TP de Cryptographie: Signatures Numériques avec RSA

I - Signatures numériques

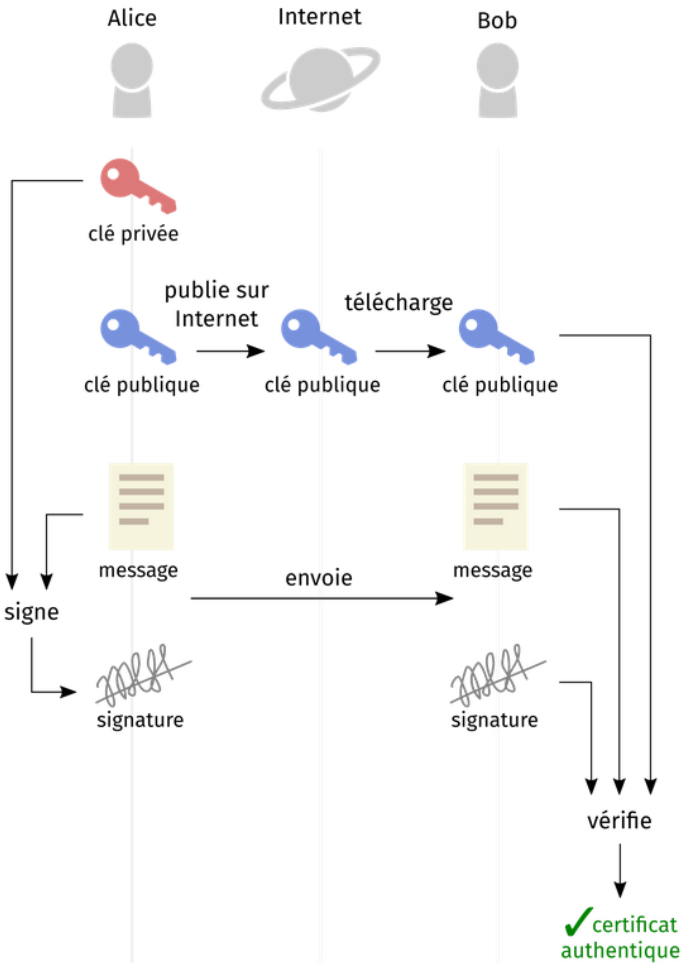
Une signature numérique est un petit morceau de données qui garantit l'authenticité d'un message. Plus exactement, une signature prouve que la personne ayant rédigé ce message est en possession d'une certaine clé secrète.

Concrètement, si Alice veut envoyer un message M à Bob et veut que Bob puisse être sûr de recevoir le bon message, Alice peut faire la chose suivante :

- créer une clé de signature K
- s'assurer que personne à part elle et Bob ne connaisse la clé
- créer une signature S à partir du message M et de la clé K
- envoyer le message M avec sa signature S à Bob

Bob peut vérifier que la signature a été créée pour le message M et avec la clé K. Si Bob est certain que personne à part Alice et lui ne connaissent la clé K, alors il peut être certain que le message M a bien été envoyé à Alice.

La méthode de signature numérique RSA, que l'on va étudier dans ce TP, est dite à clé publique (on peut aussi parler de méthode asymétrique). Cela veut dire que la clé permettant de créer des signatures n'est pas la même que celle permettant de vérifier des signatures. Dans notre exemple, cela veut dire que Alice peut rendre sa clé de vérification publique (tout le monde peut la voir), pour que tout le monde puisse la télécharger et vérifier que les messages qu'ils reçoivent censés venir d'Alice n'ont pas été écrits par quelqu'un d'autre. La clé de signature par contre, elle, doit rester secrète : quelqu'un qui réussirait à voler cette clé serait capable de signer des messages au nom d'Alice sans son accord !



