Projet RSA

Théo Bonnet et Thomas Lavaur

1 Les fonctions

✈ La fonction test_primalite :

Cette fonction teste sur le terminal si un nombre est premier, elle récupère la chaîne de caractères correspondant à la réponse du terminal grâce au subprocess qui nous est donné dans le projet. Puis à l'aide des expressions régulières elle vérifie si la chaîne "not" se trouve dans la réponse et renvoie False si oui et True si non.

✈ La fonction premiere_generation_nombre :

Cette fonction génère un nombre en base 10 de taille choisie en entrée avec la méthode donnée dans le projet.

✈ La fonction generation_nombre :

Cette fonction génère un autre nombre potentiellement premier à partir de l'ancien suivant le protocole du projet en décalant vers la gauche toutes les décimales de notre nombre. Après cela, deux situations se présentent à nous :
- soit le nombre ne comportait pas de 0 en deuxième décimale et dans ce cas, le nombre obtenu fait déjà la taille attendu (le bon nombre de décimales) et on peut directement retirer aléatoirement les deux derniers chiffres.
- soit le nombre comportait un 0 en deuxième décimale et nous avons fait le choix de décaler de plus d'une décimale notre nombre, de retirer les dernières décimales manquantes. Pour cela on compte simplement le nombre de décimales qu'il nous manque et on fait le nombre de tirages nécessaire.

✈ La fonction nombre_premier :

Cette fonction utilise les fonctions précédentes pour créer un nombre premier de la taille choisie en entrée. On génère dans un premier temps le premier nombre avec la fonction premiere_generation_nombre puis tant que celui-ci n'est pas premier on le remplace par celui renvoyé par la fonction generation_nombre. La primalité étant testée avec la fonction test_primalite.

✈ Les fonctions to_int et to_bytes :

Cette fonction transforme des bytes (respectivement un entier) en entier (respectivement en bytes). Pour cela on utilise les méthodes liées aux entiers et aux bytes :
- la fonction to_int utilise (int).from_bytes(ENTRÉE, byteorder='big') qui transforme des bytes directement en entier. L'argument byteorder indique que l'on utilise la méthode Big endian dans l'écriture de nos bytes.
- la fonction to_bytes utilise (ENTRÉE).to_bytes(((ENTRÉE).bit_length() + 7) // 8, byteorder='big') qui transforme un entier en bytes. Le premier argument donne le nombre de bytes qui permettront de stocker l'entier (ici on prend le quotient de la division euclidienne par 8 auquel on ajoute 1 si le reste est non nul) et on indique encore ici que la méthode utilisée est Big endian.

✈ La fonction split_message :

On définit dans un premier temps notre nombre d'octets maximum que l'on peut envoyer (c'est-à-dire que l'entier associé à chaque sous-message ne dépassera pas la clé RSA). Pour cela on prend la partie entière du log binaire de notre clé (qui correspond au nombre de bit maximum) que l'on divise ensuite par 8 pour connaître le nombre d'octets maximum. On transforme ensuite la totalité de notre message en bytes et l'on crée un tableau qui contiendra les valeurs entières en base 10 de chaque sous-message.
Enfin, on découpe le message par tranche d'octets max, chaque partie est transformée en entier avec la fonction to_int puis stockée dans une case du tableau.

✈ Les fonctions données :

Dans le projet, trois fonctions nous sont directement données :
- la fonction pgcd qui renvoie donc le pgcd de deux entiers.
- la fonction modinv qui renvoie l'inverse modulaire d'un nombre.
- la fonction lpowmod qui renvoie la puissance modulaire d'un nombre.

2 Le programme

Dans un premier temps, le programme demande à l'utilisateur le nombre de décimales qu'il souhaite pour sa clé RSA (noté n). Ensuite le programme génère deux nombres premiers (noté p et q) grâce à la fonction nombre_premier tel que $p * q = n$. Puis il calcule également φ(n) afin de déterminer la clé privée (notée d).

A cette étape nous avons donc généré notre clé public (n) et notre clé privée (d), il nous reste donc plus qu'à établir la connexion pour échanger nos clés et communiquer. Pour cela le programme va donc demander à l'utilisateur si il veut lancer la version client ou serveur et l'exécuter.

✈ Partie serveur :

Tout d'abord le programme va lancer un serveur TCP et attendre que le client se connecte. Une fois la connexion établie, le programme attend de recevoir la clé publique du client avant d'envoyer la sienne.

Maintenant le serveur peut créer un fork afin d'échanger avec le client. Le processus fils va permettre de recevoir les messages et le processus parent d'envoyer des messages.

Le processus fils va attendre que l'utilisateur saisisse un message (noté message) avant de le découper en tableau de sous-messages (noté m) grâce à la fonction split_message et d'envoyer le nombre de sous-messages que possède m. Pour finir le serveur va chiffrer chaque sous-message avec la fonction lpowmod et la clé du client avant de les envoyer.

