UNIVERSITE DE CAEN
IUT Caen
Département Réseaux et Télécommunications

TP cryptographie R408 – TP 2 – RSA
***

Exercise 1 Crible Erathostène
Créez la fonction *crible_Erathotene*:

- Elle prend en paramètre un entier n
- Créer une liste de booléens indiquant si les indices correspondants sont des nombres premiers
- Identifier les multiples de chaque nombre en commençant pas 2
- Elle renvoie la liste des nombre premier trouvé

Exercise 2 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. Créez la variables M valant 42, ce sera notre message.

M sera notre "message", que Alice veut signer avec sa clef de signature, appelée aussi clef privée. La clef privée d'Alice se compose de deux nombres:

- l'exposant privé, noté D
- le modulo, noté N

Créez les variables D et N valant respectivement 107 et 187.
Alice crée la signature S du message M avec sa clef : $S = M^D mod N$

Faites faire ce calcul à Python en exécutant la commande suivante: S = pow(M, D, N)
Note: On utilisera de préférence $S = M \times D\%N$ pour les grand nombres, pow ne sera plus assez efficace pour cela.

Exercise 3 Vérification de la Signature
Pour pouvoir vérifier les messages signés par Alice, Bob doit avoir téléchargé la clef publique d'Alice. Cette clef publique est constituée d'un exposant public noté E et du modulo déjà présent dans la clef privée, N. La seule partie véritablement "privée" de la clef privée est D (souvenez-vous, la clef privée est (D,N), mais N est nécessaire à la fois pour la signature et pour la vérification.

L'exposant public d'Alice en une variable noté E valant 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 clef 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.

Rédiger dans une fonction*main*le code pour tester la validité de la signature du message.

Exercise 4 Signature et Vérification

- Créez la fonction *signer_rsa* prenant en paramètre le message, l'exposant public et le modulo et calculant la signature du message.
- Créez la fonction *signature_rsa_est_valide* prenant en paramètre le message, la signature l'exposant public et le modulo. Elle renvoie " True " si S est une signature du message M faite avec la clef privée correspondant à (E, N) sinon, elle renvoie "False".

Testez le bon fonctionnement de vos fonctions avec:

- M = 42, S = 70, E = 3, N = 187 la fonction devrait renvoyer true
- M = 45, S = 70, E = 3, N = 187 la fonction devrait renvoyer false

Testez aussi le fonctionnement de la fonction signature en choisissant un M différent compris entre 1 et N-1. Créez une variable faux_M qui contient un autre nombre entre 1 et N-1.

Exercise 5 Génération de clefs

Pour l'instant les nombres E, D et N qui constituent les clefs 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 clefs RSA.

Voici comment générer une paire de clefs RSA (c'est à dire une clef publique et sa clef privée correspondante):

- On choisit un exposant public E
- On choisit deux nombres premiers P et Q
- Le modulo N de notre paire de clef sera: $N = P \times Q$
- On doit ensuite vérifier que E est premier avec le nombre :
  ϕ (N)=(P-1)×(Q-1)ϕ(N)=(P-1)×(Q-1)
  - ϕ(N) se lit "phi de N" et représente la "fonction indicatrice d'Euler"
  - Deux nombres sont dit "premiers entre eux" si leur seul diviseur commun est 1
  - Ce test est nécessaire pour permettre l'étape d'après ; pour ne pas compliquer les choses on ne vas pas expliquer pourquoi.

- Enfin on trouve l'exposant privé D en calculant l'inverse de E modulo ϕ(N), c'est à dire tel que E× D mod ϕ(N)=1 (ce qui équivaut à ce que E× D-1 soit un multiple de ϕ(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 :
$S^E mod N = (M^D mod N)^E mod N = (M^D)^E mod N = M$

Ce qui correspond à ce que l'on vérifiait pour la vérification de 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.
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.

Créer une fonction $generer\_clef\_rsa$.

- Elle ne prend pas de paramètre
- Définissez E = 3
- Trouver un couple P et Q valide ne se fait pas du premier coup, en général avec un maximum de 25 essais on est capable de trouver un couple valide.
- Pour chaque tentative
  - Générer deux nombre premier P et Q avec la fonction *crible_Erathotene*, calculez n et ϕ(N). Prenez 100 comme maximum de P et Q.
  - Si P et Q sont différents et qu'ils sont premiers (utilisez la fonction PGCD développée dans le TP précédent).
  - Calculez D à l'aide de la fonction $inverse\_modulo(a, m)$.
  - Si D n'est pas vide la fonction retourne E, D et N.