II - Premiers pas avec RSA

1) Signature

Dans la méthode de signature RSA on ne manipule que des nombres, donc pour l'instant nos « messages » ne seront que des nombres. On verra plus tard comment faire avec de « vrais » messages.
Ouvrez votre éditeur Python préféré et créez une variable M contenant le nombre 42 :

```
M = 42
```

M sera notre « message », que Alice veut signer avec sa clé de signature, appelée aussi clé privée. La clé privée d'Alice se compose de deux nombres D et N :

$$C_{Alice}^{priv} = (D, N)$$

- l'exposant privé, noté D
- et le modulus, noté N

Ajouter dans votre éditeur Python les valeurs suivantes pour D et N :

```
D = 107
N = 187
```

Voici comment Alice crée la signature S du message M avec sa clé :

$$S = M^D \mod N$$

Faites faire ce calcul à Python en écrivant le code suivant :

```
S = pow(M, D, N)
```

Note : on pourrait aussi écrire `S = M**D % N` mais plus tard on utilisera des grands nombres et seule la fonction pow sera assez efficace pour cela.

Vous pouvez afficher la valeur de S. Vous devriez obtenir la valeur 70.

2) Vérification de la signature

Pour pouvoir vérifier les messages signés par Alice, Bob doit avoir téléchargé la clé publique d'Alice. Cette clé publique se compose de deux nombres E et N :

$$C_{Alice}^{pub} = (E, N)$$

- l'exposant public, noté E
- le modulus déjà présent dans la clé privé, noté N

La seule partie véritablement « privée » de la clé privée est D (souvenez-vous, la clé privée est créée à partir de D, mais N est nécessaire à la fois pour la signature et pour la vérification.

Voici l'exposant public d'Alice, ajouter cet exposant dans votre code Python :

```
E = 3
```

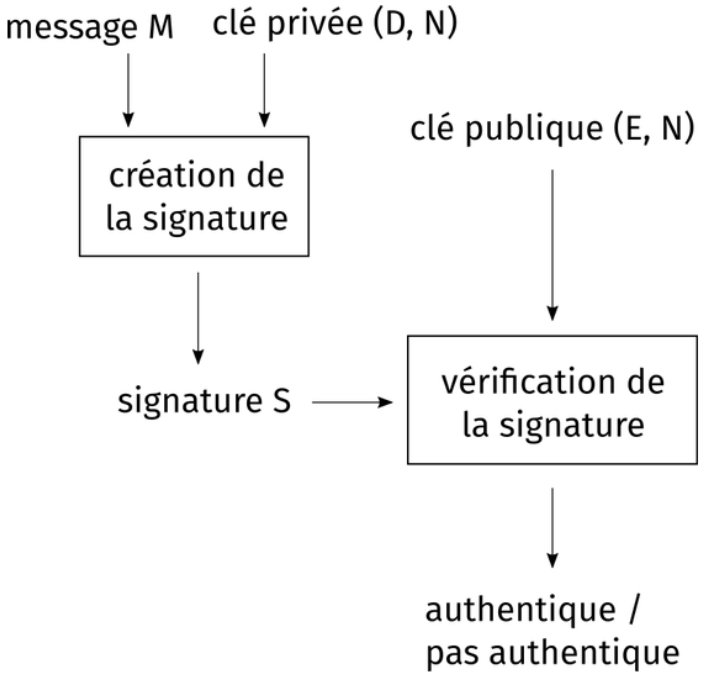
Bob peut maintenant vérifier l'authenticité du message M qu'il a reçu, c'est à dire qu'il peut vérifier que la signature S est bien une signature de M faite par quelqu'un qui connaît la clé privée d'Alice (donc, à priori, par Alice). Il suffit à Bob de calculer ceci :

$$M_2 = S^E \mod N$$

Puis de vérifier que M_2 soit égal au message M.
Modifier votre code Python pour vérifier la signature du message :

```
M_2 = pow(S, E, N)
if M_2 == M:
    print('signature valide: message authentique')
else:
    print('SIGNATURE INVALIDE: MESSAGE NON AUTHENTIQUE !!!')
```

Python devrait écrire `signature valide: message authentique`.



