

# Cryptographie Avancée

## Projet Master 1 informatique (contrôle continu)

**Déroulement général :** chaque Lab est prévu pour durer approximativement 2 à 3h, par binôme (monômes et trinômes tolérés). Le projet doit être développé en python en évitant au maximum les librairies exotiques (en fait, vous n'êtes pas sensé en utiliser). Déposer le fichier source sur la plateforme **eCampus**, une semaine après la fin de chaque cours (au plus tard 10 jours).

Un fichier `readme.md` est le bienvenu pour accompagner le code source, par exemple pour laisser une trace d'exécution commentée ou détailler ce qui ne va pas. Indiquer vos noms, prénoms et mail en début de code ou dans le `readme.md`.

**Consignes pour le code source :** même si ce n'est pas la priorité de ce projet, la qualité du code source entre dans la note. Voici quelques règles de base attendues (si certaines sont évidentes, d'autres peuvent être adaptées) :

1. Le programme doit compiler sous Linux (version des machines de TP par défaut) sans erreurs. En particulier, il ne doit pas y avoir d'erreurs dues à un mélange entre tabulations et espaces. Si une fonction cause une erreur à la compilation, on mettra son appel en commentaire.
2. Lors de l'exécution, le programme doit afficher tout ce qui est demandé de manière lisible. Par exemple, Lab1, question 1 : ..., Lab1, question 2 : ... ou encore Lab1, test Square-and-Multiply :..., etc...
3. Lors de l'exécution, le programme ne doit pas afficher des valeurs intermédiaires utilisées pour le debug. Une fois que le programme fonctionne, enlever tout ce qui a été utilisé pour le debug (notamment les `print`, même laissé en commentaire). Les fonctions de test peuvent être laissées.
4. Le code source ne doit pas contenir des fonctions, bout de fonctions ou bout de code commentés qui ne servent à rien dans le programme. De même on évitera les variables, instructions ou fonctions inutiles.
5. Le code source doit être commenté. Au minimum une phrase pour chaque fonction (pas nécessairement les fonctions tests) mais aussi à chaque fois que c'est utile pour la compréhension du code (théorèmes utilisés ou choix d'implémentation par exemple).

## Structure attendue du code source

```
#!/usr/bin/python3
# Noms, prenom, mails

Import des librairies

def MaClasse1(object):
    def __init__(self, A, B,..):
        """Constructeur de la classe où A et B sont ..."""
        self.A = A
        self.B = B
        ...

    def maMethode1.1(self, arg1, arg2, ..):
        """arg1 et arg2 sont des entiers. Retourne ..."""
        if self.A < arg1: ...

def MaClass2(MaClasse1):
    def __init__(self, C,..):
        """Constructeur de la classe où C est ..."""
        MaClasse1.__init__(self, A, B,..)
        self.C = C
        ...

    def maMethode2.1(self, arg1, arg2, ..):
        """arg1 et arg2 sont des entiers. Retourne ..."""
        x = self.maMethode1.1(arg1, arg2,..)
        ...

def test_lab1():
    ....

def test_lab2():
    ....

...

def test_lab5():
    ....

if __name__ == "__main__":
    test_lab1()
    test_lab2()
    ....
    test_lab5()
```

## Lab 1 : Cryptographie dans un groupe cyclique

Définir une classe **Groupe** avec quatre attributs : un entier premier  $p$ , l'élément neutre du groupe  $e$ , l'ordre du groupe  $N$  ainsi qu'une chaîne de caractère  $l$  qui va décrire la loi du groupe. Implémenter le constructeur correspondant. Ecrire une méthode `loi` qui prend en argument deux éléments d'un groupe  $g_1$  et  $g_2$  et retourne  $g_1 * g_2$ , où  $*$  est la loi de groupe. Dans un premier temps, on utilise le groupe additif  $\mathbb{Z}_p$ , donc la fonction doit juste retourner  $(g_1 + g_2) \bmod p$  si  $l = \text{'Zp-additif'}$  et déclencher une exception sinon.

