CHIFFREMENT OPENSSL ET CTF

L'objectif de ce TP est d'aborder les points suivants :

I°/ LE PROTOCOLE SSL

Le protocole SSL (**Secure Socket Layer**) a été développé par la société Netscape Communications Corporation pour permettre aux applications client/serveur de **communiquer de façon sécurisée**. TLS (**Transport Layer Security**) est une évolution de SSL réalisée par l'IETF.

SSL est un protocole qui s'intercale entre TCP/IP et les applications qui s'appuient sur TCP. Une session SSL se déroule en deux temps :

- Une phase de poignée de mains (**handshake**) durant laquelle le client et le serveur s'identifient, conviennent du système de chiffrement et d'une clé qu'ils utiliseront par la suite.
- La phase de communication proprement dite durant laquelle les données échangées sont compressées, chiffrées et signées.

L'identification durant la poignée de mains est assurée à l'aide de certificats **X509**.

A. OPENSSL.

`openSSL` est une boîte à outils cryptographiques implémentant les protocoles `SSL` et `TLS` et offre

- Une bibliothèque de programmation en C permettant de réaliser des applications client/serveur sécurisées s'appuyant sur `SSL/TLS`.
- Une commande en ligne (`openssl`) permettant :
  - La création de clés `RSA`, `DSA` (signature) ;
  - La création de certificats `X509` ;
  - Le calcul d'empreintes (`MD5`, `SHA`, `RIPEMD160`, ...) ;
  - Le chiffrement et déchiffrement (`DES`, `IDEA`, `RC2`, `RC4`, `Blowfish`, ...) ;
  - La réalisation de tests de clients et serveurs `SSL/TLS` ;
  - La signature et le chiffrement de courriers (`S/MIME`) ;

L'ensemble des fonctionnalités est disponible via l'aide (`# man openssl`) et la syntaxe générale de la commande est : `# openssl <commande> <options>`

II°/ LE CHIFFREMENT SYMETRIQUE.

La commande `enc` permet de chiffrer/déchiffrer avec openssl :

```
# openssl enc <options>
```

Parmi les options, on indique le système de chiffrement à choisir dans la liste :

`aes-128-cbc`	AES 128 bis in CBC mode
`aes-128-ecb`	AES 128 bis in ECB mode
`aes-192-cbc`	AES 192 bis in CBC mode
`aes-192-ecb`	AES 192 bis in ECB mode
`aes-256-cbc`	AES 256 bis in CBC mode
`aes-256-ecb`	AES 256 bis in ECB mode
`base64`	Base 64
`bf-cbc`	Blowfish in CBC mode
`bf`	Alias for bf-cbc
`bf-cfb`	Blowfish in CFB mode
`bf-ecb`	Blowfish in ECB mode
`bf-ofb`	Blowfish in OFB mode
`cast-cbc`	CAST in CBC mode
`cast`	Alias for cast-cbc
`cast5-cbc`	CAST5 in CBC mode
`cast5-cfb`	CAST5 in CFB mode
`cast5-ecb`	CAST5 in ECB mode
`cast5-ofb`	CAST5 in OFB mode
`des-cbc`	DES in CBC mode
`des`	Alias for des-cbc
`des-cfb`	DES in CBC mode
`des-ofb`	DES in OFB mode
`des-ecb`	DES in ECB mode
`des-ede-cbc`	2 key triple DES EDE in CBC mode
`des-ede`	Alias for des-ede
`des-ede-cfb`	2 key triple DES EDE in CFB mode
`des-ede-ofb`	2 key triple DES EDE in OFB mode
`des-ede3-cbc`	3 key triple DES EDE in CBC mode
`des-ede3`	Alias for des-ede3-cbc
`des3`	Alias for des-ede3-cbc
`des-ede3-cfb`	3 key triple DES EDE CFB mode
`des-ede3-ofb`	3 key triple DES EDE in OFB mode
`desx`	DESX algorithm .
`idea-cbc`	IDEA algorithm in CBC mode
`idea`	same as idea-cbc
`idea-cfb`	IDEA in CFB mode
`idea-ecb`	IDEA in ECB mode
`idea-ofb`	IDEA in OFB mode
`rc2-cbc`	128 bit RC2 in CBC mode
`rc2`	Alias for rc2-cbc
`rc2-cfb`	128 bit RC2 in CBC mode
`rc2-ecb`	128 bit RC2 in CBC mode
`rc2-ofb`	128 bit RC2 in CBC mode
`rc2-64-cbc`	64 bit RC2 in CBC mode
`rc2-40-cbc`	40 bit RC2 in CBC mode
`rc4`	128 bit RC4
`rc4-64`	64 bit RC4
`rc4-40`	40 bit RC4
`rc5-cbc`	RC5 cipher in CBC mode
`rc5`	Alias for rc5-cbc
`rc5-cfb`	RC5 cipher in CBC mode
`rc5-ecb`	RC5 cipher in CBC mode
`rc5-ofb`	RC5 cipher in CBC mode

