 ** (2 ** 4) + 1)</code> <code>'10001'</code>

^a2 * 8 produit toujours 16 (décimal). Si 16 est lu alors qu'ibase vaut déjà 16

^b*python* peut lire les nombres hexadécimaux indifféremment en majuscule et minuscule. *bc* exige des majuscules.

^cSous Windows, *bc* a des difficultés à lire le caractère *circonflexe* 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 *circonflexes* et vous devez alors en supprimer un.

⁵Lorsque *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.

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 ⁶.
5. Ajouter $s=$ devant le nombre, et ajouter une première ligne $ibase = 2 * 8$.
6. Enregistrer dans **val.b**.

Voir figures 9 et 10.

Figure 9.

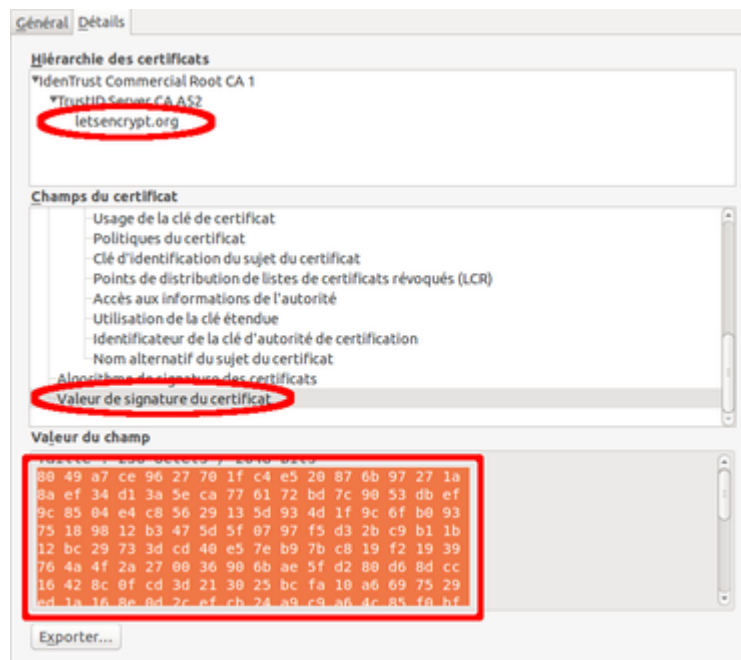
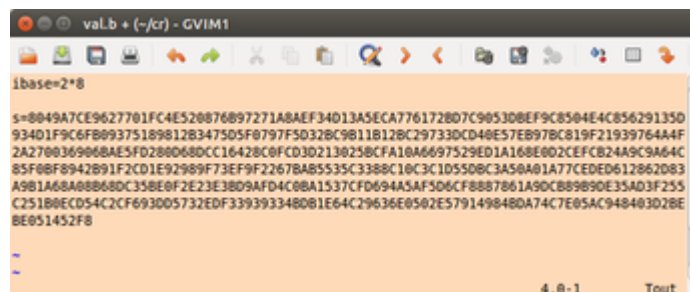


Figure 10.



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.

⁶Si vous utilisez *python*, il est inutile de passer les caractères hexadécimaux en majuscule.

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 :

Example 1. Linux

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

Example 2. Windows

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

Passer du texte en majuscule

Example 3. 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.

Example 4. 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 [8].

Note

vim et *emacs* sont disponibles sous Windows.

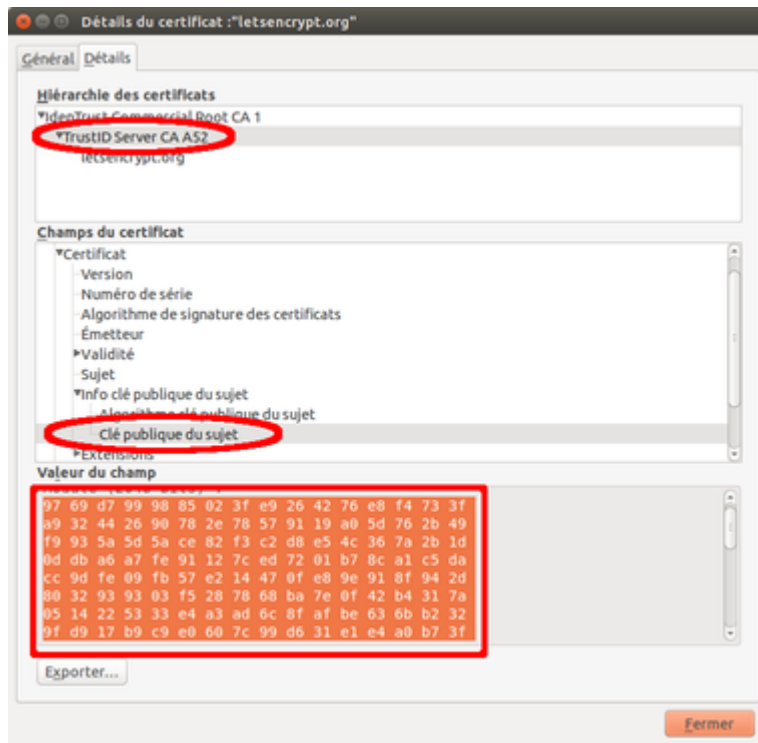
Enregistrement de la clé publique sous forme d'entier

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.

Figure 11.



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

val.b.

ibase=2*8

s=8049A7CE9627701FC4E520876B97271A8AEF34D13A5ECA776172BD7C9053DBEF9C8504E4C8562

n=9769D7999885023FE9264276E8F4733FA932442690782E78579119A05D762B49F9935A5D5ACE8

e=10001

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 [9]. 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 [10]. 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
}

```

Calcul de M

Nous voilà prêts pour calculer M .

1. Lancer la commande suivante ⁷ :

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

2. Dans le shell *bc*, exécuter ^a (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.

Figure 12.

```

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
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF003031300D0609608648016503040201050004208364DA78F1FD8DCC6812E568268BF2DAF87
91BE383109745388879C496A8C3DD

```

Important

Au démarrage de *bc*, *ibase* est égal à 16 du fait du chargement de *val.b*. Si vous exécutez *obase* = 16, *obase* 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 *obase* = 10 (*ibase* étant égal à 16), mais c'est moins clair, et vous devez être certain qu'*ibase* vaut bien 16 au

⁷Par 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.

moment d'exécuter $obase = 10$, 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.

Analyse de M

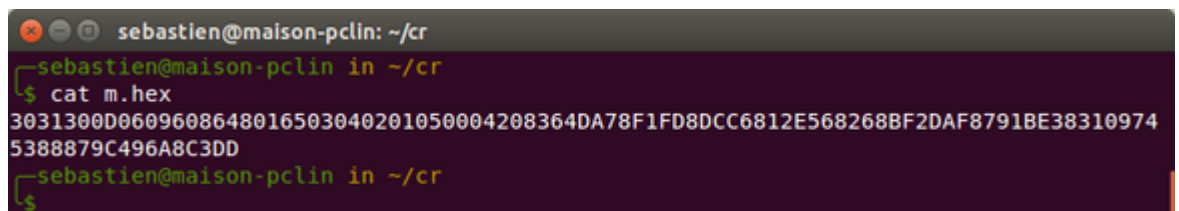
Nous allons procéder en trois étapes :

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**.

Figure 13.



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

Contenu de **m.hex**.