Munir la classe de deux méthodes `squareAndMultiply` et `MontgomeryLadder` qui prennent en argument un élément d'un groupe  $g$  et un entier  $k$  et qui retournent l'élément  $g^k$  à l'aide de l'algorithme *Square and Multiply* ou *Montgomery Ladder* en faisant appel à `loi`. Tester vos deux méthodes sur  $\mathbb{Z}_p$  avec  $p = 23$ ,  $g = 5$  et  $k = 7$  (elles doivent retourner 12). Si vous n'avez pas pris de précautions, l'appel à l'une des deux méthodes avec  $k = 0$  retourne une erreur de type *math domain error* (du au calcul d'un logarithme de 0). Comme l'on sait que  $g^0 = e$ , modifier les deux méthodes et retourner directement  $e$  si  $k = 0$ .

Pour la même raison si  $k$  est négatif, il y aura aussi une erreur. Nous allons simplifier le problème en ne traitant que le cas  $k = -1$ . Dans ce cas on aimerait que ces méthodes retournent l'inverse de  $g$ . Par le théorème de Lagrange, on a  $g^N = g * g^{N-1} = e$ , donc l'inverse de  $g$  est  $g^{N-1}$ . Modifier les deux méthodes précédentes pour qu'elles retournent l'inverse de  $g$  si  $k = -1$  (il suffit de changer  $k$  en  $N - 1$  en début d'algorithme). Tester vos deux méthodes sur  $\mathbb{Z}_{23}$  avec  $g = 5$  et  $k = -1$  (elles doivent retourner 18 car  $5 + 18 = 23$ ).

Ecrire une classe **Crypto** qui hérite de la classe **Groupe** avec un attribut supplémentaire  $g$  (l'élément générateur du groupe) et implémenter le constructeur. Munir cette classe d'une méthode `testDiffieHellman` qui réalise un échange de clé Diffie Hellman. Plus précisément, dans cette méthode, on génère deux entiers  $a$  et  $b$ , aléatoires entre 0 et  $N - 1$ , puis on utilise `squareAndMultiply` pour calculer  $g^a$  et  $(g^b)^a$  et `MontgomeryLadder` pour calculer  $g^b$  et  $(g^a)^b$ . La fonction retourne **True** si on obtient le même élément de groupe  $g^{ab}$  et **False** sinon (ce qui ne doit pas arriver). Tester la fonction dans  $\mathbb{Z}_{23}$  avec  $g = 5$

Munir **Crypto** d'une méthode `DiffieHellman` qui prend en argument deux entiers  $a$  et  $b$  et trois éléments du groupes  $A$ ,  $B$  et  $K$  et qui renvoie **True** si  $A = g^a$ ,  $B = g^b$  et  $K = A^b = B^a$  et **False** sinon. Tester la méthode dans  $\mathbb{Z}_{23}$  avec  $g = 5$ ,  $a = 5$ ,  $b = 6$ ,  $A = 2$ ,  $B = 7$  et  $K = 12$ .

## Lab 2 : Cryptographie dans le groupe des inversibles $\mathbb{Z}_p^\times$

Nous souhaitons désormais pouvoir instancier le groupe multiplicatif  $\mathbb{Z}_p^\times$ . Compléter `loi` dans ce sens en y ajoutant le cas où  $l = \text{'Zp-multiplicatif'}$ . Tester les méthodes `squareAndMultiply` et `MontgomeryLadder` sur ce groupe, avec  $p = 23$ ,  $g = 5$  (qui est une racine primitive modulo 23) et  $k = 7$ . Vérifier aussi que l'inverse de 5 modulo 23 est 14. Remarque : l'exponentiation modulaire  $g^k \bmod p$  et l'inverse modulaire  $g^{-1} \bmod p$  se calculent très bien avec la fonction `pow`, mais on privilégiera les méthodes précédentes si possible dans ce projet.

Générer un fichier `dh_param.pem` contenant les paramètres proposés par la RFC 5114 pour Diffie-Hellman, c'est-à-dire un premier  $p$  de 2048 bits, un élément  $g$  de  $\mathbb{Z}_p^\times$  et l'ordre  $N$  (premier de 256 bits) du sous groupe généré par  $g$  :

```
$ openssl genpkey -genparam -algorithm DH -pkeyopt dh_rfc5114:3
-out dh_param.pem
```

Afficher les différents paramètres à l'aide de la commande

```
$ openssl pkeyparam -in dh_param.pem -text
```