REMARQUE

`base64` n'est pas un système de chiffrement, mais un codage des fichiers binaires avec 64 caractères ASCII. Ce codage est utilisé en particulier pour la transmission de fichiers binaires par courrier électronique.

B. LAB_1 : CHIFFREMENT AVEC MOT DE PASSE.

1. Créer un fichier message contenant « Bienvenue en BTS Sio ».

2. Chiffrer le fichier `message` avec le système `Blowfish` en mode `CBC`, avec une clé générée par mot de passe, le chiffré étant stocké dans le fichier `message.chiffre`. La clef est `password`.

```
# openssl enc -bf-cbc -in message -out message.chiffre
```

3. Visualiser le contenu de `message.chiffre`.

4. Utiliser l'option `-d` pour déchiffrer le fichier `message.chiffre`.
```
# openssl enc -bf-cbc -d -in message.chiffre -out message.dechiffre
```

5. A l'aide de la commande `diff`, comparer les fichiers `message` et `message.chiffre`, puis `message` et `message.dechiffre`.

Q. Quel message est retourné si l'on tente de déchiffrer un cryptogramme en utilisant un mauvais mot de passe. ?

Q. Le fichier `cryptogram_lab_1_etud` a été chiffré avec le système `AES` en mode `CBC`, la clé de 128 bits ayant été obtenue par mot de passe. Le mot de passe codé en base 64 est `QW50b255Q0NDCg==`.
Aide : Etape de résolution

- Décoder le mot de passe `QW50b255Q0NDCg==`.
- Déchiffrer le `cryptogram_lab_1_etud`.

C. LAB_2 : CHIFFREMENT AVEC CLE EXPLICITE.

Pour chiffrer le fichier `message` avec une clé explicite, il faut utiliser les options `-K` et `-iv`

- `-K` (K majuscule) suivi de la clé exprimée en hexadécimal ;
- `-iv` (iv en minuscules) suivi du vecteur d'initialisation exprimé en hexadécimal.

6. Chiffrer le fichier `message` avec `Blowfish` en mode `CBC` avec un vecteur d'initialisation de 64 bits exprimé par 16 chiffres hexa, et une clé de 128 bits exprimée par 32 chiffres hexa.
```
# openssl enc -bf-cbc -in message -out message.chiffre \
   -iv 0123456789ABCDEF \
   -K 0123456789ABCDEF0123456789ABCDEF
```

II°/ RSA.

D. COMMANDES USUELLES D'OPENSSL.

Les principales commandes sont :

- **Générer une paire de clés RSA**
```
# openssl genrsa -out <fichier> <taille>
```
  - `fichier` est le fichier de sauvegarde de la clé.
  - `taille` est la taille souhaitée (en bits) du modulus.

  Exemple
  Génération d'une paire de clés de 1024 bits, stockée dans le fichier `maCle.pem`. Le fichier obtenu est au format `PEM` (Privacy Enhanced Mail, format en base 64)
```
# openssl genrsa -out maCle.pem 1024
```

- **Visualiser le contenu d'un fichier au format PEM contenant une paire de clés RSA**
```
# openssl rsa -in <fichier> -text -noout
```
  - `text` demande l'affichage décodé de la paire de clés.
  - `noout` supprime la sortie.
  - `pubin` pour afficher la partie publique.

  Exemple
  Affichage du fichier `maCle.pem`.
```
# openssl rsa -in maCle.pem -text -noout
```

- **Chiffrer une paire de clés à l'aide d'une passphrase**
  `# openssl rsa -in <fichier> -<Algo> -out <fichier>`

  - 3 options sont possibles pour l'algorithme de chiffrement symétrique : `-des`, `-des3` et `-idea`.