```
3031300D0609608648016503040201050004208364DA78F1FD8DCC6812E568268BF2DAF8791BE383109745388879C496A8C3DD
```

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.

Exemple 5. Linux

Exécuter la commande

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

Exemple 6. Windows

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

Figure 14.

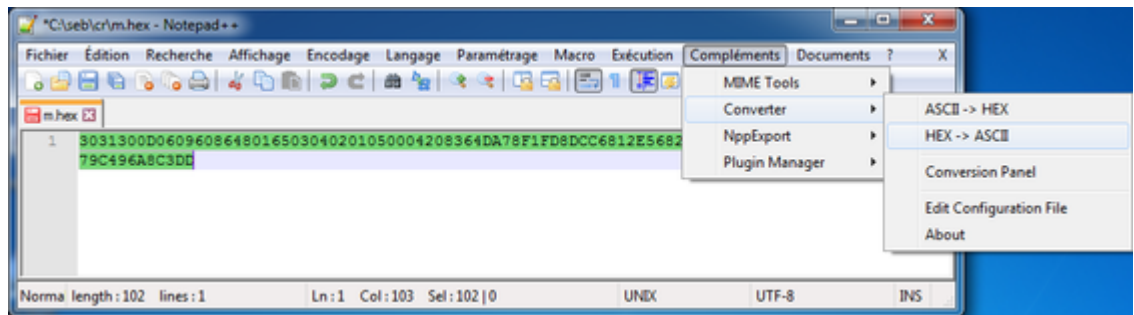
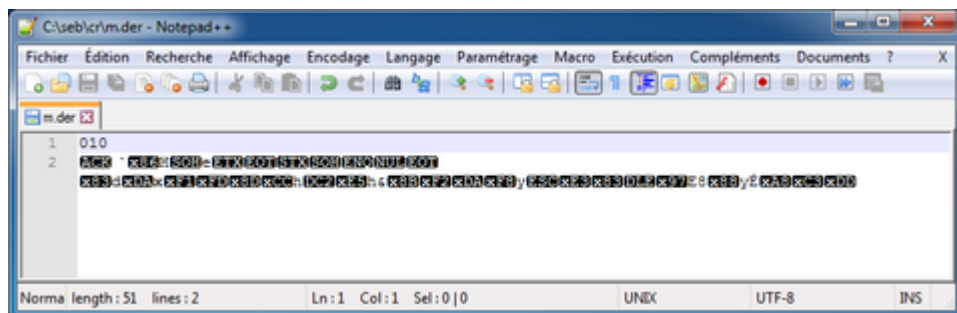


Figure 15.



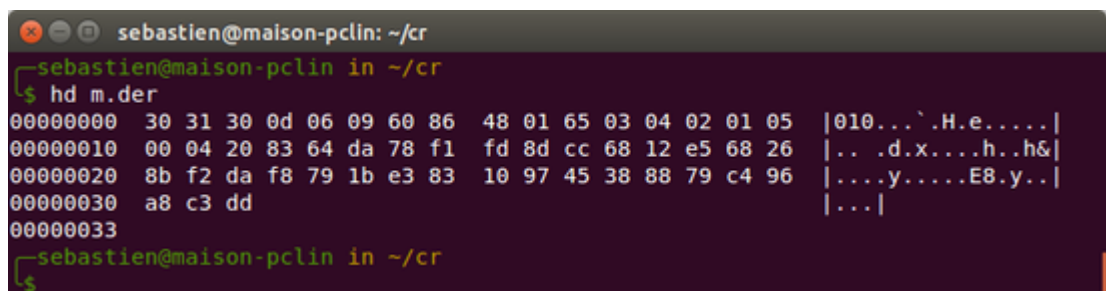
Caution

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.

Figure 16.



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 [11].

pkfile est disponible à l'URL [12].

Exécutable openssl

- 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 [13].

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
```

Figure 17.

```
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
$
```

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' .

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 BER⁸, CER⁹ ou DER¹⁰.

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 qu'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 DER¹¹.

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 <i>primitive</i>	La donnée ne contient pas de sous-structure
Encodage <i>constructed</i>	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 ¹² .
Structure hiérarchique	La différence <i>primitive</i> - <i>constructed</i> est à la base de la structure hiérarchique d'une

⁸BER signifie **B**asic **E**ncoding **R**ules

⁹CER signifie **C**anonical **E**ncoding **R**ules

¹⁰DER signifie **D**istinguished **E**ncoding **R**ules

¹¹La 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.

¹²Certains 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.

spécification ASN.1, les éléments *constructed* étant les branches, les éléments *primitive* étant les feuilles.

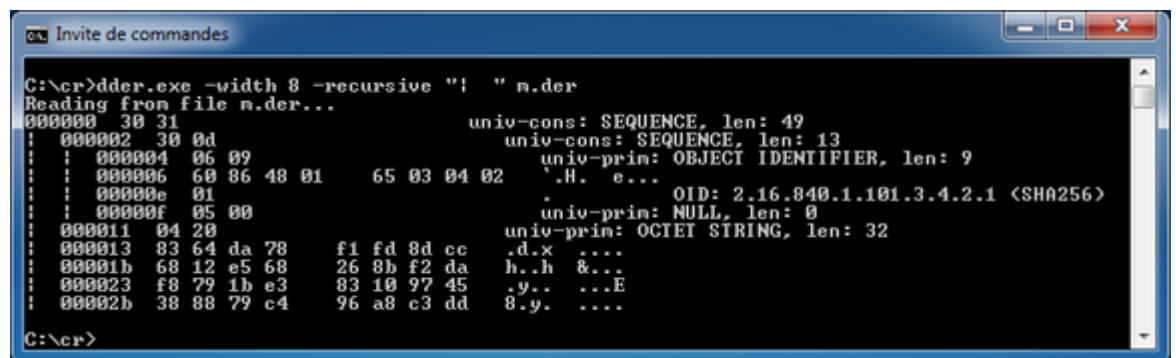
Dans la définition d'un *Certificate* ci-dessus, le type *SEQUENCE* est, par définition du type *SEQUENCE* en ASN.1, *constructed*. Cela signifie que la valeur *SEQUENCE* est elle-même une structure ASN.1, ce qu'on peut voir dans la liste des trois composants d'un certificat définie entre accolades.

[L]length	Donne la longueur de la valeur en octets.
[V]alue	Est la donnée elle-même. Dans ce document nous employons le terme <i>valeur</i> (dans le document X.690 le terme est <i>data value</i>).

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

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

Figure 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* *" | "*.