DH Parameters: (2048 bit)

```
prime:
  00:87:a8:e6:1d:b4:b6:66:3c:ff:bb:d1:9c:65:19:
  ...
  5f:69:38:77:fa:d7:ef:09:ca:db:09:4a:e9:1e:1a:15:97
generator:
  3f:b3:2c:9b:73:13:4d:0b:2e:77:50:66:60:ed:bd:
  ...
  5e:23:27:cf:ef:98:c5:82:66:4b:4c:0f:6c:c4:16:59
subgroup order:
  00:8c:f8:36:42:a7:09:a0:97:b4:47:99:76:40:12:
  9d:a2:99:b1:a4:7d:1e:b3:75:0b:a3:08:b0:fe:64:f5:fb:d3
```

On rappelle que le format PEM est un format binaire (DER) en base64. On peut changer le format du fichier par la commande

```
$ openssl dhparam -in dh_param.pem -outform DER -out dh_param.der
```

et retrouver les différentes composantes par la commande `$ xxd dh_param.der` :

```
00000000: 3082 022c 0282 0101 0087 a8e6 1db4 b666  0...,.....f
00000010: 3cff bbd1 9c65 1959 998c eef6 0866 0dd0  <....e.Y....f..
...
00000100: cadb 094a e91e 1a15 9702 8201 003f b32c  ...J.....?.,
00000110: 9b73 134d 0b2e 7750 6660 edbd 484c a7b1  .s.M..wPf'..HL..
...
00000200: cfef 98c5 8266 4b4c 0f6c c416 5902 2100  ....fKL.l..Y.!.
00000210: 8cf8 3642 a709 a097 b447 9976 4012 9da2  ..6B.....G.v@...
00000220: 99b1 a47d 1eb3 750b a308 b0fe 64f5 fbd3  ...}.u.....d...
```

On rappelle que la première série de données de chaque ligne est un compteur d'octet proposé par `xxd`, en hexadécimal. Ainsi le `0x10` = 16ème octet (en partant de zéro) est `3c` et le 17ème est `ff`. Par ailleurs la fin de chaque ligne est la traduction ASCII des données de la ligne (un point signifie non affichable).

Les 4 premiers octets 3082 022c signifient que ce qui suit est une suite de 0x22c = 556 octets. Les quatre suivants 0282 0101 signifient que ce qui suit est un entier de 0x101 = 257 octets, c'est-à dire l'entier premier  $p$  qui commence donc au 8eme octet (en partant de zéro) et se termine donc au 0x109 = 265eme octet (vérifier dans le résultat affiché par `xxd`). Entre le 266eme (0x10a) et le 269eme (0x10d) octet, on a de nouveau quatre octets 02 8201 00 qui annonce que ce qui suit est un entier de 0x100 = 256 octets, c'est-à-dire l'élément  $g$  qui commence au 270eme octet (0x10e) et termine au 524eme octet (0x20c) octet (vérifier dans le résultat affiché par `xxd`). Enfin entre le 525eme octet (0x20d) et le 526eme octet (0x20e) on a deux octets 0221 qui annonce un entier de 0x21 = 33 octets qui est l'ordre du sous groupe généré par  $g$  qui commence au 527eme octet (0x20f) et se termine à la fin du fichier.

Récupérer les trois paramètres  $p$ ,  $g$  et  $N$  à l'aide des lignes suivantes :

```
f = open("dh_param.der", 'rb')
dh_param = f.read()
f.close()
```

Dans ce cas la variable `dh_param` contient la suite d'octet contenu dans le fichier et dans ce cas, par exemple, l'entier  $p$  pourra être obtenu par la ligne

```
p = int.from_bytes(dh_param[8:265], byteorder='big').
```

Les entiers  $g$  et  $N$  s'obtiennent de la même manière. Appeler ensuite la méthode `testDiffieHellman` en utilisant les paramètres que vous venez de récupérer.

Bravo, vous venez de réinventer la roue (ou plutôt le parseur). Il est très important de pouvoir analyser un fichier `pem` ou `der` en cas de besoin, mais vous n'êtes pas obligé de récupérer ces données de cette manière à chaque fois. Dans la suite, le plus simple est d'écrire ces données en dur dans le code.