3) Fonctions Python pour la signature et la vérification

Créer un nouveau fichier python et copier le suivant :

```
def signer_rsa(M, D, N):
    S = #[A REMPLACER]
    return S

def signature_rsa_est_valide(M, S, E, N):
    '''renvoie "True" si S est une signature du message M
    faite avec la cle privee correspondant a (E, N);
    sinon, renvoie "False"'''
    M_2 = #[A REMPLACER]#
    if M_2 == M:
        return #[A REMPLACER]#
    else:
        return #[A REMPLACER]#
```

Remplacez les blocs #[A REMPLACER]# par le code nécessaire pour que les fonctions aient l'effet attendu. Ça ne devrait pas être très difficile : on vient de voir comment calculer S et M_2, et la documentation de la fonction `signature_rsa_est_valide` explique ce que doit renvoyer la fonction dans chaque cas.

Maintenant, testez le bon fonctionnement de vos fonctions avec les exemples suivants :

```
signature_rsa_est_valide(42, 70, 3, 187)
signature_rsa_est_valide(45, 70, 3, 187)
```

La première commande devrait renvoyer **True** car ce sont les valeurs de M, S, E et N créées plus haut. La seconde devrait renvoyer **False** car on a modifié le message original, 42, en 45.

On a donc bien réussi à détecter que le message avait été modifié !

Testons aussi notre fonction de signature en signant un message différent.
Mettez un nouveau nombre dans la variable M. Celui que vous voulez, du moment qu'il est compris entre 1 et $N-1$ (rappel : $N = 187$).
Créez aussi une variable $faux\_M$ qui contient un autre nombre pris au hasard entre 1 et $N-1$.
Puis testez les exemples suivants :

```
M = #a completer
faux_M = #a_completer
E = 3
D = 107
N = 187
S = signer_rsa(M, D, N)
signature_rsa_est_valide(M, S, E, N)
signature_rsa_est_valide(faux_M, S, E, N)
```

À nouveau, le premier appel à `signature_rsa_est_valide` devrait renvoyer **True** parce que le message est bien celui qu'on a signé, et le deuxième appel avec un message modifié devrait renvoyer **False**.

III - Génération de clés

Pour l'instant les nombres E, D et N qui constituent les clés de signature et de vérification ont été générés pour nous et nous ont été donné. Maintenant, nous allons voir comment générer nous-même notre paire de clés RSA.
Voici comment générer une paire de clés RSA (c'est à dire une clé publique et sa clé privée correspondante) :

- On choisit un exposant public E.
- On choisit deux nombres premiers P et Q
- Le modulus N de notre paire de clé sera : $N = P \times Q$
- on doit ensuite vérifier que E est premier avec le nombre $\phi(N) = (P-1)(Q-1)$

> En mathématiques, $\phi(n)$ (se lit "phi de n"), la fonction indicatrice d'Euler est une fonction arithmétique de la théorie des nombres, qui à tout entier naturel n non nul associe le nombre d'entiers compris entre 1 et n (inclus) premiers avec n.

- Enfin on trouve l'exposant privé D en calculant l'inverse de E modulo $\phi(N)$, c'est à dire tel que $E \times D \mod \phi(N) = 1$ (ce qui équivaut à ce que $E \times D - 1$ soit un multiple de $\phi(N)$).

On a ainsi trois nombres E, D et N tels que pour tout nombre M compris entre 1 et $N-1$, on ait :

$$(M^D)^E \mod N = M$$

C'est cette propriété qui fait que la signature RSA fonctionne.
Rappelez-vous, on créé la signature de la manière suivante :

$$S = M^D \mod N$$

Rappelez-vous également que l'on vérifie la signature en effectuant l'opération suivante :

$$M_2 = S^E \mod N$$

soit $M_2 = (M^D \mod N)^E \mod N = (M^D)^E \mod N = M$ car les 3 nombres E, D et N ont été créés de tel sorte que $(M^D)^E \mod N = M$.
Nous avons donc ici vérifié la signature !

La démonstration de cette propriété est intéressante, mais pour ne pas trop rallonger le TP on ne la verra pas ici. Elle repose sur le théorème d'Euler (voir « Théorème d'Euler (arithmétique) » sur Wikipedia).

En pratique on ne va pas prendre E au hasard mais prendre une valeur fixe, disons $E = 3$. Il se trouve que ça ne rend pas le système moins sécurisé, donc c'est une pratique courante.

