Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/is

Introduction à la sécurité

TP 3 : RSA

Dans ce TP :
— Implémenter le cryptosystème RSA.

Exercice 0.

Recommandations.

1. Ce TP est à faire en Python.
2. Faites les questions dans l'ordre et pensez à tester votre code.
3. N'hésitez jamais à rajouter de la sortie de debug pour comprendre tout ce qui se passe.

Exercice 1.

Génération de clef.

1. Le but de cet exercice est d'écrire une fonction `gen_rsa_keypair` qui prend en paramètre une taille de clef exprimée en bits, et qui renvoie une paire de clef publique/privée de cette taille.

   → Commencez par retrouver dans le cours la méthode de génération des clefs RSA. En quoi consiste chacune des deux clefs de la paire ?
2. Ce qu'on appelle la taille de la clef, c'est celle du module. Quand on multiplie deux nombres, la taille du résultat est la somme des tailles des deux nombres.

   → Si on appelle `bits` la taille de la clef demandée, de quelle taille doit-on générer les nombres premiers aléatoires `p` et `q` ?
3. Pour générer un nombre premier aléatoire, vous pouvez utiliser la commande `openssl` ou la fonction `getPrime` de la bibliothèque Python `Crypto.Util.number`.

   → Dans votre fonction `gen_rsa_keypair`, générez `p`, `q`, `n` et `phi_n`.
4. Pour l'exposant de chiffrement `e` (public), on prend généralement un "petit" nombre premier, soit aléatoirement soit choisi en dur comme par exemple 65537 en s'assurant évidemment qu'il respecte les conditions nécessaires.

   → Quelles sont ces conditions ?
5. → Dans votre fonction, générez `e` et vérifiez qu'il respecte les conditions nécessaires.
6. L'exposant de déchiffrement `d` (privé) se calcule en fonction de valeurs qu'on connaît déjà.

   → Lesquelles, et comment ?
7. Pour calculer un inverse modulaire, vous pouvez utiliser la fonction `inverse` de la bibliothèque `Crypto.Util.number` ou vous pouvez trouver une implémentation de celle-ci sur le site Rosetta Code.

   → Dans votre fonction, générez `d`.
8. → Faites renvoyer à votre fonction la paire de clef publique/privée.

Exercice 2.

Chiffrement et déchiffrement.

1. Soit (A_p, A_s) la paire de clefs d'Alice, et (B_p, B_s) la paire de clefs de Bob.

   (a) → Quelle clef doit utiliser Bob pour chiffrer un message à destination de Alice ?
   (b) → Quelle clef doit utiliser Alice pour déchiffrer un message qui lui est adressé ?
2. → En quelle opération consiste chacune des opérations décrite à la question précédente ?
3. → Implémentez la fonction `rsa` qui prend en paramètre un message (représenté par un entier) et une clef (représentée par une paire comprenant un exposant et un module) et qui fait l'exponentiation modulaire.
4. Pour la suite, vous aurez besoin de la fonction `rsa`, il faut donc qu'elle fonctionne bien.

   → Vérifiez que vous avez correctement écrit votre exponentiation modulaire (indice : n'essayez pas qu'avec des petits nombres…), et corrigez la si besoin.

5. En Python, pour convertir une `str` en `int`, il faut d'abord la convertir en `bytes` avec la méthode `encode` de `str` (on utilise UTF-8) puis convertir le `bytes` en `int` avec la méthode `from_bytes` de `int` (on utilise la convention "big endian").

   Par exemple :

```
>>> msg = int.from_bytes('all cats are beautiful'.encode('utf-8'), 'big')
>>> msg
364504681283170925927024603099703282495116612818998 84
```

   En sens inverse on utilise la méthode `to_bytes` de `int` et la méthode `decode` de `bytes`. Vous pouvez récupérer la taille en bits d'un entier avec la méthode `bit_length` de `int`.

   Par exemple :

```
>>> msg.to_bytes((msg.bit_length() + 7) // 8, 'big').decode('utf-8')
'all cats are beautiful'
```

   → Écrivez les fonctions `rsa_enc` et `rsa_dec` qui permettent de chiffrer et déchiffrer des chaînes de caractères (pensez à vérifier la condition de possibilité de déchiffrement avant de chiffrer !).

