TD 3: clés publiques

1 RSA: attaque par factorisation

Alice et Bob utilisent RSA. avec $n = 209$. L'affichage de la clé publique d'Alice par la commande `openssl rsa -in cle -text -noout` donne:

```
Public-Key: (6 bits)
modulus: 209 (0xd1)
publicExponent: 17 (0x11)
```

(1) retrouvez la clé privée d'Alice;
(2) avec des clairs dont la valeur numérique est inférieure à $n - 1$, retrouvez le clair correspondant au chiffré 20.

2 Mauvais usage de RSA

En `Python`, choisissez un module supérieur à 256, créez une clé publique et une clé privée puis chiffrez un petit texte.
Pour cela, convertissez chaque caractère en un entier (avec `ord()` et sa fonction réciproque `chr()`), transformez la chaîne de caractères en une liste d'entiers puis chiffrez chaque élément de la liste au moyen de la fonction `pow()`.
Vérifiez que vous êtes capables de déchiffrer puis faites une analyse des fréquences du chiffré.
Qu'en déduisez-vous?

3 RSA: construction d'une clé

On souhaite construire une (petite) biclé RSA compatible avec `openssl` et la librairie `Cryptography` de `Python`. Il faut pour cela

- choisir deux grands entiers premiers p et q (de 20 chiffres décimaux) à partir d'une table;
- effectuer différents calculs (p.e. en utilisant la bibliothèque `sympy` de `Python`):
  - trouver les exposants de chiffrement et de déchiffrement (resp. e et d)
  - calculer les valeurs $e_1 = d \mod (p-1)$, $e_2 = d \mod (q-1)$ et coeff $= q^{-1} \mod p$

On range ces valeurs dans un fichier texte formatté comme ci-dessous pour une valeur de module de 187:

```
asn1=SEQUENCE:rsa_key

[rsa_key]
version=INTEGER:0
modulus=INTEGER:187
pubExp=INTEGER:7
privExp=INTEGER:23
p=INTEGER:17
q=INTEGER:11
e1=INTEGER:7
e2=INTEGER:3
coeff=INTEGER:14
```

A l'aide de ce fichier (`rsaval.txt`), on peut construire une biclé au format DER (`newkey.der`) au moyen de la commande:

```
openssl asn1parse -genconf rsaval.txt -out newkey.der
```

Il faut ensuite convertir cette clé au format PEM pour otenir la biclé `psk.pem` compatible avec `openssl`:

```
openssl rsa -in newkey.der -inform DER -outform PEM -out psk.pem
```

(1) Construire une biclé `openssl` de taille supérieure ou égale à 128 bits comme indiqué ci-dessus.

(2) Extraire la clé publique de la biclé.

(3) Chiffrer un (tout petit) texte en utilisant la commande `rsautl` d'`openssl`.

(4) Déchiffrer le message obtenu précédemment pour vérifier le bon fonctionnement.

(5) Quelle est la plus petite taille de biclé qu'il est possible d'engendrer par la commande `genrsa` d'`openssl`?

(6) Pourquoi a-t-on du construire "à la main" une clé de taille 128 bits et pas une de plus petite taille ?

(7) Quels sont les types de bourrage qu'on peut utiliser avec RSA?

NB: Nous allons à présent utiliser la librairie `Cryptography` de `Python` qui impose un padding différent du padding `None` qu'on a utilisé ici.

4 Utilisation de la clé avec la librairie `Cryptography`

Récupérez d'abord le code source qui génère une biclé, chiffre et déchiffre avec RSA utilisant un padding OAEP.
Avec la documentation officielle, adaptez ce code pour qu'il utilise la clé de 128 bits engendrée à l'exercice précédent. Celle-ci est trop petite pour utiliser le padding OAEP. Il faut utiliser le padding `PKCS1v15`.
Chiffrez puis déchiffrez un petit message au format `bytestream`.

5 Gestion de clés

Soit le protocole suivant:

(1) $A \rightarrow B : K^a \mod p$

(2) $A \leftarrow B : (K^a)^b \mod p$

(3) $A \rightarrow B : (K^{ab})^{a^{-1}} \mod p$

- Trouvez l'information partagée par A et B pour que le protocole fonctionne.
- Quelles sont les informations qui sont conservées secrètes par les parties?
- Décrivez et expliquez le fonctionnement de ce protocole.
- Montrez que le protocole n'assure pas l'authentification des parties et construisez-une attaque. Proposez une amélioration du protocole.