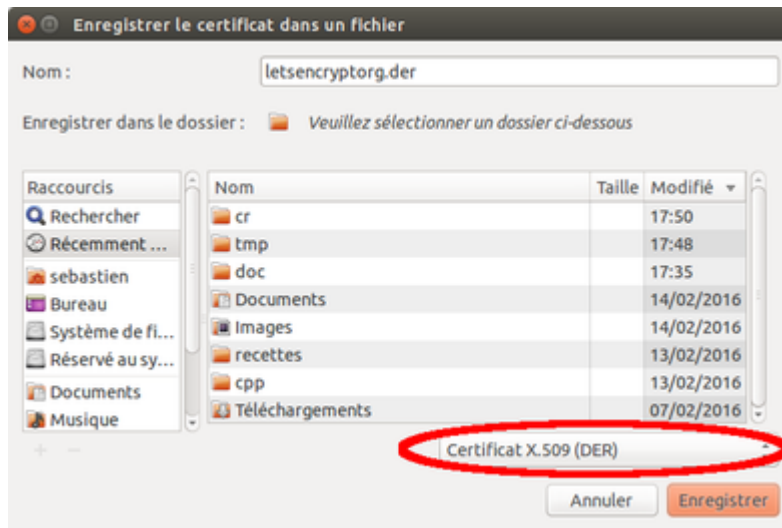
Extraction de tbsCertificate du certificat *letsencrypt.org*

Étape 1 de l'extraction

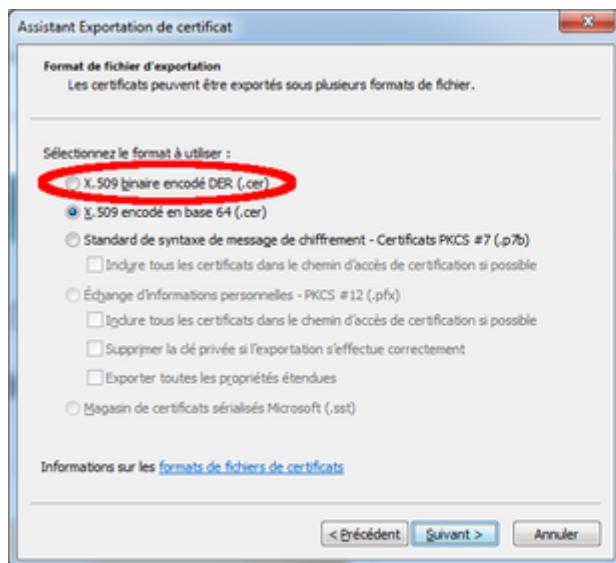
Commençons par enregistrer le certificat depuis le navigateur, dans le fichier *letsencryptorg.der*.

Exemple 7. Firefox

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

Figure 19.**Exemple 8. Internet Explorer et Chrome**

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

Figure 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.cer* (PEM) que l'on convertit en *letsencrypt.der* (DER) - comme on peut voir, l'encodage par défaut avec *openssl* est PEM.

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

Figure 21.

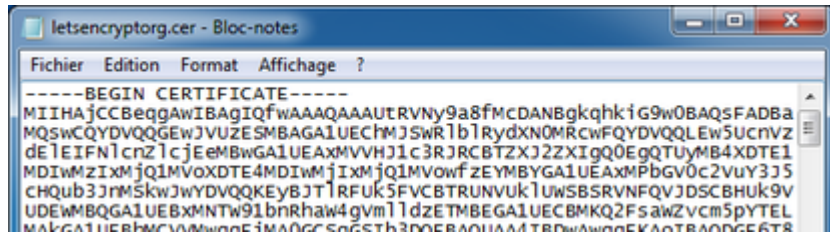
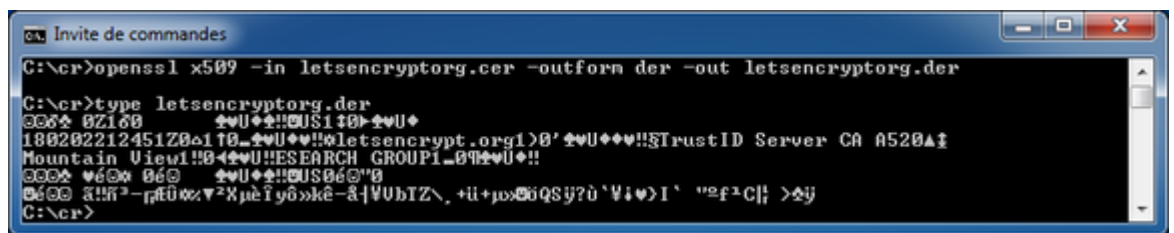


Figure 22.



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 [12].

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.

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

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

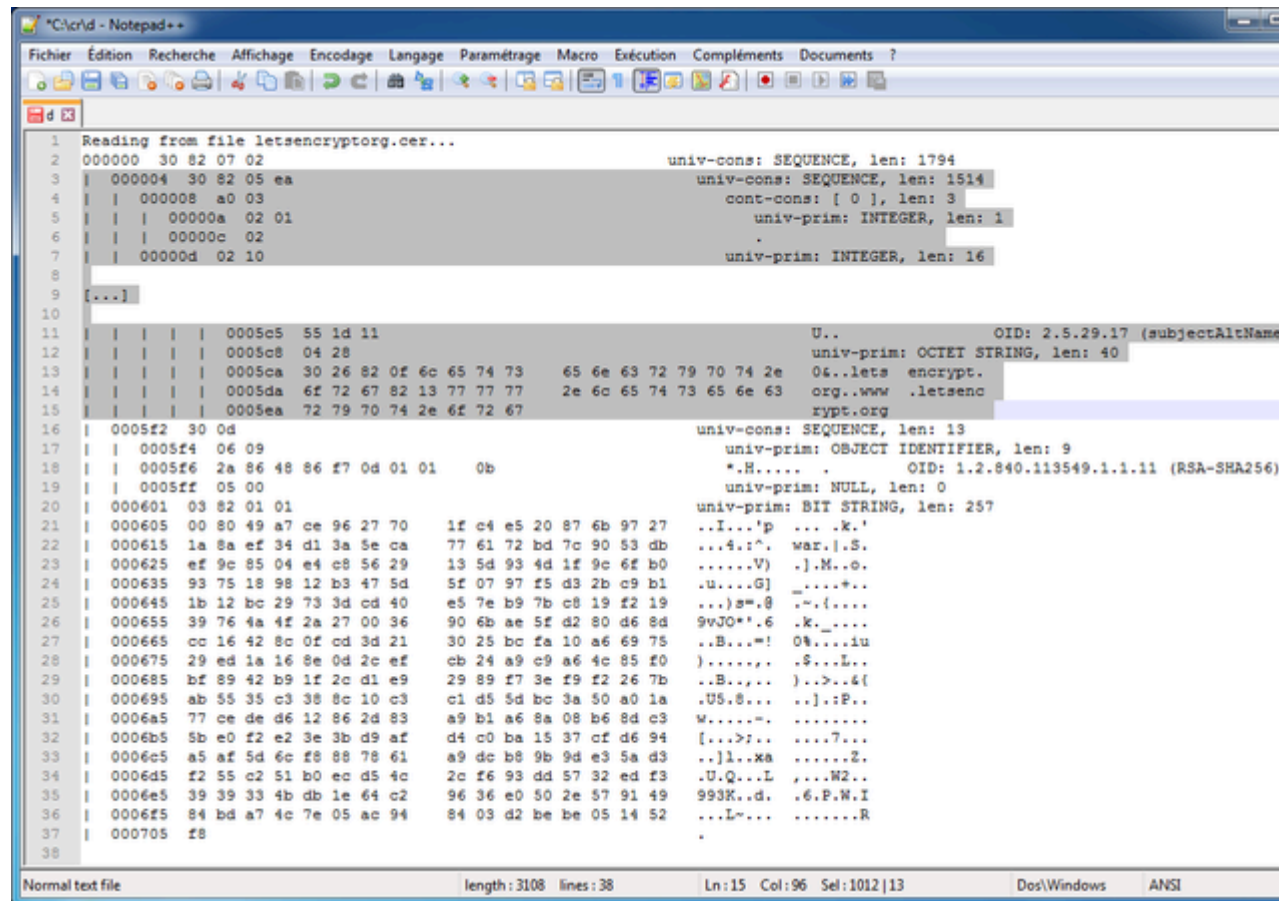
Ouvrir le fichier *d* dans un éditeur de texte. Comme on a affiché le contenu avec un décalage à chaque niveau hiérarchique (option *-recursive "/>*

