TP4 : Hachés et signatures

Exercice 1 (Collisions *). L'objectif des fonctions de hachages naïves dans cet exercice est de vous permettre de trouver des collisions facilement grâce à un petit programme et un peu de réflexion.

1. Soient les hachés suivants par une certaine fonction de hachage H :

   ```
   H(A) = 65
   H(B) = 66
   H(C) = 67
   H(Z) = 90
   H(a) = 97
   H(z) = 122
   H(AB) = 131
   H(ABC) = 198
   H(CB) = 133
   ```

   Quel est le calcul effectué par cette fonction de hachage H ? Trouvez (au moins) une pré-image de 298 (qui a du sens ou pas).

2. Pour éviter les soucis des anagrammes, on modifie un peu la fonction de hachage et on construit la fonction G suivante, qui produit les hachés suivants :

   ```
   G(A) =  65
   G(B) =  66
   G(C) =  67
   G(D) =  68
   G(AB) =  197
   G(ABC) =  398
   G(ABCD) =  670
   G(BAC) =  397
   ```

   Trouvez le fonctionnement de cette nouvelle fonction de hachage G et deux prénoms d'au plus cinq lettres qui forment une collision. Les personnages célèbres suivants peuvent vous aider : Federer, Hitchcock, Knuth, Obispo, Polo, Popeye, Murphy, Shannon, Smith, Turing, Turner, Tyson, Wilde.

Exercice 2 (Adobe*). Voici un extrait (fictif) de la base de données Adobe qui a (réellement) fuité il y a maintenant quelques années.

Login	Hint	Hash SHA-256
Bart	element 74	068be8be83f9bfafd1545d357fd3cd132f8c659effd11e635a698811b796c880
Bob	numbers	15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225
Carlton	1 to 9	15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225
Homer	metal	068be8be83f9bfafd1545d357fd3cd132f8c659effd11e635a698811b796c880
John	nine	15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225
Lisa	mendeleiev 74	068be8be83f9bfafd1545d357fd3cd132f8c659effd11e635a698811b796c880
March	hard rock	068be8be83f9bfafd1545d357fd3cd132f8c659effd11e635a698811b796c880
William	numbers - 0	15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225

Dans un terminal, la commande `echo -n password | shasum -a 256` ou la commande `echo -n 'password' |openssl sha256` génére le haché du mot `password` avec la fonction de hachage `SHA-256` :
`5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8`
Le programme Python suivant calcule aussi le haché du mot `password` avec la fonction de hachage `SHA-2566`.

```python
import hashlib

mot="password"
sha256=hashlib.sha256(mot.encode('utf-8')).hexdigest()

print(sha256)
```

Listing 1 – Haslib pour SHA-256.

Trouvez les 2 mots de passe qui sont utilisés par ces utilisateurs d'Adobe, prouver vos résulats.

Exercice 3 (Signature RSA naïve *).
Rappel du fonctionnement de la signature RSA naïve :

- Clé publique : $pk = (e, n)$
- Clé privée : $sk = d$
- Signature : σ de m avec sk vaut m^d mod n
- Vérification de $sigma$ avec pk : tester que σ^e mod n est bien égal à m

1. Codez la fonction de signature de RSA. Pour calculer les puissances modulaires rapidement et efficacement, vous souhaiterez peut-être utiliser la commande `pow(a,b,c)`.
2. Avec $pk = (e = 17, n = 3233)$ et $d = 2753$, signez 855. On ne vous demande pas de vérifier que les clés RSA sont correctes.
3. Montrez qu'il est possible d'obtenir une signature de 2197.

Exercice 4 (OpenSSL).
Chiffrement Symétrique

1. Créez un fichier texte `message.txt` avec le message de votre choix. Utilisez la commande suivante pour chiffrer ce message avec `openssl` :

   `openssl enc -aes-256-cbc -pbkdf2 -in message.txt -out ciphersym.bin`

2. Envoyez ce message chiffré à une autre personne avec le mot de passe. La personne peut alors déchiffrer le message avec la commande suivante :

   `openssl enc -d -aes-256-cbc -pbkdf2 -in ciphersym.bin -out plainsym.txt`

Génération de clefs RSA

1. Générez une paire de clef RSA de longueur 4096 avec la commande suivante :

   `openssl genrsa -aes256 -out rsakey.pem 4096`

   La passphrase doit faire entre 4 et 1023 caractères.
2. Extrayez la clé publique de `rsakey.pem` avec la commande suivante :

   `openssl rsa -in rsakey.pem -pubout -out public.pem`
3. Les informations sur le module et les nombres premiers utilisés sont chiffrées avec AES-256 dans le fichier `rsakey.pem`. Utilisez la commande suivante pour trouver les éléments N, p, q, e et d de la clé RSA générée.

   `openssl rsa -in rsakey.pem -noout -text`
4. Utilisez Python pour vérifier que les valeurs sont correctes.