On veut écrire une fonction qui génère une paire de clés RSA suivant la méthode que l'on vient de décrire. Recopier le code suivant dans un nouveau fichier python :

```
# voici des fonctions fournies par le fichier lib_tp_rsa.py
# que l'on importe pour les utiliser dans notre fonction "generer_cle_rsa"
# voir leur documentation dans le fichier lib_tp_rsa.py
from lib_tp_rsa import generer_premier, sont_premiers_entre_eux, inverse_modulo

def generer_cle_rsa():
    "genere une paire de cles RSA"
    E = 3

    nombre_max_tentatives = 5
    for i in range(nombre_max_tentatives):
        P = #[A REMPLACER]
        Q = #[A REMPLACER]

        N = #[A REMPLACER]
        phi_N = #[A REMPLACER]

        if ( P != Q and #[A REMPLACER] ):
            D = #[A REMPLACER]

            # "return" va nous faire sortir de la fonction
            # et donc de la boucle "for" aussi
            return (E, D, N)

    # si on arrive la c'est que toutes les tentatives ont echoue
    # (on n'a jamais atteint le "return")
    # donc on renvoie une Erreur avec "raise Exception()"
    raise Exception('toutes les tentatives ont echoue')
```

Encore une fois vous devez remplacer les blocs #[À REMPLACER].
Pour vous simplifier la tâche, un fichier `lib_tp_rsa.py` contient déjà des fonctions permettant de :

- générer des nombres premiers aléatoires (fonction `generer_premier`)
- vérifier que deux nombres sont premiers entre eux (fonction `sont_premiers_entre_eux`)
- calculer l'inverse d'un nombre modulo un autre nombre (fonction `inverse_modulo`)

Chaque fonction a une documentation détaillée qui devrait vous permettre de comprendre comment l'utiliser.
Vous testerez votre fonction avec l'exemple suivant :

```
(E, D, N) = generer_cle_rsa()
M = 135 # ou n'importe quel autre nombre entre 1 et N-1
faux_M = 28 # un nombre different de M
S = signer_rsa(M, D, N)
signature_rsa_est_valide(M, S, E, N)
signature_rsa_est_valide(faux_M, S, E, N)
```

IV - Signer des ≪ vrais ≫ messages

En pratique, les messages à signer sont de la donnée, pas des nombres. Pour les signer, on utilise un **"encodage"** qui transforme cette donnée en nombre. À nouveau, on vous fourni dans le fichier `lib_tp_rsa.py` la fonction `encoder_msg_en_nombre` qui fait cet encodage.
Copiez et complétez la fonction suivante dans votre fichier python. Vous allez devoir faire appel à la fonction `signer_rsa` que vous avez déjà écrite dans ce même fichier.

```
from lib_tp_rsa import encoder_msg_en_nombre

def signer_message_rsa(msg, D, N):
    M = encoder_msg_en_nombre(msg, N)
    S = #[A REMPLACER]#
    return S
```

Faites de même pour cette fonction qui vérifie un message plutôt qu'un nombre :

```
def signature_message_rsa_est_valide(msg, S, E, N):
    M = #[A REMPLACER]#
    if #[A REMPLACER]#:
        return True
    else:
        return False
```

Testez vos deux fonctions avec les exemples suivants :

```
(E, D, N) = generer_cle_rsa()
msg = "Bien sur que oui !"
faux_msg = "Bien sur que non !"
S = signer_message_rsa(msg, D, N)
signature_message_rsa_est_valide(msg, S, E, N)
signature_message_rsa_est_valide(faux_msg, S, E, N)
```

V - Vérifier l'authenticité de véritables certificats sur Internet !

RSA est toujours très utilisé pour sécuriser les communications sur Internet, et en particulier pour authentifier les certificats des sites web.

Quand on va sur un site web, l'adresse du site commence soit par « http » ou par « https ». HTTPS est la version sécurisée du protocole HTTP.