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).

Figure 23.

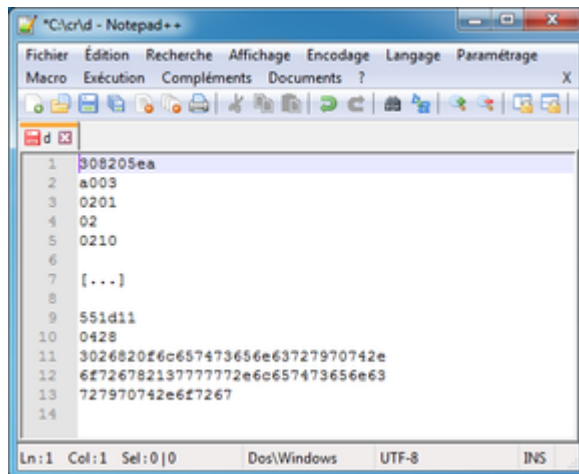
```
C:\cr>dder -recursive "! " -hex letsencryptorg.cer > d
C:\cr>_
```

Figure 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

Figure 25.

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

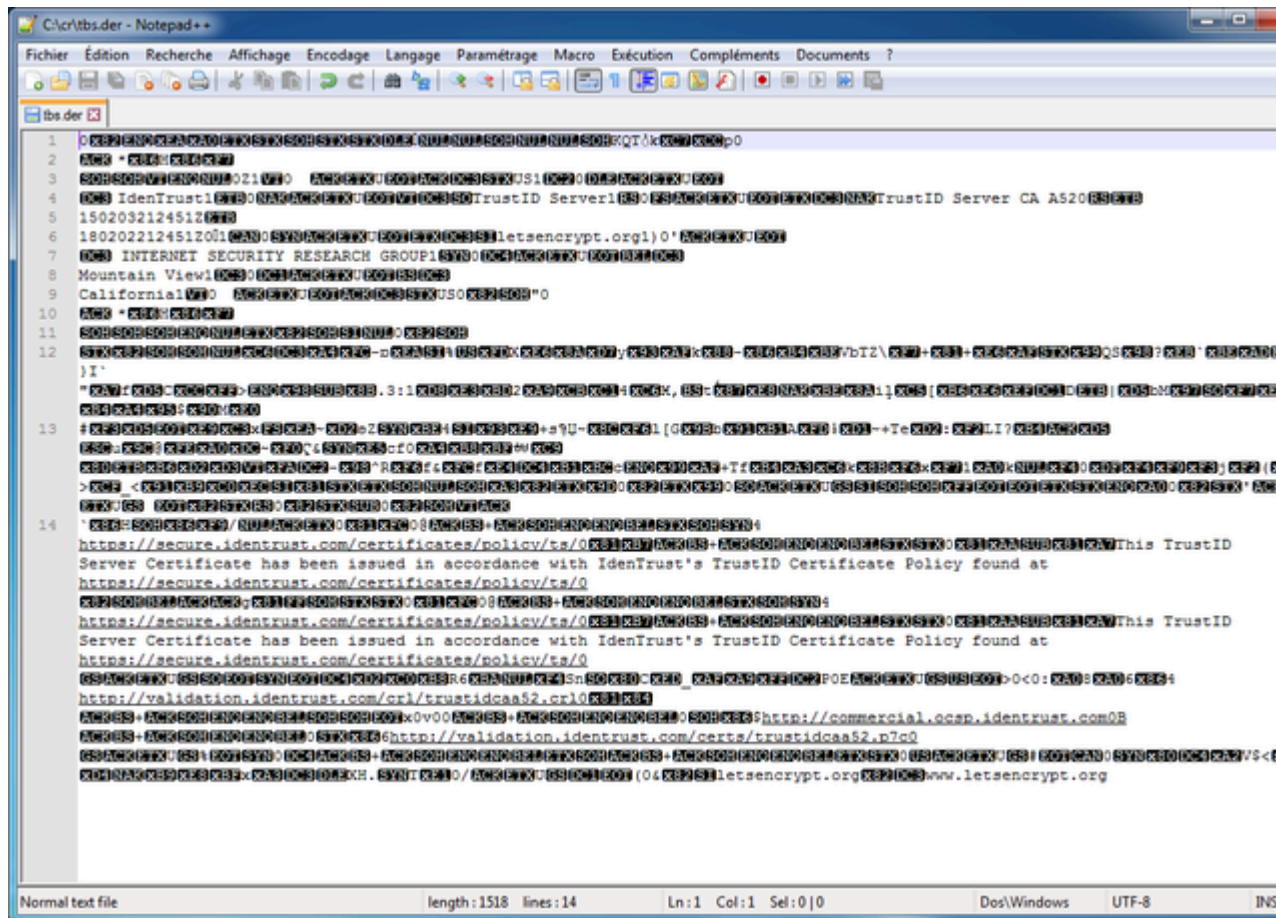
Note

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.