Le processus parent va attendre de recevoir le nombre de sous-messages puis commencer à recevoir les sous-messages et les déchiffrer à l'aide de la fonction lpowmod et de ses clés, publique et privée. Ensuite il va concaténer tous les sous-messages pour reformer le message en entier et l'afficher. Si le programme reçoit le message "EXIT" il va fermer le processus son processus fils, envoyer le message "EXIT" chiffré et se fermer. Enfin si le programme reçoit "(EXIT)" il ferme son processus fils et se ferme.

✈ Partie client :

Premièrement le programme va demander à l'utilisateur si il souhaite établir une connexion locale ou par ip, s'il choisit par ip le programme va également demander l'adresse ip où se connecter. Puis le programme va essayer de se connecter, si aucune connexion ne se fait au bout d'un certain temps le programme va automatiquement s'arrêter sinon le client envoie sa clé publique et attend de recevoir celle du serveur. Une fois les clés échangées, le client crée exactement les mêmes processus parent et fils que le serveur qui va lui permettre de communiquer.

3 Problèmes rencontrés et choix d'implémentations

✈ La génération de nombre premier :

Lors de la génération de nombres premiers p et q, nous avons rencontré plusieurs difficultés :

La première étant de savoir comment déterminer la longueur des nombres premiers à générer pour trouver un produit de longueur donné. En effet, si on choisit une taille t pour notre clé publique n, on ne peut pas prendre un nombre très grand et un très petit, comme on ne peut pas prendre deux nombres de taille $\frac{t}{2}$ (sinon l'encodage serait sensible aux attaques car p ou q serait soit proche de n, soit proche de $\sqrt{n}$).
Nous avons donc fait le choix de générer dans un premier temps p de taille $\lfloor \frac{t}{2} \rfloor - 2$. Puis, dans un second temps nous générons un nouveau nombre premier q tant que le produit $p * q$ n'est pas de taille t. Pour accélérer ce processus, nous générons q tantôt de taille $\lfloor \frac{t}{2} \rfloor + 2$, tantôt de taille $\lfloor \frac{t}{2} \rfloor + 3$.

La seconde porte sur la génération elle-même. L'apparition de 0 sur les premières décimales après un shift à gauche comme indiqué dans la méthode à suivre, nous amène à une taille de nombre qui n'est plus celle demandée.
Pour palier à ce problème nous avions dans un premier temps simplement répété l'opération de shift à gauche. Cependant, comme l'utilisation de la fonction rand est coûteuse d'entropie et qu'elle nécessite d'être utilisée deux fois dans chaque shift, nous avons fait le choix de ne faire qu'un seul "grand shift" qui décalerait le nombre de fois nécessaire pour avoir un chiffre différent de 0 en tête, puis on retire aléatoirement les décimales manquantes suivant le même processus : n'importe quel chiffre pour les décimales centrales et un chiffre parmi $\{1, 3, 7, 9\}$ pour la dernière.

✈ La transformation d'une chaîne de caractères en nombre :

Dans un premier temps, nous avions fait le choix de simplement concaténer la valeur de chaque octet en base 10 qui code un caractère en utf-8. Chaque octet prenant des valeurs entre 0 et 255 et un caractère pouvant s'écrire sur maximum 4 octets, il était représenté sur un nombre d'au maximum 12 décimales. Puis, nous concaténions les entiers de chaque caractère entre eux tant que cela ne dépassait pas la taille de la clé publique du destinataire.
Seulement avec cette méthode, nous n'optimisions pas du tout l'usage de la clé. Si chaque caractère était représenté sur 4 octets et que la clé publique était de 100 décimales, nous ne pouvions envoyer que 8 caractères.

Dans un second temps, en ayant une approche binaire, nous avons procédé exactement de la même façon mais tout en base 2. Ainsi, si par exemple nous voulions encodé le caractère Ï, ce caractère est codé sur 2 octets : $[195, 143]$. La première méthode transforme donc Ï en 195143 qui comporte 6 décimales. La deuxième méthode prend 195 et 143 qui valent en binaire respectivement 11000011 et 10001111, on concatène pour avoir 1100001110001111 qui vaut 50063 en base 10 donc codé sur 5 décimales. On gagne ainsi de nombreuses décimales lorsque le message est plus conséquent.

✈ Le découpage du message :

Le problème était de déterminer la longueur de nos sous-messages pour qu'elle ne soit pas plus grande que celle de la clé du destinataire.

