Hachage et signature électronique avec OpenSSL (Linux)

Partie C : Mise en pratique complète (TP pas à pas)

TP1 : Hachage et détection de modification

1. Créer `doc.txt` contenant ~5 lignes de texte.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ vim doc.txt

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ cat doc.txt
line1
line2
line3
line4
line5
```

2. Calculer SHA256 et SHA512 : `openssl dgst -sha256 -sha512 doc.txt` (ou deux commandes séparées).

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 doc.txt
SHA2-256(doc.txt)= 77165950de0a3f78fa118b964335e26deb3c895a7de50035bb5f8505a19f19cf

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 doc.txt
SHA2-512(doc.txt)= 340368e48e61b51898b605952ed832b927e298426449b1e990a7a7fa9637da5ad9fd230bae133cb8c7196ebc3f1e11ef45510d5872aec1aa0feb29464bb04d05
```

`Taille de sortie de sha256 est 64 caractères en hexa en revanche sha512 est de taille 128 caractères en hexa`

3. Sauvegarder les empreintes dans `doc.txt.sha256` et `doc.txt.sha512`.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 doc.txt >>doc.txt.sha256

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 doc.txt >>doc.txt.sha512
```

4. Modifier `doc.txt` (ajouter espace ou lettre) et vérifier que les empreintes ne correspondent plus.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ echo " this is a new line " >> doc.txt

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 doc.txt
SHA2-256(doc.txt)= d1520947910db9c0576f0810183736cea033e198484582a09e84219f45dc2938

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 doc.txt
SHA2-512(doc.txt)= 28f4a611467af0c4584614e1e0dac24587064554bae57512f63d91529067448702723bdcd47931d0d47abb58be8ee3f552ded52701dac2f5eab80670060bee9c
```

les deux empreintes sont diffèrent soit avec l'algorithme sha256 soit avec sha512 la moindre modification est affectée

Livrable : capture de commandes et explication (1–2 paragraphes).

TP2 : Signature RSA simple

1. Générer `rsa_private.pem` et `rsa_public.pem`.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl genpkey -algorithm RSA -out rsa_private.pem -pkeyopt rsa_keygen_bits:2048
..+...+.++++++++++++++++++++++++++++++++++++++++*...+...++++++++++++++++++++++++++++++++++++++++*........+......+...+......+..
..+..+.+............+...........+....+......+..+............+............+..+...+..........+.........+.....+.+...+..+......
...+..........+...+..+......+...+..............+...+.....+.+..+.......+...+..+...+.......+...............+.+............
..+.........+....+........+...+......+.+...+..............+.+......+......+..+..........+.....+.....+.....+.........+.+..
.+..+......+....+...+..+.........+.........+.+..+.......+..............+...+..............+.+...............+.......+++++
+
.+.........+....+.........+............++++++++++++++++++++++++++++++++++++++++*......+.....+......+...+.+...+.....+.......+.
..+.....+............+.....+.........++++++++++++++++++++++++++++++++++++++++*...+.................+....+......++++++

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl pkey -in rsa_private.pem -pubout -out rsa_public.pem
```

2. Signer `doc.txt` en `doc.sig`.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 -sign rsa_private.pem -out doc.sig doc.txt
```

La signature se fait par la clé privée et prend comme paramètre le nom d'algorithme, la clé privée, le nom de fichier de sortie et le document original

3. Vérifier la signature.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 -verify  rsa_public.pem -signature  doc.sig doc.txt
Verified OK
```

La vérification se fait par la clé publique prend comme paramètre le document signé et le document original

4. Tenter de vérifier `doc.sig` avec une clé publique différente (générer une 2e paire) — expliquer le résultat.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl genpkey -algorithm RSA -out rsa_private2.pem -pkeyopt rsa_keygen_bits:2048
..+.+......+...+.....+.++++++++++++++++++++++++++++++++++++++++*....+.........+.+..++++++++++++++++++++++++++++++++++++++++*..
.....+..............+...+..........+............+......+.............+.+.....+.+........+...+.........++++++
...+..+..+++++++++++++++++++++++++++++++++++++++*...+..++++++++++++++++++++++++++++++++++++++++*........++++++

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl pkey -in rsa_private2.pem -pubout -out rsa_public2.pem

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha512 -verify  rsa_public2.pem -signature  doc.sig doc.txt
Verification failure
40F7211ABA7F0000:error:0200008A:rsa routines:RSA_padding_check_PKCS1_type_1:invalid padding:../crypto/rsa/rsa_pk1.c:79:
40F7211ABA7F0000:error:02000072:rsa routines:rsa_ossl_public_decrypt:padding check failed:../crypto/rsa/rsa_ossl.c:796:
40F7211ABA7F0000:error:1C880004:Provider routines:rsa_verify_directly:RSA lib:../providers/implementations/signature/rsa_sig
.c:1041:
```

la vérification échoue avec une mauvaise clé seul la clé correspondante utilisé pour vérifier cela prouve l'authenticité

Livrable : fichiers `.pem`, `.sig`, log de vérification et explication.

TP3 : ECDSA + comparaison

1. Générer `ec_private.pem` et `ec_public.pem` (prime256v1).

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl ecparam -name prime256v1 -genkey -out ec_private.pem
```

```
└─$ openssl ec -in ec_private.pem -pubout -out ec_public.pem
read EC key
writing EC key
```

2. Signer `doc.txt` → `doc-ecc.sig`.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 -sign ec_private.pem -out doc-ecc.sig  doc.txt
```

3. Vérifier.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 -verify  ec_public.pem -signature  doc-ecc.sig doc.txt
Verified OK
```

4. Mesurer tailles : `stat -c%s doc.sig doc-ecc.sig` et expliquer pourquoi ECDSA a généralement des signatures plus courtes.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ stat -c%s doc.sig ; stat -c%s doc-ecc.sig
256
72
```

ECDSA produit des signatures 3 fois plus petit que RSA

Livrable : comparaison table + courte conclusion.

TP4 : HMAC et intégrité symétrique

1. Choisir clé secrète `MYKEY="tp_secret"`.

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ MYKEY="tp_secret"
```

g

```
┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 -hmac "$MYKEY" -binary doc.txt | xxd -p
e793bc917b5a42c9c8720f4ff62d5ac0e2bea59f1ce56f2080d8fa1dba8f
3974
```

2. Calculer HMAC-SHA256 :
3. `openssl dgst -sha256 -hmac "$MYKEY" -binary doc.txt | xxd -p`
4. Modifier doc et montrer mismatch.

```
3974

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 -hmac "$MYKEY" doc.txt >> doc.txt.hmac

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ echo "modif">> doc.txt

┌──(kali㉿kali)-[~/tp_openssl_hash_sign]
└─$ openssl dgst -sha256 -hmac "$MYKEY" doc.txt
HMAC-SHA2-256(doc.txt)= b5c0819cf88ec8a7bf36735dc845ff81cf9f8a9ef67685c11f2580d612ef5507
```

Avec hmac qui nécessite une clé secrète partagé, utilisé pour l’ authentification
Avec Rsa seule la clé publique est distribué seul le détenteur de la clé privé peut signé
Diffèrent hash

Livrable : commandes + explication sur différences entre HMAC et signature asymétrique (usage, sécurit)