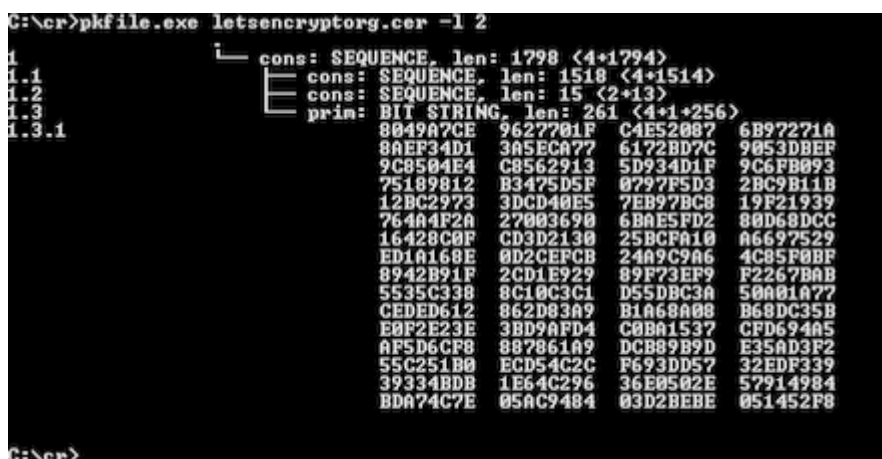
Figure 26.



Extraction de tbsCertificate : alternative 2 (à l'aide de *pkfile*). Exécuter la commande

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

Figure 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
```

Figure 28.

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

Encodage des fichiers analysés par pkfile

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

Calcul du hash de tbsCertificate

Exemple 9. 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
```

Figure 29.

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

Exemple 10. Windows

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 hash¹³.
- Installer les outils *GNUWin32*, disponibles à cette URL [13].
- 7-ZIP ajoute des menus contextuels dans l'explorateur pour calculer différents hashes, dont SHA-256. 7-ZIP est disponible à cette URL [14].

Exemple avec *pkfile*, exécuter

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

Figure 30.

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

Figure 31.

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

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*.

¹³*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.

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

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* (*Certificate Revocation List*).

Vérification de signature ECDSA

Cryptographie à courbes elliptiques

Ce document se concentre sur les problématiques de calcul effectif et de formats de fichiers. Nous n'allons pas expliquer ici toute la cryptographie à base de courbes elliptiques, nous allons simplement en résumer les grandes lignes, donner les formules applicables et quelques définitions.

Les deux familles de courbes elliptiques

Les courbes sur F_{2^m} (*binary field*)

Cette notation est un raccourci. On a $F_{2^n} = F_2[x]/(p(x))$ où $p(x)$ est un polynôme irréductible de degré n (source : [15]). Les "nombres" manipulés sont des polynômes sur F_2 et les opérations sont faites modulo $p(x)$. L'équation de la courbe (source : [16]) est

$$y^2 + xy = x^3 + ax^2 + b$$

Les opérations sur F_{2^m} portent sur des polynômes et leur arithmétique est distincte de l'arithmétique usuelle sur les nombres entiers. Par exemple quel que soit A de F_{2^m} , on a $A + A = 0$ (soit $A = -A$ pour tout A).

Les courbes sur F_p (*prime field*)

p est un nombre premier. On a $F_p = \mathbb{Z}/p\mathbb{Z}$ (source : [17]). Les opérations sont faites modulo p . L'équation de la courbe est (source : [16])

$$y^2 = x^3 + ax + b$$

Les paramètres de domaine

Ces paramètres définissent les caractéristiques de la courbe. Une clé est définie pour une courbe et une seule, la courbe étant entièrement déterminée par ses paramètres de domaine. On parle aussi de paramètres de la courbe.

Source : [16].

- Sur F_{2^m} , les paramètres sont $(m, f(x), a, b, G, n, h)$
 - m , entier qui définit F_{2^m} .
 - $f(x)$, polynôme irréductible de degré m .
 - a et b , les paramètres de l'équation $y^2 + xy = x^3 + ax^2 + b$.
 - G , le générateur du groupe noté $E(F_{2^m})$ (G est donc un point de la courbe).
 - n est l'ordre du groupe généré par G .
 - h est le cofacteur, c'est le nombre d'éléments de $E(F_{2^m})$ divisé par n .
- Sur F_p , les paramètres sont (p, a, b, G, n, h)
 - p , nombre premier.
 - a et b , les paramètres de l'équation $y^2 \pmod{p} = x^3 + ax + b \pmod{p}$.
 - G , le générateur du groupe noté $E(F_p)$ (G est donc un point de la courbe).
 - n est l'ordre du groupe généré par G .
 - h est le cofacteur, c'est le nombre d'éléments de $E(F_p)$ divisé par n .

Les courbes nommées. Toute information pratique sur les courbes elliptiques (clé privée ou publique, signature...) n'est exploitable qu'avec les paramètres de domaine de la courbe elliptique utilisée. Les paramètres de domaine peuvent être donnés de deux manières : tels quels (les différents éléments vus précédemment et qui définissent une courbe, tels que a, b, G , etc.), ou bien sous la forme d'une *courbe nommée*.

openssl peut lister les courbes nommées :

```
$ openssl ecparam -list_curves
secp112r1 : SECG/WTLS curve over a 112 bit prime field
secp112r2 : SECG curve over a 112 bit prime field
[...]
brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
```

La commande suivante permet d'afficher les détails d'une courbe, exemple avec *secp256r1* :

```
$ openssl ecparam -name secp256r1 -noout -text
using curve name prime256v1 instead of secp256r1
ASN1 OID: prime256v1
NIST CURVE: P-256
```

openssl nous avertit que *secp256r1* (définition "SECG" : [18]) est en fait la courbe *NIST P-256* (définie ici [19]), qui porte le nom *prime256v1* dans le standard *X9.62*.

Tout ça pour dire quoi ? Rien, à part que l'on peut se féliciter de vivre à l'heure d'Internet pour s'y retrouver dans la jungle des documents de normalisation.

Et les paramètres de domaine ? Il faut demander à *openssl* d'afficher les paramètres de la courbe sous forme explicite :

```
$ openssl ecparam -name secp256r1 -noout -text -param_enc explicit
using curve name prime256v1 instead of secp256r1
Field Type: prime-field
Prime:
  00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
  00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:ff:ff
A:
  00:ff:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:
  00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:ff:fc
B:
  5a:c6:35:d8:aa:3a:93:e7:b3:eb:bd:55:76:98:86:
  bc:65:1d:06:b0:cc:53:b0:f6:3b:ce:3c:3e:27:d2:
  60:4b
Generator (uncompressed):
  04:6b:17:d1:f2:e1:2c:42:47:f8:bc:e6:e5:63:a4:
  40:f2:77:03:7d:81:2d:eb:33:a0:f4:a1:39:45:d8:
  98:c2:96:4f:e3:42:e2:fe:1a:7f:9b:8e:e7:eb:4a:
  7c:0f:9e:16:2b:ce:33:57:6b:31:5e:ce:cb:b6:40:
  68:37:bf:51:f5
Order:
  00:ff:ff:ff:ff:ff:00:00:00:00:ff:ff:ff:ff:ff:ff:
  ff:ff:bc:e6:fa:ad:a7:17:9e:84:f3:b9:ca:c2:fc:
  63:25:51
Cofactor:  1 (0x1)
Seed:
  c4:9d:36:08:86:e7:04:93:6a:66:78:e1:13:9d:26:
  b7:81:9f:7e:90
```

Tous ces paramètres ont été vus plus haut, sauf *Seed*. Qu'en est-il ?