Générez les clés d'Alice et de Bob pour réaliser le protocole Diffie-Hellman :

```
$ openssl genpkey -paramfile dh_param.pem -out dhkey_alice.pem
$ openssl genpkey -paramfile dh_param.pem -out dhkey_bob.pem
$ openssl pkey -in dhkey_alice.pem -text
$ openssl pkey -in dhkey_bob.pem -text
$ openssl pkey -in dhkey_alice.pem -pubout -out dhpublish_alice.pem
$ openssl pkey -in dhkey_bob.pem -pubout -out dhpublish_bob.pem
$ openssl pkey -pubin -in dhpublish_alice.pem -text
$ openssl pkey -pubin -in dhpublish_bob.pem -text
$ openssl pkeyutl -derive -inkey dhkey_alice.pem -peerkey dhpublish_bob.pem -out dhkey1.bin
$ openssl pkeyutl -derive -inkey dhkey_bob.pem -peerkey dhpublish_alice.pem -out dhkey2.bin
```

Récupérer dans le code python les différentes clés d'Alice et Bob de la manière de votre choix. Vérifier bien les valeurs que vous avez récupéré (il se peut que les clés soient de tailles légèrement différentes entre Alice et Bob). Vérifier que celles-ci sont cohérentes par un appel à la méthode `DiffieHellman`.

### Lab 3 : Cryptographie sur la courbe elliptique P-256

La courbe elliptique sur laquelle on va travailler est la courbe P-256 définie par l'équation de Weierstrass  $Y^2 = X^3 + AX + B$  sur  $\mathbb{Z}_p$  où l'entier premier  $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ , et les éléments  $A$  et  $B$  de  $\mathbb{Z}_p$  sont  $A = -3$  et

$B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b$ .

Le groupe des points de cette courbe est d'ordre

$N = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551$ .

On prend comme élément générateur du groupe, le point  $G = [G_x, G_y]$  où

$G_x = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296$ ,

$G_y = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5$ .

L'élément neutre du groupe est le point à l'infini représenté par la liste  $[0,0]$ .

Ajouter deux attributs  $A$  et  $B$  à la classe **Groupe** qui vont définir la courbe elliptique (modifier le constructeur). Remarque : dans le cas des groupes  $\mathbb{Z}_p$  et  $\mathbb{Z}_p^\times$ , ces attributs auront la valeur **None**.

Modifier la méthode **loi** en implémentant la loi de groupe sur une courbe elliptique quand  $l = \text{'courbe-P-256'}$  (pour les calculs d'inverse modulaire, instancier un objet de classe **Groupe** sur  $\mathbb{Z}_p^\times$  et appeler **squareAndMultiply**. Vous pouvez aussi appeler **loi** sur cet objet (le faire juste une fois par exemple)).

Appeler ensuite la fonction **testDiffieHellman** sur la courbe P-256.

Générer les clés d'Alice et de Bob pour réaliser le protocole Diffie-Hellman :

```
$ openssl ecparam -out ecdhkeyAlice.pem -name prime256v1 -genkey
$ openssl ecparam -out ecdhkeyBob.pem -name prime256v1 -genkey
$ openssl ec -in ecdhkeyAlice.pem -text -noout
$ openssl ec -in ecdhkeyBob.pem -text -noout
$ openssl ec -in ecdhkeyAlice.pem -pubout -out ecdhpubkeyAlice.pem
$ openssl ec -in ecdhkeyBob.pem -pubout -out ecdhpubkeyBob.pem
$ openssl pkeyutl -derive -inkey ecdhkeyAlice.pem -peerkey ecdhpubkeyBob.pem
-out ecdhkey1.bin
$ openssl pkeyutl -derive -inkey ecdhkeyBob.pem -peerkey ecdhpubkeyAlice.pem
-out ecdhkey2.bin
```

Le format d'un point  $(P_x, P_y)$  de la courbe est de la forme  $04 \parallel P_x \parallel P_y$ , où  $P_x$  et  $P_y$  font la même taille (32 octets ici). Vérifier en python que la clé publique d'Alice est bien un point de la courbe, et qu'elle correspond à la multiplication de la clé privée et de  $G$  par un appel à **squareAndMultiply**. Vérifier que la méthode **squareAndMultiply** permet de retrouver l'inverse d'un point. Récupérer les clés générées avec OpenSSL et appeler **DiffieHellman** (qu'il faudra probablement modifier en fonction de  $l$  car ici on ne partage que la coordonnée en x).

