Projet Réseaux et Systèmes, RSA .
2020-2021

Faculté des Sciences et Techniques
Master 1 - Maths. CRYPTIS

Protocole de communication basé TCP sécurisé par RSA

A l'attention de :
M. BROS Maxime

Rédigé par :
PIARD A.
JACQUET R.

Table des matières

Introduction

Le cryptosystème RSA doit son nom aux initiales de ses 3 créateurs, à savoir Rivest (Ron), Shamir (Adi) et Adleman (Leonard). C'est un algorithme de cryptographie asymétrique inventé en 1977. Ce cryptosystème date par conséquent de plus de 40 ans mais est encore utilisé dans certains cas.

A l'époque il était très efficace car les ordinateurs étaient bien moins puissants qu'aujourd'hui, donc des tailles de clés de 256 bits étaient suffisantes pour chiffrer. Aujourd'hui, des tailles de clés de 1024 bits sont nécessaires de manière à ce que le cassage des clefs pour assurer un déchiffrement soit très coûteux, principalement en temps. D'autres méthodes de chiffrement comme l'AES, Advanced Encryption Standard, sont bien plus efficaces aujourd'hui. Mais l'AES est un chiffrement symétrique.

Il est donc fréquent d'utiliser RSA pour échanger des clefs de chiffrement symétriques (pour AES par exemple) de manière sécurisée, puis de s'échanger des informations à travers ce chiffrement symétrique.

1 Fonctions et problèmes rencontrés

1.1 La fonction creationNombreTaille(n)

Le paramètre n est un entier en base 10 qui indiquera que le nombre créé doit être composé de n chiffres.
Etant donné que nous cherchons des nombres premiers, dans cette fonction seuls des nombres impairs sont créés. C'est la fonction suivante qui déterminera si un nombre est premier ou non.

1.2 La fonction premier(n)

Cette fonction, grâce à la commande "openssl prime " va nous dire si le paramètre n, qui est un nombre, est premier ou non. Si ce nombre n'est pas premier, le but va être de le modifier tout en utilisant le plus possible les chiffres qui le composent. Comme expliqué dans le sujet.
C'est une expression régulière qui va récupérer l'information de si le nombre est ou non premier. Cette expression régulière va rechercher "is not prime" dans la sortie de la commande *openssl prime n*. Ici, rechercher "not" est suffisant. Si l'expression régulière ne match pas, cela veut dire que le nombre est premier donc la fonction va renvoyer ce nombre. Si l'expression régulière match, cela veut dire que le nombre n'est pas premier et qu'il faut le changer puis tester sa primalité.
A ce moment là on le modifie en gardant le plus possible sa forme précédente. On avait ici soulevé un problème.

Problème : Si le nombre testé, qui est non premier, est par exemple de cette forme : 50421, c'est à dire que son deuxième chiffre en partant de la gauche, est un 0, alors suite à la modification de ce nombre, on obtiendra $0421X$, où $X \in \{1, 3, 7, 9\}$. C'est à dire $421X$. Or, ce nombre est composé de 4 chiffres. Dans le cas général, de $n - 1$ chiffres. Donc il n'est plus de la taille souhaitée.

Solution : On a rajouté une condition lors de la modification de n. Si le deuxième chiffre est un 0, on en choisit au hasard un nouveau, compris entre 1 et 9. Ainsi, le nombre voulu sera composé de n chiffres. On test alors la primalité de ce nouveau nombre, de la même manière que la première fois et on répète cette opération tant qu'on ne tombe pas sur un nombre premier.

De plus, lors de la modification du nombre, s'il n'est pas premier, il faut changer l'avant dernier chiffre car celui-ci sera égal au dernier chiffre du précédent nombre. Il sera donc égal soit à 1, soit à 3, soit à 7 ou soit à 9. Or, Les chiffres qui composent le nombre, hormis le premier et le dernier chiffre, doivent être compris entre 0 et 9. Donc on remplacera celui-ci par un chiffre compris entre 0 et 9.

1.3 Les fonctions egcd(a, b) et modinv(a, m)

Ces deux fonctions sont liées. La fonction egcd renvoie le pgcd des nombres a et b ainsi que les coefficient de Bézout de a et b. Cette fonction correspond donc à l'algorithme d'Euclide étendu.
Quant à elle, la fonction modinv(a,m), renommée en inverseModulo(a,m) dans notre programme, fait appel à la fonction egcd et renvoie l'inverse modulaire d'un nombre. On utilisera donc ces fonctions pour calculer $d = e^{-1}(mod(\phi(n)))$ avec $\phi(n) = (p-1) \times (q-1)$ et $n = p \times q$.

1.4 La fonction lpowmod(x, y, n)

Cette fonction a été renommée powmod(x,y,n) dans notre programme.
Comme indiqué dans le sujet, cette fonction permet de calculer $x^y \ mod(n)$ bien plus rapidement qu'avec l'opération d'exponentiation ** de python, suivi de la réduction modulo n du résultat obtenu.
Cette fonction sera utilisée pour chiffrer et déchiffrer par RSA. En effet, l'opération de chiffrement (respectivement de déchiffrement) pour RSA est $m^e \ mod(n)$ (respectivement $c^d \ mod(n)$).
Etant sujets à pratiquer RSA avec de grands nombres premiers, l'implémentation de cette fonction est primordiale pour optimiser la vitesse des calculs.