Seed est un nombre arbitraire à partir duquel les autres paramètres de la courbe ont été déterminés, de manière vérifiable par n'importe qui, selon une spécification *X9.62*. Cela garantit qu'ils n'ont pas été choisis en raison de propriétés cachées qui permettraient à leur inventeur de casser la protection facilement. *Seed* est à l'origine des paramètres de la courbe mais il n'en fait pas partie. Le passage de *Seed* aux paramètres utilise une fonction de hachage à sens unique, ce qui garantit que le choix ne s'est pas fait dans l'autre sens (détermination de paramètres ad hoc puis du *Seed* correspondant).

Le x509. La RFC 5480 ([20]) interdit expressément l'usage de paramètres explicites dans le standard x509, seules les courbes nommées peuvent y être utilisées.

Nous allons maintenant nous intéresser aux courbes sur les corps premiers (F_p). Les courbes binaires fonctionnent selon les mêmes principes mais les "nombres" manipulés sont des polynômes sur F_2 , et

si l'on souhaite y mener des calculs avec *bc* ou *python*, cela nécessite de coder spécifiquement leur arithmétique. Dans ce document nous ferons des calculs sur F_p uniquement.

L'arithmétique des courbes sur F_p

Les calculs sont faits modulo p et nous ne le précisons plus par la suite ¹⁴.

Clés privée et publique. La construction d'une paire de clés privée et publique consiste à

1. Choisir un entier k inférieur à p
2. Calculer $Q = kG$ où G est l'un des paramètres de domaine de la courbe (voir ci-après la multiplication d'un point par un scalaire)

Le point Q est la clé publique, l'entier k est la clé privée. Ainsi contrairement au RSA où la clé privée et publique ont même nature (ce sont deux entiers), ici la clé publique est un point tandis que la clé privée est un entier.

Addition de deux points distincts (et non alignés verticalement). La signification de l'addition de deux points est géométrique et on peut le voir ici [21].

Les formules qui en découlent sont les suivantes, pour $R = P + Q$ de coordonnées (r_x, r_y) , (p_x, p_y) et (q_x, q_y) respectivement, P étant différent de Q et non aligné verticalement avec Q :

$$s = \frac{p_y - q_y}{p_x - q_x}, \quad r_x = s^2 - p_x - q_x, \quad r_y = s(p_x - r_x) - p_y$$

Addition d'un point avec lui-même. Pour $P = Q$, le calcul de $R = P + P = 2P$ est (en reprenant les notations précédentes) :

$$s = \frac{3(p_x^2 + a)}{2p_y}, \quad r_x = s^2 - 2p_x, \quad r_y = s(p_x - r_x) - p_y$$

Important

Le symbole de fraction n'est pas la division euclidienne mais la multiplication par l'inverse modulo p .

Multiplication d'un point par un entier naturel. Avec les formules qui précèdent, on sait calculer $P + P = 2P$, $P + 2P = 3P$, et ainsi de suite. On sait donc calculer kP , k entier.

Addition de deux points alignés verticalement

Si P et Q sont alignés verticalement ils sont opposés (cela découle du terme y^2 de l'équation), on a donc $P = -Q$ que l'on peut aussi écrire $P + Q = O$. O est le "point à l'infini" de la courbe elliptique et il joue le même rôle que zéro dans l'addition des entiers. Ainsi on a $P + O = O + P = O$ quel que soit P sur la courbe.

Multiplication par un entier k très grand. Les k utilisés en pratique étant des grands nombres (de l'ordre de 2^{256} avec *prime256v1*), il n'est pas possible de calculer kP en ajoutant P à lui-même k fois.

¹⁴Dans la signature ECDSA certains calculs sont faits modulo n , l'ordre du groupe associé à la courbe. Cela sera signalé. À défaut, tous les calculs sont faits modulo p .

Il faut utiliser l'algorithme d'exponentiation rapide vu précédemment : [9]. Remplacer la multiplication de deux entiers par l'addition de deux points et l'élévation à la puissance par la multiplication d'un point par un entier.

Fonctions de calcul sur F_p avec le programme *bc*

Nous allons maintenant écrire les fonctions de calcul d'après les formules qui précèdent.

Un point est représenté par un tableau et la convention du script est la suivante :

- Le point à l'infini est codé par *point*[2] différent de zéro
- Sinon (si *point*[2] est nul), *point*[0] est la coordonnée *x* et *point*[1] est la coordonnée *y*.

Le calcul de l'inverse modulaire utilise l'*algorithme d'Euclide étendu* [22].

Note

Ce script contient la fonction *powmod* utilisée précédemment, pour le RSA. Nous n'en avons pas besoin pour les calculs ECDSA.

L'opérateur modulo

L'opérateur % (*modulo*) ne fonctionne pas à l'identique d'un langage à l'autre ou d'une librairie à l'autre.

- *bc* renvoie un nombre dont le signe est le signe du premier argument. Ainsi $(-5) \% 3$ renvoie -2.
- *python* renvoie toujours une valeur positive. $(-5) \% 3$ renvoie 1.
- La librairie *libgmp* (*GNU Multiple Precision Arithmetic Library*) renvoie toujours une valeur positive. $(-5) \% 3$ renvoie 1.

Cela ne change pas la nature des calculs effectués mais doit parfois être pris en compte. Ainsi la fonction *invmod* (définie dans *crypto.b* ci-dessous) ne peut pas fonctionner avec des nombres négatifs et elle commence par "normaliser" le premier argument pour qu'il soit compris entre 0 et $n - 1$.

crypto.b.

```
/* Calculate the invert of a modulo n */
define invmod(a, n) {
    auto aa, bb, r, t, anc_t, nou_t, negflag
    aa = n
    negflag = 0
    if (a < 0) {
        negflag = 1
        a %= n
        if (a < 0) a += n
    }
    bb = a
    r = 1
    t = 1
    anc_t = 0
    while (1) {
        q = aa / bb
        anc_r = r
        r = aa - bb * q

        nou_t = anc_t - q * t
        if (nou_t >= 0) nou_t %= n
```

```

        if (nou_t < 0) nou_t = n - (-nou_t % n)
        anc_t = t
        t = nou_t

        aa = bb
        bb = r
        if (r <= 1) break;
    }

    if (r != 1) {
        /*
            No invert can be returned => error
            Alternate solution: return -1

            I find triggering an error best: the calculation stops
            immediately instead of continuing with meaningless values
        */
        return 1 % 0
    } else {
        if (negflag) t -= n
        return t
    }
}

/*
    The well-known powmod function, sometimes referred to in bc scripts
    as "mpower".
*/
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
}

/*
    The EC functions below use the following conventions.

    - The point at infinity has non null element of index 2 (array[2]!=0),
      for any other point, array[2] is zero.

    - If array[2] is zero, array[0] is the x coordinate and
      array[1] is the y coordinate.
*/

/*
    ECC addition of p and q, p[0] being different from q[0]
*/
define void ec_add_core(*r[], p[], q[], m) {
    auto s
    s = ((p[1] - q[1]) * invmod(p[0] - q[0], m)) % m
    r[0] = (s^2 - p[0] - q[0]) % m