  Exemple
  Chiffrement en `des3` du fichier `maCle.pem`.
  `# openssl rsa -in maCle.pem -des3 -out maCle.pem`

- **Exportation de la partie publique d'une paire de clés RSA**
  `# openssl rsa -in <fichier_cle_privee> -pubout -out <fichier_cle_publique>`

  - La partie publique peut être communiquée à n'importe qui.
  - `pubout` exporte la partie publique d'une clé.

  Exemple
  Extraction de la partie publique à partir de la clef privée.
  `# openssl rsa -in maCle.pem -pubout -out maClePublique.pem`

- **Chiffrer/déchiffrer des données avec une clé RSA**
  `$ openssl rsautl -encrypt -in <fichier_entree> -inkey <cle> -out <fichier_sortie>`

  - `fichier_entree` est le fichier des données à chiffrer. Le fichier des données ne doit pas avoir une taille excessive (ne doit pas dépasser 116 octets pour une clé de 1024 bits).
  - `cle` est le fichier contenant la clé RSA. Si ce fichier ne contient que la partie publique de la clé, il faut rajouter l'option `-pubin`.
  - `fichier_sortie` est le fichier de données chiffré.

  Exemple
  Pour déchiffrer, on remplace l'option `-encrypt` par `-decrypt`.

E. LAB_3 : ENVOI D'UN MESSAGE DE BOB A ALICE.

7. Générer une paire de clef (privée et publique) pour Bob.
   a) Créer une clef privée de 2048 bits pour Bob.

   b) Chiffrer la clef avec l'option des3.

   c) Exporter la clef publique de Bob.

8. Générer une paire de clef (privée et publique) pour Alice.

9. Chiffrer un message d'amour de Bob pour Alice.
   Bob doit créer un message et le chiffrer afin de l'envoyer à Alice.

Q. Quelle clef avez-vous utilisé pour chiffrer le message ?

Q. Quel est le contenu du message chiffré ?

10. Déchiffrer le fichier de Bob par Alice.
    Alice doit déchiffrer le message précédant reçu de Bob.

11. Afficher le message déchiffré par Alice et comparer le au fichier original écrit par Bob.

F. LAB_4 : ENVOI D'UN MESSAGE DE BOB A ALICE CODE AVEC UN MOT DE PASSE.

Le mot de passe choisi par les amoureux est : `L0v€`

12. Coder le fichier `message` avec le système `Blowfish` en mode `CBC`, avec une clé générée par mot de passe. Le chiffré sera stocké dans le fichier `messageCode.txt`.

13. Chiffrer le message de Bob pour Alice.

14. Déchiffrer le fichier de Bob par Alice.

15. Décoder le message d'Alice.

16. Vérifier que le message original de Bob et le message final d'Alice sont identiques.

G. LAB_5 : SIGNATURE DE FICHIERS

Il n'est possible de signer que de petits documents. Pour signer un gros document, on calcule d'abord une empreinte de ce document via la commande `dgst` :

```
# openssl dgst <hachage> -out <empreinte> <fichier_entree>
```

où hachage est une fonction de hachage. Avec openssl, plusieurs fonctions de hachage sont proposées dont :

- `MD5` (option `-md5`), qui calcule des empreintes de 128 bits,
- `SHA1` (option `-sha1`), qui calcule des empreintes de 160 bits,
- `RIPEMD160` (option `-ripemd160`), qui calcule des empreintes de 160 bits.

17. Créer le fichier `message.txt`.

18. Créer l'empreinte du fichier précédent.
```
# openssl dgst -sha1 -out msg.empreinte msg.txt
```

Signer un document revient à signer son empreinte. Cela peut être effectué avec l'option `-sign` de la commande `rsautl`.

```
# openssl rsautl -sign -in <empreinte> -inkey <cle> -out <signature>
```

19. Signer l'empreinte avec une clef privée chiffrée.
```
openssl rsautl -sign -in msg.empreinte -inkey uneclefpriveechiffree.pem
                                                    -out msg.signe
```

Pour vérifier une signature, on utilise la commnade :

```
# openssl rsautl -verify -in <signature> -pubin -inkey <cle> -out <empreinte>
```

L'option -pubin indique que la clé utilisée pour la vérification est la partie publique de la clé utilisée pour la signature.

20. Vérifier la signature.
```
# openssl rsautl -verify -in msg.signe -pubin -inkey uneclefpublique.pem
                                                    -out msg.empreinte.verif
```

Pour que la signature soit vérifiée, il faut que l'empreinte ainsi produite soit la même que celle que l'on peut calculer.
```
# diff msg.empreinte msg.empreinte.verif
```