Implementer deux nouvelles méthodes de la classe **Crypto**, appelées **ecdsa\_sign** et **ecdsa\_verif**, pour pouvoir signer un message et vérifier la signature. Dans le premier cas, il y aura deux arguments : le message et la clé privée et dans le second cas, il y aura trois arguments : le message, la clé publique et la signature. Signer le fichier **Labs.pdf** (attention à la gestion de la fonction de hachage) avec OpenSSL et python, puis vérifier dans les deux cas la signature avec **ecdsa\_verif**. Remarque : les calculs d'inverse modulaire, réalisés mod  $N$  (premier), sont à faire avec une nouvelle instance de **Groupe**.

## Lab 4 : Cryptographie sur la courbe Curve25519

La courbe elliptique sur laquelle on va travailler est la courbe Curve25519 définie par l'équation de Weierstrass  $Y^2 = X^3 + AX^2 + X$  sur  $\mathbb{Z}_p$  où l'entier premier  $p = 2^{255} - 19$  et  $A = 486662$ . Le groupe des points de cette courbe est d'ordre  $N = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$ . On prend comme élément générateur du groupe, le point  $G = [G_x, G_y]$  où  $G_x = 9$  et  $G_y = 0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9$ . L'élément neutre du groupe est le point à l'infini représenté par la liste  $[0,1]$ .

Modifier la méthode `loi` en implémentant la loi de groupe sur cette courbe quand  $l = \text{'Curve25519'}$  (instancier encore un objet de classe `Groupe` sur  $\mathbb{Z}_p^\times$  pour l'inverse modulaire). Appeler la fonction `testDiffieHellman` sur la courbe.

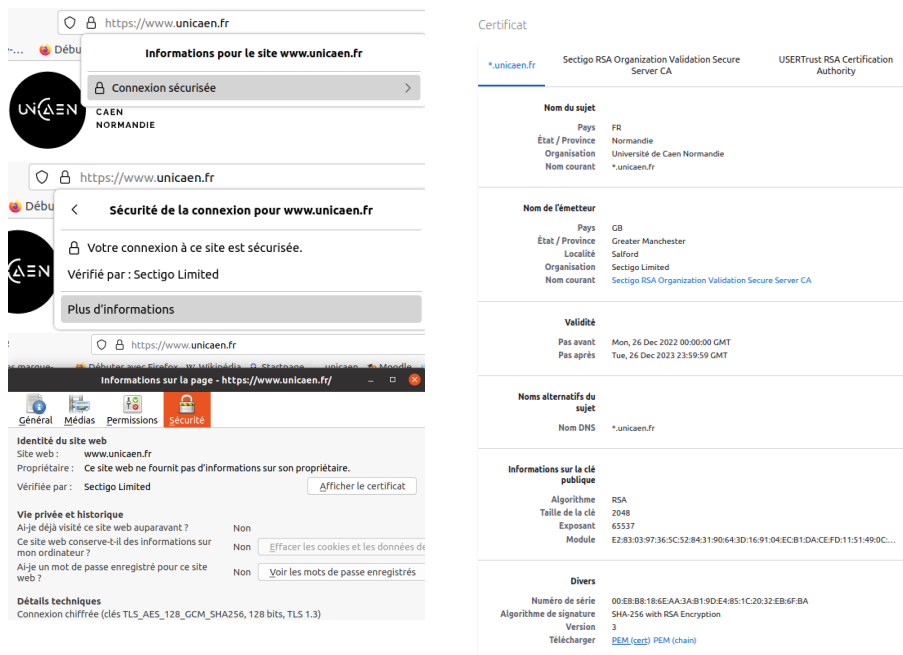
Générer les clés d'Alice et de Bob pour réaliser le protocole Diffie-Hellman :

```
$ openssl genpkey -algorithm x25519 -out keyAlicex25519.pem
$ openssl genpkey -algorithm x25519 -out keyBobx25519.pem
$ openssl pkey -in keyAlicex25519.pem -text
$ openssl pkey -in keyBobx25519.pem -text
$ openssl pkey -in keyAlicex25519.pem -pubout -out pubkeyAlicex25519.pem
$ openssl pkey -in keyBobx25519.pem -pubout -out pubkeyBobx25519.pem
$ openssl pkey -pubin -in pubkeyAlicex25519.pem -text -noout
$ openssl pkey -pubin -in pubkeyBobx25519.pem -text -noout
$ openssl pkeyutl -derive -inkey keyAlicex25519.pem -peerkey pubkeyBobx25519.pem
  -out X25519key1.bin
$ openssl pkeyutl -derive -inkey keyBobx25519.pem -peerkey pubkeyAlicex25519.pem
  -out X25519key2.bin
```