```

```

        r[1] = (s * (p[0] - r[0]) - p[1]) % m
        r[2] = 0
    }

    /*
     * ECC point doubling
     */
    define void ec_dbl_core(*r[], p[], a, m) {
        auto s
        s = ((3 * p[0]^2 + a) * invmod(2 * p[1], m)) % m
        r[0] = (s^2 - 2 * p[0]) % m
        r[1] = (s * (p[0] - r[0]) - p[1]) % m
        r[2] = 0
    }

    /*
     * ECC addition of p and q for any value of p and q
     */
    define void ec_add(*r[], p[], q[], a, m) {
        if (p[2]) { r[0] = q[0]; r[1] = q[1]; r[2] = q[2]; return }
        if (q[2]) { r[0] = p[0]; r[1] = p[1]; r[2] = p[2]; return }
        if (p[0] == q[0]) {
            if (p[1] != q[1]) {
                r[2] = 1 /* We don't verify whether p[1]==-q[1] as it is
                           * already checked in the caller
                           */
            } else {
                if (p[2]) {
                    r[2] = 1
                    return
                }
                ec_dbl_core(r[], p[], a, m)
            }
        } else {
            ec_add_core(r[], p[], q[], m)
        }
    }

    /*
     * ECC scalar point multiplication
     */
    define void ec_mul(*r[], p[], k, a, m) {
        auto tmp[]
        r[2] = 1
        if (p[2]) return
        while (k > 0) {
            if ((k % 2) == 1) {
                ec_add(r[], r[], p[], a, m)
            }

            /*
             * ec_dbl(p[], p[], a, m) does not work with bc,
             * need to use a temporary array.
             */
            ec_add(tmp[], p[], p[], a, m)
            p[0]=tmp[0]
            p[1]=tmp[1]
            p[2]=tmp[2]
            k /= 2
        }
        if (!r[2]) {

```

```

        if (r[0] < 0) r[0] += m
        if (r[1] < 0) r[1] += m
    }
}

```

Optimisation et robustesse

Sur Internet on trouve des scripts qui appliquent la première formule (addition de deux points distincts) et la deuxième (doubler un point) sans vérifier les cas "P et Q alignés verticalement" ou "Point à l'infini".

Dans la pratique ce n'est pas un souci car les nombres manipulés avec des clés x509 ne conduisent jamais à ces cas limites. La fonction *ec_mul* pourrait ainsi être plus efficace, en appelant directement les fonctions *ec_add_core* et *ec_dbl_core*.

Représentation d'un point dans le standard x509

Un point de la courbe d'abscisse x donnée ne peut avoir que deux valeurs de y (du fait du terme y^2 des équations).

Le standard X9.62 définit la méthode pour désigner quel y est le bon, en utilisant un seul bit d'information. Ainsi l'enregistrement de x et du bit d'information pour retrouver le bon y suffisent à coder un point, et cette forme est dite *compressée*. Comme il est en plus possible de mélanger les formes (compressée et non compressée), un point peut être codé de trois manières :

La forme non compressée	Enregistrement des deux coordonnées
La forme compressée	Enregistrement de x et du bit d'information relatif à y
La forme hybride	Enregistrement de x , y et du bit d'information relatif à y

Sur F_p on a $y_1 = -y_2$ (car non seulement y apparaît sous la forme y^2 , mais en plus, il est isolé à gauche du signe *égal*) et comme nous sommes dans l'arithmétique modulaire on peut ré-écrire cette égalité $y_1 + y_2 = p$. p étant impair, y_1 et y_2 n'ont pas même parité. Le bit d'information est le dernier bit de y (sa parité). On comprend aisément que la forme hybride ne présente aucun intérêt sur F_p , puisqu'un simple examen de y donne le bit d'information à employer dans la forme compressée. La forme hybride apporte un plus sur F_{2^m} où le bit d'information doit être calculé.

Le standard x509 (ici la RFC 5480) permet de coder un point soit sous forme compressée, soit sous forme non compressée. La RFC 5480 interdit expressément la forme hybride définie par le standard X9.62.

Selon le standard X9.62 en vigueur dans la RFC 5480 (nous ne mentionnons donc pas ici la forme hybride, interdite par la RFC 5480) les formes compressée et non compressée sont codées ainsi¹⁵ :

- Premier octet = 02 : le point est compressé, et y est pair. x est codé juste après 02.
- Premier octet = 03 : le point est compressé, et y est impair. x est codé juste après 03.
- Premier octet = 04 : le point est non compressé. x est codé juste après 04, et y est codé juste après x . Comme x et y ont même taille, il n'y a pas besoin de délimiteur pour retrouver la position à laquelle on passe de x à y .

La courbe prime256v1 (secp256r1)

Les paramètres de la courbe prime256v1. Récupérons les paramètres (depuis *openssl* ou depuis ce document [18]) de la courbe et enregistrons-les dans le fichier *secp256r1.b*.

¹⁵Le standard X9.62 définit 00 comme seul octet pour coder le point à l'infini. La RFC 5480 interdit toute autre valeur que 02, 03 et 04, et interdit donc 00. C'est logique car on ne voit pas dans quel but le point à l'infini serait enregistré dans le cadre x509. Dans le standard x509, un point est une clé publique.

secp256r1.b.

Au passage, on peut vérifier que le point G fait bien partie de la courbe :

```

❌ BC_LINE_LENGTH=0 bc secp256r1.b
$ BC_LINE_LENGTH=0 bc secp256r1.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=2^4
(g[0]^3+a*g[0]+b)%p
55DF5D5850F47BAD82149139979369FE498A9022A412B5E0BEDD2CFC21C3ED91
g[1]^2%p
55DF5D5850F47BAD82149139979369FE498A9022A412B5E0BEDD2CFC21C3ED91

```

Figure 33.

```

$ BC_LINE_LENGTH=0 bc secp256r1.b crypto.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'.
ec_mul(r[], g[], n, a, p)
r[2]
1

```

```
$ openssl ec -in k.pem -noout -text
```

33

Figure 34.

```

sebastien@maison-pclin: ~
sebastien@maison-pclin:~$ openssl ec -in k.pem -noout -text
read EC key
Private-Key: (256 bit)
priv:
  67:3d:ca:d1:21:e3:7b:7e:15:ac:79:2f:8c:a0:67:
  08:56:e2:de:9c:23:36:84:a9:a0:fa:3d:76:7b:61:
  fb:f1
pub:
  04:13:4d:4e:5f:63:2e:05:45:35:0f:b8:9e:e0:52:
  00:45:0e:07:a5:b1:15:bc:a7:2d:f5:f4:89:6d:24:
  33:a6:5a:a1:a3:3a:97:ac:b4:e4:0d:05:19:cb:02:
  9b:1e:85:e2:6d:3d:6b:3d:7f:f3:d3:62:61:27:d0:
  c0:45:39:2d:91
ASN1 OID: prime256v1
NIST CURVE: P-256
sebastien@maison-pclin:~$

```

Par défaut, *openssl* enregistre les clés publiques sous forme non compressée, ce que l'on peut voir au 04 qui démarre la suite d'octets en-dessous de "pub:".