  Vous pouvez testez la fonction dans votre*main* :
  - Générez les valeurs E, D et N
  - Définissez un message M = 135 et $faux\_M = 28$
  - Signez le message
  - Testez la validité de la signature du message et du faux message

Exercise 6 Signer des " vrais " messages

En pratique, les messages à signer sont de la donnée, pas des nombres. L'exercice suivant est très similaire à ce qui a été fait précédemment à la différence prêt que l'on cherchera à traiter des chaînes de caractères plutôt que des nombre.
Créez une fonction *encoder_msg_en_nombre* :

- Elle prend en paramètre une chaîne de caractère
- Elle encode chaque lettre avec l'indice du caractère.
- Elle retourne l'entier constitué de l'ensemble des nombres concaténés.

Créez une fonction *signer_message_rsa* :

- Elle prend en paramètre une chaîne de caractère, l'entier exposant secret D, et la clef N
- Utilisez la fonction *encoder_msg_en_nombre* qui transforme la chaîne de caractère en nombre.
- Elle fait appel à la fonction *signer_rsa* pour signer le message.
- Elle retourne le message signé

Créez une fonction *signature_message_rsa_est_valide* qui vérifie la validité de la fonction:

- Elle prend en paramètre une chaîne de caractère, son équivalent signé, l'entier exposant publique E, et la clef N
- Il est tout d'abord nécessaire de récupérer la version encodé du message avec *encoder_msg_en_nombre*
- Elle fait appel à la fonction *signer_message_rsa* pour déchiffrer le message signé à l'aide de l'exposant public.
- S'il y a correspondance entre le message d'origine encodé et le message déchiffré alors elle renvoie True, sinon False

Vous pouvez tester votre fonction dans votre*main* :

- Générez les clefs RSA.
- Définissez d'abord des message cours un message M = "AB" et $faux\_M$ = "B"
- Signez le message
- Testez la validité de la signature du message et du faux message. Vous vous apercevrez que parfois le message qui devrait être valide est noté comme faux, cela provient de la taille de N qui n'est pas suffisante face au message transposé en entier. Modifiez la fonction *crible_Erathotene* pour que vous puissiez encoder des messages plus long

(entre 10 et 20 caractères) à coup sûr.

Explication : Dans la vraie vie, on privilégiera l'utilisation de fonction de hashage qui associe, à une donnée de taille non bornée, une donnée de taille fixe.

Exercise 9 Casser des clefs RSA

Dans cette section bonus, pour comprendre pourquoi la signature RSA est sécurisée on va essayer justement de la casser.

On se met dans le rôle d'un attaquant qui veut pouvoir signer des messages au nom d'Alice sans l'autorisation d'Alice.

Voici notre stratégie: Alice a publié sur son site web sa clef publique constituée des nombres E et N. On cherche à calculer l'exposant secret D de Alice à partir de ces nombres. Si on y arrive, on sera alors capable de créer des signatures au nom d'Alice en utilisant D et N.

Pour rester concret, voici la clef publique dont on essaie de retrouver la clef privée:

- E = 3
- N = 3399130201

Cette clef a été générée tout simplement en faisant appel à la fonction $generer\_cle\_rsa()$ que vous avez du créer pendant le TP, donc vous pouvez aussi en générer une vous-même sans regarder la valeur de D.

Donc, on veut retrouver la valeur de D correspondant aux nombres précédents. Regardez plus haut dans le TP comment la valeur de D était calculée: c'est l'inverse de E modulo ϕ(N).

C'est là que nos problèmes commencent en tant qu'attaquants : on connaît le nombre N, mais pas ϕ(N). Quand on devait générer des clefs, on avait calculé ϕ(N) en faisant (P-1)×(Q-1) où P et Q sont les facteurs de N, mais les nombres P et Q ne sont pas rendus publiques et ne sont pas connus par l'attaquant (et la raison devrait maintenant être évidente: donner P et Q revient à donner D).

On a déjà vu plus haut que cette fonction ϕ s'appelle la "fonction indicatrice d'Euler". Elle correspond à la quantité de nombres entre 1 et N qui sont premiers avec N. On a déjà vu que deux nombres sont premiers entre eux (on peut aussi dire qu'ils sont "co-premiers") quand leur seul diviseur commun est le nombre 1.