6. → Utilisez votre fonction `gen_rsa_keypair` pour générer deux paires de clefs par exemple pour Alice et Bob, puis simulez un échange de messages confidentiels entre les deux en utilisant les fonctions de la question précédente.

Exercice 3.

Signature et vérification.

1. Soit (A_p, A_s) la paire de clefs d'Alice, et (B_p, B_s) la paire de clefs de Bob.

   (a) → Quelle clef doit utiliser Bob pour signer un message ?

   (b) → Quelle clef doit utiliser Alice pour vérifier l'authenticité du message qui prétend être signé par Bob ?

2. Pour signer un message, on signe généralement un condensé (hash) du message, ce qui permet de signer des messages de toutes tailles et de s'assurer au passage de l'intégrité du message.

   → Quelle est la procédure de signature et quelle est la forme du message signé ?

3. La bibliothèque `hashlib` de Python vous permet d'utiliser par exemple l'algorithme de hashage `sha256`. Attention, cette bibliothèque travaille avec des `bytes` (tableaux d'octets).

   → Écrivez une fonction `h` qui prend en paramètre un entier et renvoie l'entier correspondant à son condensé.

4. → Écrivez les fonctions `rsa_sign` et `rsa_verify`.

5. → Avec les paires de clef de l'exercice précédent, simulez un échange de messages authentifiés et intègres entre Alice et Bob en utilisant les fonctions de la question précédente.

Exercice 4.

Vulnérabilités !

Pour cet exercice et le suivant, on travaillera directement sur des entiers (et donc on utilisera directement notre fonction `rsa` de l'exercice 2).

1. La version de RSA qu'on a implémentée ici, qu'on appelle communément *textbook RSA*, souffre de plusieurs problème de sécurité.

   Par exemple, il est possible de *forger* des chiffrés valides à partir de chiffrés existants qu'on aurait interceptés. C'est ce qu'on appelle la malléabilité.

   → Comment ?

2. Un autre soucis est le déterminisme du chiffrement. C'est à dire que si on chiffre deux fois le même message avec la même clef, on obtient deux fois le même chiffré.

   → En quoi est-ce un problème ?

3. Conclusion : c'est *compliqué* d'implémenter correctement des algorithmes cryptographiques. Il vaut mieux éviter de le faire soi-même et plutôt utiliser des bibliothèques déjà existantes, libre, et surtout, largement éprouvées.

Exercice 5.

Bonus : RSA-OAEP version jouet.

1. Pour pallier au problème de malléabilité et de déterminisme, on utilise un "bourrage" ou *padding* pour RSA.

   L'idée est de compléter notre message avec des bits aléatoires jusqu'à une taille donnée de façon à casser le déterminisme et la malléabilité.

   → Lisez attentivement la page Wikipédia de RSA-OAEP : `https://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding`.

2. RSA ne peut chiffrer que des nombres plus petits que son module. Ce n'est donc pas approprié au chiffrement de message qui peuvent faire une longueur arbitraire.

   → Quelle solution utilise-t-on en pratique ?

3. Imaginons qu'on utilise AES-128, c'est à dire un cryptosystème symétrique avec une clef de taille 128. On a donc besoin de RSA seulement pour la signature et pour le chiffrement d'une clef de 128 bits générée aléatoirement pour chaque message.

   → En utilisant la fonction `h` et en supposant que le message `m` fait toujours 128 bits, écrivez les fonction `oaep_pad` et `oaep_unpad` (vous pourrez prendre une taille de 64 bits pour l'aléa r).

4. → Utilisez les fonctions de la question précédente pour écrire les fonctions `rsa_oaep_enc` et `rsa_oaep_dec`.

5. → Générez maintenant une paire de clef par exemple sur 768 bits, et utilisez RSA-OAEP pour vérifier que les vulnérabilités liées aux propriétés de malléabilité et de déterminisme sont bien absentes de cette nouvelle version des algorithmes de chiffrement et déchiffrement.