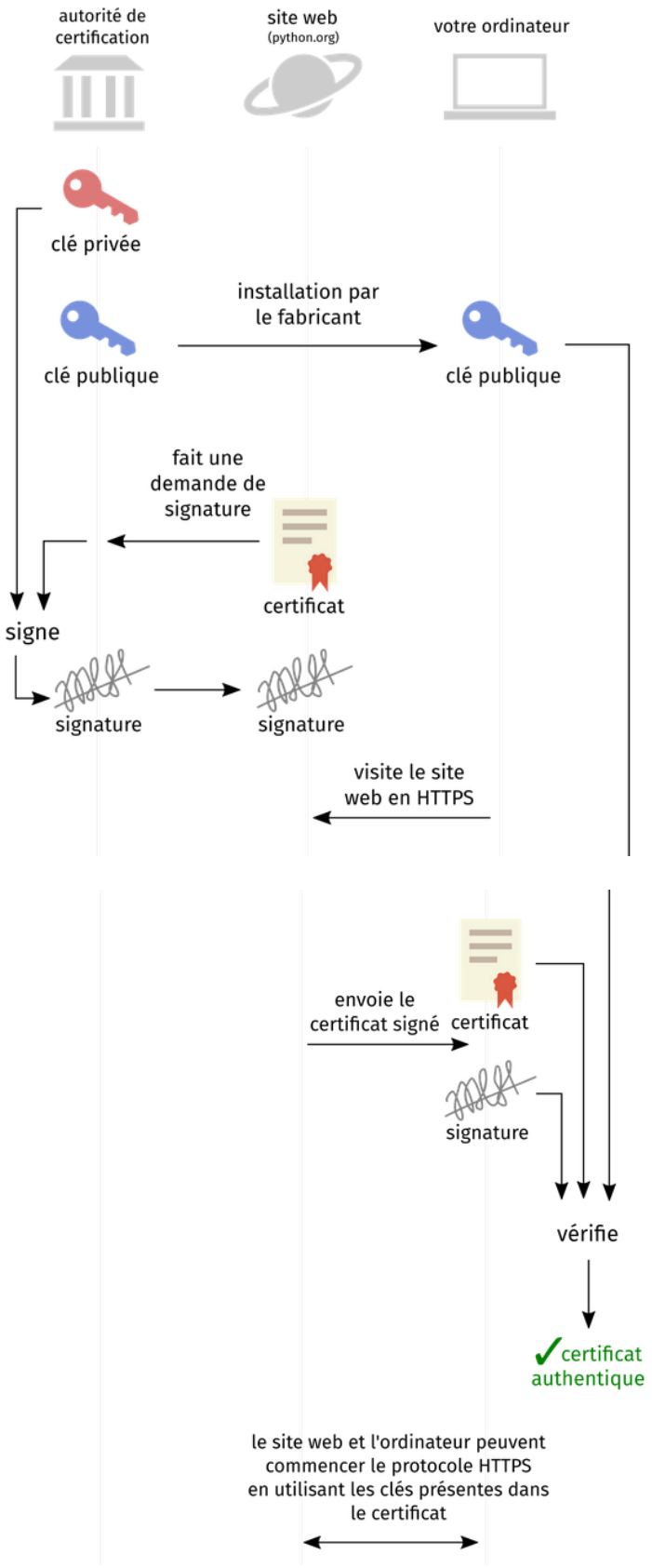
Quand votre ordinateur envoie une requête à un site web en utilisant le protocole HTTPS, le site web commence par envoyer son certificat. Ce certificat contient principalement la clé publique du site web, que votre ordinateur devra utiliser pour vérifier l'authenticité des pages web reçues de la part de ce site web.

Mais il y a un problème : comment être sûr que le certificat que l'on reçoit soit lui-même authentique ? Si un adversaire a la possibilité de modifier les messages entre le site web et nous, il lui suffit de modifier ce premier message et de remplacer la clé publique par la sienne dans le certificat. Cela s'apelle l'**attaque de l'homme du milieu**.