Chiffrement à clef publique

1. En utilisant la commande suivante, chiffrez le même message `message.txt` avec la clef RSA que vous avez générée précédemment :

   `openssl pkeyutl -encrypt -in message.txt -pubin -inkey public.pem -out cipherasym.bin`
2. Comparez les deux chiffrés obtenus (symétrique et asymétrique) et leurs tailles.
3. Déchiffrez le chiffré `cipherasym.bin` avec la commande suivante :

   `openssl pkeyutl -decrypt -in cipherasym.bin -inkey rsakey.pem -out plainasym.txt`
4. Quelle est le nombre maximal de caractères que peut contenir le fichier `message.txt` ? Pour info, un fichier `.txt` commence par un header (invisible) de 12 octets. Vous voudrez peut-être vous aider d'une commande du type `python3 -c 'print("a"*10)' > message2.txt` pour vérifier vos hypothèses.
5. Envoyez votre clé publique à une autre personne et demandez-lui de vous envoyer un message chiffré, puis déchiffrez-le avec votre clé privée.

Signature

1. Pour signer un message, on utilise le chiffrement RSA, avec la commande suivante :

   `openssl pkeyutl -sign -in messageasigner.txt -inkey rsakey.pem -out signedmessageasigner.bin`
2. Quelle est le nombre maximal de caractères que peut contenir le fichier `messageasigner.txt` ?
3. Comment signer des messages plus gros ? La commande suivante peut vous aider :

   `openssl dgst -sha256 -binary message.txt > hmessage.dgst`

4. Pour vérifier la validité d'une signature il faut utiliser la commande suivante :
   `openssl pkeyutl -verify -in messageasigner.txt -pubin -inkey public.pem -sigfile signedmessageasigner.bin`

Exercice 5 (Injection de faute sur la signature RSA CRT).

1. Choisissez un paramètre de sécurité `k` et générez des clés RSA. Vous voudrez peut-être utiliser la fonction `randprime` de la bibliothèque `sympy`, la fonction `randint` de la bibliothèque `random` et la fonction `gcd` de la bibliothèque `math`.
2. On rappelle que la signature RSA naïve d'un message `m` consiste à calculer $\sigma = m^d \mod n$, où d est la clef secrète et $n = p \cdot q$ le module, tandis que la vérification de la signature σ d'un message m consiste à tester si $\sigma^e \mod n$ est égal à m.
   Générez un message `m` au hasard, signez-le et vérifiez-en la signature.
3. Pour la signature RSA CRT (où RCT = Chinese Remainders Theorem), on commence par précalculer en privé $a = q \cdot (q^{-1} \mod p)$ et $b = p \cdot (p^{-1} \mod q)$. Ensuite, la signature du message m est $s = a \cdot s_1 + b \cdot s_2$, où $s_1 = m^d \mod p$ et $s_2 = m^d \mod q$, et la vérification de la signature s d'un message m consiste à tester si $s^e \mod n$ est égal à m.
   Calculez `a` et `b`, générez un message `m` au hasard, signez-le et vérifiez-en la signature.
4. La signature RSA CRT est vulnérable à une attaque par injection de faute, qui permet de trouver un des deux facteurs premiers de n (et donc l'autre, et donc la clef privée...). Pour cela, l'attaquant demande deux fois la signature d'un même message m, et la deuxième fois, il perturbe le calcul de s_1. Il obtient ainsi une signature correcte s et une signature erronée s^* du message m. Or s et s^* ont le même reste modulo q (c'est s_2) et des restes différents modulo p (pour s c'est s_1 et pour s^* c'est une valeur erronée), par conséquent, q divise $s - s^*$, mais p ne le divise pas. L'attaquant calcule alors $PGCD(s - s^*, n)$, qui vaut q. En effet, les seuls diviseurs de n sont 1, p, q et n.
   Générez une valeur erronée aléatoire pour `s1`, et calculez `q`.

Exercice 6 (Algorithme de signature de Schnorr *).
À partir d'un nombre premier p et d'un générateur g publics, Alice se munit d'une clef secrère s_k aléatoire (mais comprise entre 2 et $p - 1$) et publie la clef publique associée $p_k = g^{s_k} \mod p$. Elle souhaite envoyer un message m authentifié à Bob. Le schéma de signature de Schnorr choisit un k aléatoire entre 2 et $p - 1$, puis calcule $r = g^k \mod p$. Soit $e = H(r \parallel m) \mod p$, où $\parallel$ représente la concatenation, et H une fonction de hachage. Soit $s = k - s_k e \mod \mathbf{(p\text{-}1)}$, alors la signature est $\sigma = (s, e)$. La vérification consiste à calculer $r_v = g^s \cdot p_k^e \mod p$, et à vérifier que $H(r_v \parallel m)$ est égal à e.

1. Codez la fonction de génératiron de p_k à partir de p, g et s_k. Choisissez un paramètre de sécurité et générez p et g, puis s_k et p_k. Vous voudrez peut-être utiliser des fonctions des bibliothèques `random` et `sympy`.
2. Codez les fonctions de signature et de vérification. Vous voudrez peut-être utiliser la bibliothèque `hashlib`.
3. Signez le message $m = 42$ et vérifiez la signature.