Vérifier en python que la clé publique d'Alice est bien un point de la courbe, et qu'elle correspond à la multiplication de la clé privée et de  $G$  par un appel à `squareAndMultiply` (utiliser la fonction `reverse_bytes` comme dans le cours). Vérifier que `squareAndMultiply` permet aussi de retrouver l'inverse d'un point. Récupérer les clés générées avec OpenSSL et appeler la méthode `DiffieHellman`. Celle-ci devra probablement être modifiée en fonction de l'attribut  $l$  car ici on ne partage que la coordonnée en  $x$  et on utilise la fonction `reverse_bytes` : Soient  $a$  et  $b$  les clés privées obtenues dans les fichiers ci-dessus. On note  $s(x) = \text{reverse\_bytes\_25519}(x) \& ((1 \ll 255) - 8) \mid (1 \ll 254)$ . Soient  $A = aG$  et  $B = bG$ , alors les clés publiques sont `reverse_bytes_25519(A[0])` et `reverse_bytes_25519(B[0])`. Finalement, la clé partagée est : `reverse_bytes_25519((aB)[0]) = reverse_bytes_25519((bA)[0])`.

## Lab 5 : Certificats

Aller sur le site de l'université de Caen. Sous Firefox, cliquer sur le cadenas, puis sur *Connexion sécurisée, Plus d'information, Afficher le certificat*. Vous avez alors accès aux informations contenues dans le certificat de l'unicaen, ainsi que sur l'autorité de certification (la première ligne est constituée de trois onglets, chacun affichant un certificat). Télécharger le certificat (PEM(cert) en bleu).



The image shows a Firefox browser window with the address bar at <https://www.unicaen.fr>. The address bar shows a lock icon and the text "Connexion sécurisée". Below the address bar, there is a section titled "Sécurité de la connexion pour www.unicaen.fr" which states "Votre connexion à ce site est sécurisée." and "Vérifié par : Sectigo Limited". A button "Plus d'informations" is visible. Below this, there is a section titled "Informations sur la page - https://www.unicaen.fr/" with tabs for "Général", "Médias", "Permissions", and "Sécurité". The "Sécurité" tab is selected, showing details about the site's identity, privacy, and technical details. The technical details section shows "Connexion chiffrée (clés TLS\_AES\_128\_GCM\_SHA256, 128 bits, TLS 1.3)". To the right of the browser window, there is a "Certificat" section showing the details of the SSL certificate. The certificate is issued by "Sectigo RSA Organization Validation Secure Server CA" and is for "www.unicaen.fr". The certificate details include the subject, issuer, validity, and public key information.

Certificat	
<a href="#">*unicaen.fr</a>	Sectigo RSA Organization Validation Secure Server CA
USERTrust RSA Certification Authority	
<b>Nom du sujet</b>	
Pays	FR
État / Province	Normandie
Organisation	Université de Caen Normandie
Nom courant	*.unicaen.fr
<b>Nom de l'émetteur</b>	
Pays	GB
État / Province	Greater Manchester
Localité	Salford
Organisation	Sectigo Limited
Nom courant	Sectigo RSA Organization Validation Secure Server CA
<b>Validité</b>	
Pas avant	Mon, 26 Dec 2022 00:00:00 GMT
Pas après	Tue, 26 Dec 2023 23:59:59 GMT
<b>Noms alternatifs du sujet</b>	
Nom DNS	*.unicaen.fr
<b>Informations sur la clé publique</b>	
Algorithme	RSA
Taille de la clé	2048
Exposant	65537
Module	E2:83:03:97:36:5C:52:84:31:90:64:3D:16:91:94:EC:B1:DA:CE:FD:11:51:49:0C:...
<b>Divers</b>	
Numéro de série	00:EB:B8:18:6E:AA:3A:B1:9D:E4:85:1C:20:32:EB:6F:BA
Algorithme de signature	SHA-256 with RSA Encryption
Versión	3
Télécharger	<a href="#">PEM (cert)</a> <a href="#">PEM (chain)</a>