1.5 La fonction chiffrementRSA(plaintext, keyClient)

C'est cette fonction qui s'occupera de chiffrer le message **plaintext** par la méthode RSA. Nous avons ajouté certaines choses à cette fonction. Premièrement, nous voulons chiffrer par blocs de 3 caractères UTF-8, ce qui entraînera quelques lignes de code supplémentaires.
Pour chiffrer, on traduit chaque caractère UTF-8 du message reçu en son ordre dans la table UTF-8 avec la commande *ord(X)*, où X est le caractère dont on souhaite l'ordre. Chaque ordre sera stocké sous forme de chaîne de caractère dans une cellule de liste. de plus, comme nous voulons coder par blocs de 3 caractères, il faut s'assurer que la liste des ordres est bien de taille un multiple de 3. C'est à dire que son nombre de cellules est un multiple de 3. Si non, on y ajoute l'ordre d'un élément qu'on aura choisi au préalable, ici '爨', jusqu'à ce que la taille de la liste soit un multiple de 3.
Maintenant, tous les caractères UTF-8 n'ayant pas la même taille, (peu importe la base choisie), nous avons choisi de faire du padding à chaque ordre de caractère, de manière à ce que chacun d'eux soient composés de 6 chiffres. On concatènera le caractère '4' à gauche de la chaîne de caractères représentant l'entier, jusqu'à ce que cette chaîne soit de taille 6.
Il ne reste plus qu'à concaténer trois par trois les éléments de la liste et de stocker ces nouvelles chaînes de caractères dans une cellule, chacune, d'une nouvelle liste. Ensuite, on chiffre les éléments de cette nouvelle liste un par un et on crée une dernière chaîne de caractères à partir de la concaténation des blocs chiffrés, suivi du caractère |, de manière à ce que les blocs soient visibles à l'oeil nu et que ça aère légèrement l'affichage dans le terminal. Le chiffrement s'effectue alors simplement à l'aide la fonction powmod(int(element), e, int(keyClient)), où element correspond

à un élément de la dernière liste, où $e = 65537$ et où keyClient est le n du client ($n = p \cdot q$). Donc p et q restent secrets.

1.6 La fonction dechiffrementRSA(ciphertext, nc , dc)

C'est cette fonction qui s'occupera du déchiffrement d'un message chiffré **ciphertext** par la méthode RSA présentée précédemment. On crée une liste remplie cellule par cellule des blocs du chiffré, séparés par le caractère | grâce à la fonction split. Il suffit alors de déchiffrer les éléments de la liste un par un et d'ajouter chacun des blocs déchiffrés à une liste initialement vide. Pour déchiffrer blocs par blocs (on appellera element chaque bloc du chiffré), il suffit d'appeler la fonction *powmod*(int(element), int(dc), int(nc). Il faut ensuite diviser chaque chaîne de caracteres de la liste en 3 chaînes de caractères de taille 6 et de supprimer les caractères '4' que l'on avait concaténé pour obtenir des nombres de taille similaire. On retrouve alors une liste de chaine de caractères correspond aux ordres, en UTF-8, des caractères du message clair. Il suffit donc de les récupérer par la fonction *chr()* et de les concaténer pour retrouver le message initial, sans oublier d'enlever les caractères '𠮷' du padding. Il est quasiment impossible de perdre de l'information ici car ce caractère n'est presque plus utilisé au Japon, voir plus du tout. Se référer au premier paragraphe du site : `https://fr.wikipedia.org/wiki/`𠮷.

The next code will be directly imported from a file

2 Client et serveur

2.1 Le serveur

Le serveur, ouvert dans un terminal, attend la connexion d'un client. Le serveur étant le fichier **serveur.py** à éxécuter dans le terminal avec la commande *python3 serveur.py*. L'utilisateur du terminal entrera un message dans le serveur pour que celui-ci soit chiffré par la méthode RSA grâce à la fonction *chiffrementRSA* détaillée plus haut. Pour cela, il aura besoin de la clé publique du client qui lui est envoyée après que le messager ait écrit son message dans le terminal, et avant que s'initie le chiffrement, évidemment.
La clef est récupérée ainsi :

```
key_client = ligne.decode("utf-8").split("|")[0]
```

La fonction split("|") permet de découper une chaine de caractères en plusieurs chaînes de caractères dès que le symbole | est rencontré. Ces chaînes sont stockées dans une liste, chaque chaîne correspond à un élément de la liste et on récupère donc la clef en demandant le contenu de la première case de cette liste, c'est à dire celle contenue à l'indice 0.
Une fois le chiffrement effectué, il ne reste plus qu'à envoyer le chiffré au client sous forme de bytes.

2.2 Le client

Le client, ouvert dans un terminal, se connecte au serveur qui est ouvert au préalable et attend une connexion. Le client étant le fichier **client.py** à éxécuter dans le terminal avec la commande *python3 client.py*. La partie déchiffrement se fait chez le client. Celui-ci intercepte le message chiffré envoyé par le serveur et l'affiche dans le terminal. Le client a alors la possibilité de demander le déchiffrement ou non. S'il répond Oui, alors le déchiffrement du message s'affichera. S'il répond autre chose que Oui, le message sera perdu. Il n'y aura plus la possibilité de le déchiffrer.
Le client a la possibilité de ne pas déchiffrer le message reçu tout de suite. Le serveur peut envoyer un second message, et le client attendre avant de les déchiffrer, même si le serveur arrête la connexion. Le client n'aura qu'à répondre Oui aux deux demandes de déchiffrement lui apparaissant à l'écran.
Si le client ne répond pas aux deux demandes de déchiffrement, le serveur ne pourra pas renvoyer de messages. Il devra attendre qu'au moins une réponse ait été apportée aux demandes de déchiffrement.

3 Jeu d'essais

4 Conclusion