La solution est la suivante : votre ordinateur possède déjà les clés publiques d'un certain nombre d'organisations appelées « **autorités de certification** ». La plupart du temps ces clés ont été installées par le fabricant de votre ordinateur avant de vous le livrer. L'administrateur d'un site web peut alors demander à l'une de ces autorités de lui signer son certificat, et il vous envoie cette signature en même temps que le certificat.

Résumons :

- le site web vous envoie son certificat avec sa clé publique
- le certificat est signé par une autorité dont vous avez déjà la clé publique
- donc le certificat est authentique
- donc vous êtes sûr d'avoir la véritable clé publique du site web
- donc vous pouvez utiliser cette clé pour vérifier les pages web reçues par ce site web



Place à la pratique !

Le fichier lib_tp_rsa.py vous fourni la fonction **recuperer_certificat** qui prend en paramètre le nom de domaine d'un site web et renvoie son certificat (qui est téléchargé via Internet, exactement comme le ferai votre navigateur web) ainsi que le certificat de l'autorité de certification qui a signé ce certificat-là. Voici comment l'appeler et stocker les deux certificats dans des variables séparées :

```
from lib_tp_rsa import recuperer_certificat

cert_site, cert_autorite = recuperer_certificat("python.org")
```

Quelques suggestions de noms de domaines qui devraient fonctionner :

- python.org
- google.com
- ubuntu.com

Les certificats que vous récupérez avec cette fonction sont de vrai certificats, les nombres utilisés sont très grands pour des raisons de sécurité, et les certificats stockés dans les variables `cert_site` et `cert_autorite` ont une foule de paramètres, fonctions et autres que l'on ne va pas utiliser mais que l'on a laissé pour vous mettre en condition réelles. Si vous êtes curieux/se vous pouvez exécuter les commandes suivantes :

- `cert_site.issuer` renvoie un objet représentant l'autorité de certification qui a signé le certificat du site. L'objet est à un format un peu spécial qui n'est pas évident à manipuler ou à lire, mais on peut deviner la signification de certaines valeurs.
- `cert_site.signature_algorithm_oid` précise l'algorithme qui a été utilise pour créer la signature.
  Dans ce TP on ne va que manipuler des signatures faites avec l'algorithme **sha256WithRSAEncryption**, dans lequel on reconnaît le mot « RSA ».
- `cert_autorite.public_key().public_numbers()` contient les nombres E et N de la clé publique de l'autorité. Dans beaucoup de cas le nombre E sera le même, typiquement 65537, on a déjà dit que c'était une pratique courante. Remarquez la taille impressionnante du nombre N !

  La raison à cela est que RSA n'est sécurisé que si N est suffisamment grand pour qu'aucun ordinateur ne puisse retrouver les nombres premiers P et Q dont il est issu.
  Cette opération s'appelle la « factorisation » de N. Or il y a eu de très importants progrès dans les algorithmes de factorisation, donc on doit utiliser des moduli de taille de plus en plus grande, ce qui pose un problème de performance.

On va maintenant vérifier l'authenticité du certificat du site internet en utilisant la clé publique contenue dans le certificat de l'autorité de certification.

Copier le code suivant à la fin de votre fichier python et remplacez l'unique bloc #[A REMPLACER] :

```
def verifier_certificat_site(cert_site, cert_autorite):
    # la version "a signer" du certificat (notre "message")
    # ("TBS" est pour "To Be Signed", "A signer" en anglais)
    msg = cert_site.tbs_certificate_bytes

    # La signature du certificat du site
    S = cert_site.signature
    S = int.from_bytes(S, byteorder='big')

    # La cle publique de l'autorite que l'on va stocker dans deux variables N et E
    N = cert_autorite.public_key().public_numbers().n
    E = cert_autorite.public_key().public_numbers().e

    # A vous de jouer !
    # vous avez deja code la fonction qu'il faut utiliser ici,
    # il n'y a qu'a l'appeler avec les bons parametres !

    if #[A REMPLACER]# :
        return True
    else:
        return False
```

Vous pouvez alors vérifier l'authenticité de certificat que vous venez de télécharger :

```
if verifier_certificat_site(cert_site, cert_autorite):
    print('Certificat valide.')
else:
    print('CERTIFICAT INVALIDE !!!')
```