Afficher le contenu du certificat :

```
$ openssl x509 -in unicaen-fr.pem -text -noout
```

```
Signature Algorithm: sha256WithRSAEncryption
...
Modulus:
 00:e2:83:03:97:36:5c:52:84:31:90:64:3d:16:91:
...
Exponent: 65537 (0x10001)
...
Signature Algorithm: sha256WithRSAEncryption
 93:a3:15:d8:87:2a:3a:67:6d:08:88:04:92:2c:15:f0:93:95:
...
20:e0:34:a3
```

On y trouve notamment la clé publique RSA de l'université (modulo e283... et exposant public = 65537). A la fin on trouve une signature RSA : 93a3...34e3. Bien sur, pour vérifier cette signature il faut utiliser la clé publique de l'autorité de certification qui a signé le certificat et non celle qui se trouve dans ce certificat. Votre objectif est de vérifier en python cette signature.



On rappelle que pour vérifier une signature RSA  $s$ , on calcule  $s^e \bmod M$  où  $e$  est l'exposant public et  $M$  le modulo. Le résultat doit être le message signé  $m$  (plus précisément l'empreinte de  $m$ , sous un certain encodage).

La signature du certificat se trouve à la fin du fichier, et fait  $0x101 = 257$  octets, ce qui correspond à la taille du modulo. En particulier, on voit qu'elle commence à l'octet  $0x5d9$  (vu sa taille, récupérez là comme au début du lab2) :

```
$ openssl x509 -in unicaen-fr.pem -outform DER -out unicaen-fr.der
$ xxd unicaen-fr.der
00000000: 3082 06d5 3082 05bd a003 0201 0202 1100
...
000005d0: 010b 0500 0382 0101 0093 a315 d887 2a3a
000005e0: 676d 0888 0492 2c15 f093 9597 94fe dd0b
...
000006d0: 5e56 e444 2b20 e034 a3
```

Pour récupérer l'empreinte du certificat (sans la signature), on voit d'après la première ligne précédente, que le fichier `unicaen-fr.der` fait  $0x6d5 + 4 = 1753$  octets (ce que confirme la commande `wc unicaen-fr.der`) et surtout que le certificat (sans la signature) fait  $0x5bd + 4 = 1473$  octets, situé entre le 4ième octet et l'octet  $0x5c5$ . Ainsi l'empreinte du certificat se calcule par la commande

```
$ cat unicaen-fr.der | tail -c +5 | head -c 1473 | openssl dgst
(stdin)= 3bbd74b5d9a6dac2485786bc417b5e474ddf38552a363e0485ae4818b00a65bf
```

On retrouve aussi cette empreinte directement en python par les lignes :

```
from hashlib import sha256
f = open("./unicaen-fr.der", 'rb')
cert = f.read()
print(sha256(cert[4:1477]).hexdigest())
```

Il reste à obtenir la clé publique de vérification (plus précisément le modulo car  $e = 0x10001$ ). Ainsi, après avoir récupéré le certificat de l'autorité de certification et l'avoir transformé au format DER :

```
$ openssl x509 -in unicaen-fr_certification.pem -text -noout
Modulus:
    00:9c:93:02:46:45:4a:52:48:92:fc:57:8d:f9:2d:
    ...
    1c:db
```

```
$ xxd unicaen-fr_certification.der | grep 9c93
00000190: 0100 9c93 0246 454a 5248 92fc 578d f92d
```

On en déduit que celui-ci commence à l'octet  $0x192$  et qu'il fait  $0x100 = 256$  octets. On le récupère donc par une ligne python du type

```
N = int.from_bytes(certificat[0x192:0x292], byteorder='big')
```

Pour le calcul de  $s^e \bmod M$ , privilégiez cette fois la fonction `pow` plutôt qu'un appel à `squareAndMultiply` sur  $\mathbb{Z}_M^\times$  car vous ne connaissez pas  $\phi(M)$ . Vous devez trouver l'empreinte avec un encodage conforme à PKCS 1.5 pour RSA :

```
0x1fff ... fff[Prefix]3bbd74b5d9a6dac2485786bc417b5e474ddf38552a363e0485ae4818b00a65bf
```

Comme l'empreinte  $h$  de cette donnée fait 32 octets, on récupère les 32 octets de poids faible de  $h$  avec le masque suivant `h & ((1 << 256) - 1)`.

Vous êtes en forme : vérifiez aussi en python la signature du certificat de Wikipedia (obtenu sur <https://fr.wikipedia.org>). Attention, il y a un peu de travail.