Du coup, est-ce qu'on ne peut pas calculer ϕ(N) tout simplement à partir de N, juste en comptant les nombres premiers avec N un par un ? La réponse est oui, on peut. On pourrait faire notre attaque à partir de cette observation, mais on va utilisez un point de cours qui permet de prendre une autre approche qui est plus efficace.

Souvenez-vous, si on trouve les nombres premiers P et Q tels que N=P×Q, alors on a ϕ(N) (car ϕ(N)=(P-1)×(Q-1), ce qui nous donne D, et on a réussi notre attaque. Une autre approche consiste donc à trouver les diviseurs de N. Or comment trouve-t-on les diviseurs

d'un nombre N ? C'est quelque chose que vous devriez déjà savoir: on teste chaque nombre pour voir s'il divise N, sauf qu'on n'a pas besoin de tester tous les nombres jusqu'à N-1. Il y a une astuce : on peut s'arrêter à ... $\sqrt{N}$. Cette astuce fait une sacrée différence: notre N est de plusieurs milliards, alors que sa racine carrée est de "seulement" quelques dizaines de milliers !

- Créer une fonction $factoriser$ qui prend en paramètre un entier N
- vous utiliserez la bibliothèque math pour accéder au calul de la racine carré.
- Pour chaque nombre i entre 2 et $\sqrt{N}$:
  - Vérifier si le reste de la division de N par i est égal à zéro
  - Si c'est le cas, on a trouvé P et Q qui sont égaux à i et $\frac{N}{i}$
- Elle retourne P et Q

À partir de là, ré-utilisez votre code du TP pour re-calculer D maintenant que vous avez P et Q. Vous devriez obtenir la valeur suivante pour D = 2266009003.

La signature RSA est-elle si sécurisée ? La réponse est : RSA est sécurisé quand on utilise des clefs assez grandes. Quand on génère des clefs RSA avec des nombres P et Q suffisamment grand (donc N est grand aussi), les opérations que doivent effectuer les personnes désirant échanger (signature et vérification) sont un peu plus longues à effectuer. Mais les opérations nécessaires pour casser une clef RSA (en gros, la factorisation de N) demandent, elles, beaucoup plus de temps.

Exercise 10 Prise de mesure

- Créez la fonction $get\_int\_max\_from\_n\_bit$
  - Elle prend en entrée un entier nb_bit
  - Elle calcul l'entier maximum que l'on peut composer avec nb_bit en additionnant les puissances de 2
  - Elle retourne l'entier maximum
- Créez une fonction $get_experimental\_time()$
  - Elle prend en paramètre un entier $taille\_clef\_max$
  - Pour chaque taille max de clef faite appel à $get\_int\_max\_from\_n\_bit$ et à la fonction $crible\_Erathotene$ pour générer deux nombre premier P et Q, calculez N puis factorisez N.
  - Utilisez la bibliothèque **time** pour récupérer les temps d'exécution. Pour cela encadrer l'appel de la fonction factoriser avec des "flag" avec la fonction *time()* de la bibliothèque **time**.

  - Récupérez le temps de calcul en soustrayant le flag 1 du flag 2 dans un tableau pour chaque itération de la clef.
  - La fonction retourne le tableau de temps

- Dans votre *main* faite appel à la fonction $get_experimental\_time()$ avec un ordre de grandeur avec un maximum de taille de clef de 24 bit dans un premier temps puis montez à 26 voir 28 (attendez vous à des temps assez long).

- Utilisez la bibliothèque *scatter* de **matplotlib** pour mettre en forme votre résultat. Votre graphe doit comporter :

  - Une légende
  - Un titre
  - Des noms sur les axes et les grandeurs manipulées (le temps en seconde et le nombre de bit maximum de la clef en nombre de bit)

  Vous rendrez votre graphique dans fichier texte comportant votre nom ("*nom.doc*").

  la taille de clef recommandée aujourd'hui pour RSA est d'avoir un N de 3072 bits (donc P et Q d'environ 1536 bits chacun).

Rendu : Vous rendrez dans une archive avec votre nom et prénom, les fichiers python développé dans le TP que vous aurez commenté et le document contenant votre graphe.