Pour commencer, on considérait que chaque caractère était codé sur 4 octets et donc 12 décimales (comme expliqué ci-dessus), donc on divisait la longueur de la clé du destinataire par 12 pour obtenir le nombre de caractères par sous-message. Mais cette procédure faisait perdre beaucoup de place donc nous avons décidé de regarder la taille en bits de la clef du destinataire. Pour cela, on cherche un n tel que $2^n \leqslant clé$; on utilise alors $\lfloor \log_2(clé) \rfloor$, puis le nombre d'octets par sous-message en divisant par 8. Ainsi on obtient une majoration du nombre d'octets que l'on peut encoder avec notre clé RSA. Ce principe n'est pas encore optimal car il ne prend pas en compte la clé directement mais la puissance de deux inférieure. Nous pourrions donc encore améliorer ce point mais nous avons fait le choix de laisser ainsi.

✈ Les données envoyées dans le réseau :

A l'origine, nous envoyions notre message codé donc représenté par un nombre que l'on transformait en string puis en bytes avec bytes(str("message"),"utf-8"). Cela était très coûteux et peu sécurisé car pour envoyer une décimale de notre nombre, nous l'encodions en utf-8 et cette décimale valait alors 1 octet complet! Mais cela nous permettait d'envoyer des séparateurs pour gérer nos différents sous-messages car seul des strings de chiffres circulaient et on pouvait ainsi librement utiliser des lettres pour communiquer des informations non liées au tchat.

Dans la version finale, nous envoyons dans un premier temps le nombre de sous-messages qui vont être envoyés et cela sur 4 octets. Si le nombre tient sur moins de 4 octets, on fait du padding pour forcer l'envoi de 4 octets. Cela nous limite donc à $2^{8*4+1} - 1 = 8\ 589\ 934\ 591$ sous-messages mais cela nous semble largement suffisant (la bible tient sur environ 4 millions de caractères). Pour envoyer le message, on envoie dans un second temps les sous-messages en les transformant directement en bytes avec la méthode .to_bytes liée aux entiers et en précisant la taille de sortie en $\lceil \frac{\log_2(clé)}{8} \rceil$. En effet, une fois codé, notre sous-message ne dépassera pas la taille de la clé et on envoie alors toujours la même taille de sous-message en faisant de nouveau du padding.

✈ Les problèmes du tchat :

Nous avons trouvé deux problèmes :

Le premier survient lorsqu'un message est reçu pendant que nous sommes en train d'écrire, le début de notre réponse se trouve au-dessus du message reçu et la suite en-dessous.

Le deuxième surgit quand on copie/colle du texte avec des sauts de ligne, on a plusieurs prompts qui apparaissent à la fin de la ligne après avoir envoyé le message.
Nous n'avons pas cherché à apporter une solution à ces problèmes car ils ne nous paraissaient pas essentiels.

✈ La fermeture de la communication :

Cette partie nous à poser beaucoup de problèmes car il fallait réfléchir à une méthode permettant de fermer les 4 processus sachant que le client comme le serveur devaient avoir la possibilité de fermer à tout moment le tchat. Ils devaient donc jouer un rôle symétrique. Cependant, le processus fils ne peut communiquer avec son parent et ne sait même pas qu'il existe. Nous avons donc chercher un moyen de fermer proprement tout les processus en évitant un maximum de contraintes pour l'utilisateur. Finalement, lorsque l'utilisateur rentre le mot EXIT, notre programme suit le schéma joint au format GIF. Nous avons donc muni la partie réception qui est le processus parent d'une partie envoie de message qu'il n'utilise que lorsqu'il reçoit le mot EXIT. Il envoie alors à son tour une confirmation qui permettra au destinataire de bien se fermer. C'est cette confirmation qui amène une contrainte à l'utilisateur et lui interdit de la saisir lui-même. Dans notre programme, si l'utilisateur choisit d'écrire le mot (EXIT), celui-ci lui sera refuser.

4 Les améliorations introduites

Nous avons ajouter 3 principales améliorations dans notre programme :

-L'ajout de time out si le serveur ne répond pas au bout d'un certain temps, le programme s'arrête et indique une attente trop longue.

-L'embellissement du tchat pour l'utilisateur avec l'utilisation de couleurs sur le terminal et la verbose centrée.

-Une utilisation simple pour l'utilisateur qui n'a qu'un seul programme à exécuter et seulement à suivre les indications.

5 Les améliorations possibles

Les améliorations que nous avons envisagées :

-L'ajout de la possibilité de communiquer en même temps sans mélanger les messages sur le terminal.

-L'optimisation de l'utilisation de la clé jusqu'à ses limites : ne pas se restreindre à la puissance de 2 inférieures mais à la clé entière.

-Éviter le padding, en envoyant par exemple avant la longueur du message une fois chiffré avant.

-Améliorer la fermeture des différents processus pour éviter les messages interdits (comme (EXIT) pour nous).

-Proposer un arrêt propre du programme si l'on désire le stopper avant la mise en place de la connexion.

-Proposer un tchat pour un groupe de plus que 2 personnes (avec plusieurs clients pour un même serveur).

-L'ajout d'un niveau de verbose que pourrait choisir l'utilisateur.

6 Un exemple d'utilisation