À l'aide de la commande *openssl* ci-dessus (ou bien avec *pkfile*, ou encore *dder*), copier/coller les valeurs de la clé privée et publique dans *k.b*, sous les noms *k* et *q[]*, respectivement.

k.b.

```

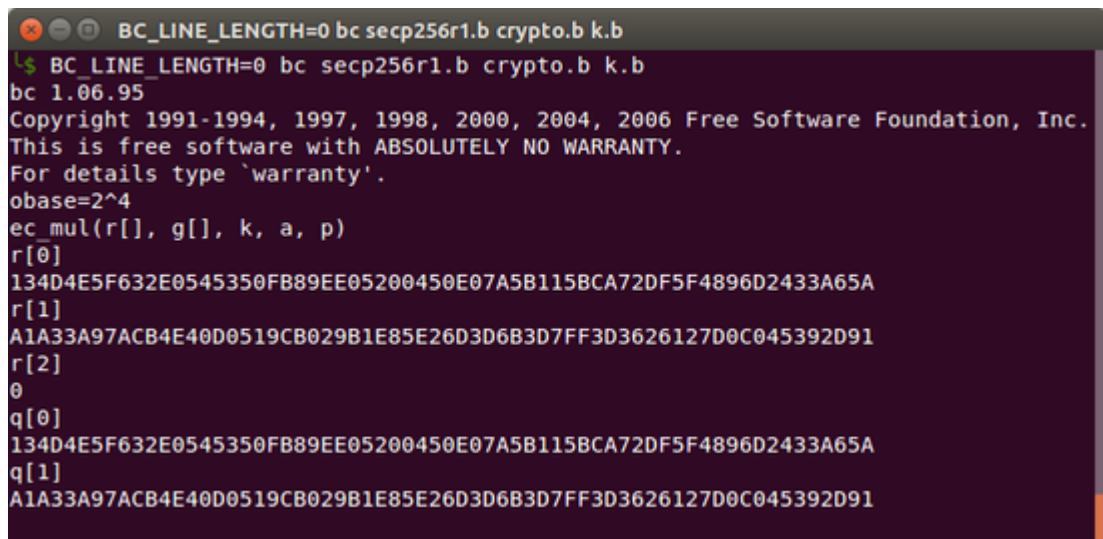
save_ibase=ibase
ibase=2^4
k=673DCAD121E37B7E15AC792F8CA0670856E2DE9C233684A9A0FA3D767B61FBBF1
q[0]=134D4E5F632E0545350FB89EE05200450E07A5B115BCA72DF5F4896D2433A65A
q[1]=A1A33A97ACB4E40D0519CB029B1E85E26D3D6B3D7FF3D3626127D0C045392D91
q[2]=0
ibase=save_ibase

```

Calcul de la clé publique d'après la clé privée. On peut facilement calculer Q (la clé publique) d'après k (la clé privée). Au passage, cela permet de vérifier le bon fonctionnement de la chaîne :

1. Les fonctions de *crypto.b*
2. Les paramètres de domaine de *prime256v1*
3. La définition dans *k.b* des valeurs de la paire de clés privée et publique

Figure 35.



```

BC_LINE_LENGTH=0 bc secp256r1.b crypto.b k.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=2^4
ec_mul(r[], g[], k, a, p)
r[0]
134D4E5F632E0545350FB89EE05200450E07A5B115BCA72DF5F4896D2433A65A
r[1]
A1A33A97ACB4E40D0519CB029B1E85E26D3D6B3D7FF3D3626127D0C045392D91
r[2]
0
q[0]
134D4E5F632E0545350FB89EE05200450E07A5B115BCA72DF5F4896D2433A65A
q[1]
A1A33A97ACB4E40D0519CB029B1E85E26D3D6B3D7FF3D3626127D0C045392D91

```

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] RFC 5280 (certificats x509 v3) : <https://tools.ietf.org/html/rfc5280>
- [5] Article Wikipédia consacré au RSA : https://fr.wikipedia.org/wiki/Chiffrement_RSA
- [6] Page d'accueil de *bc* : <https://www.gnu.org/software/bc/>
- [7] *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>
- [8] Le site de notepad++ : <https://notepad-plus-plus.org/fr>
- [9] Article Wikipédia sur l'algorithme d'exponentiation rapide : https://fr.wikipedia.org/wiki/Exponentiation_rapide
- [10] Article Wikipédia sur l'exponentiation modulaire : https://fr.wikipedia.org/wiki/Exponentiation_modulaire
- [11] Obtenir *dder* : <http://dder.sourceforge.net>
- [12] Obtenir *pkfile* : <http://pkfile.sourceforge.net>
- [13] GNUWin32 : <http://gnuwin32.sourceforge.net/packages/openssl.htm>
- [14] Télécharger 7-ZIP : <http://www.7-zip.org/download.html>
- [15] Peter Seibt, Andreea Dragut Cours de cryptographie Chapitre V (format PDF) : <http://www.dil.univ-mrs.fr/~jfp/master/m03/CompresChap5AES.pdf>

- [16] An introduction to elliptic curve cryptography : <http://www.embedded.com/design/safety-and-security/4396040/An-Introduction-to-Elliptic-Curve-Cryptography>
- [17] Corps finis (format PDF) : <http://iml.univ-mrs.fr/~rodier/Cours/RappelCorps%20finis.pdf>
- [18] SEC 2: Recommended Elliptic Curve Domain Parameters (format PDF) : <http://www.secg.org/SEC2-Ver-1.0.pdf>
- [19] RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE (format PDF) : <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>
- [20] Elliptic Curve Cryptography Subject Public Key Information : <https://tools.ietf.org/html/rfc5480>
- [21] "Courbe elliptique" sur Wikipédia : https://fr.wikipedia.org/wiki/Courbe_elliptique#Additionner_les_points:_d.C3.A9finition_par_la_m.C3.A9thode_des_tangentes_et_des_s.C3
- [22] Article Wikipédia sur l'inverse modulaire : https://fr.wikipedia.org/wiki/Inverse_modulaire
- [23] Conception d'algorithmes et applications (LI325) COURS 10 (format PDF) : <http://www.di.ens.fr/~busic/cours/LI325/coursCAAC10.pdf>
- [99] ECC Tutorial (en anglais) : <http://www.johannes-bauer.com/compsci/ecc>
- [98] Paramètres de domain brainpool (format PDF) : <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>