Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Sécurité des usages TIC

TP n°2

Manipulation des éléments de sécurité

Présentation d'openSSL

La bibliothèque openSSL est une boîte à outils cryptographiques servant de référence :

- 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, ECDSA, PQC ;
  - la création de certificats X509 ;
  - le calcul d'empreintes (MD5 (collisionnable), SHA256/SHA-3, RIPEMD160, . . .) ;
  - le chiffrement et déchiffrement (AES, ChaCha20-poly1305, . . .) ;
  - la réalisation de tests de clients et serveurs SSL/TLS ;
  - la signature et le chiffrement de courriers (S/MIME).

Pour connaître toutes les fonctionnalités d'openSSL : `man openssl`.
Pour exécuter une commande : `openssl <commande> <options>`

1 – Regardez les possibilités offertes par openSSL :

a. obtenez la liste des algorithmes de chiffrement supportés par openSSL avec la commande :

```
openssl enc -help
```

b. essayez de chiffrer, puis déchiffrer et enfin, de vérifier le chiffrement de la manière suivante :
Pour chiffrer le fichier « toto » avec l'algorithme *AES* en mode CBC, avec une **clé générée par mot de passe**, le document chiffré étant stocké dans le fichier toto.chiffre, on utilise la commande :

```
openssl enc -aes-256-cbc -in toto -out toto.chiffre -pbkdf2 -iter 100000
```

Vous essaierez la commande simplifiée :
`openssl enc -aes-256-cbc -in toto -out toto.chiffre`,
que vous retourne-t-elle ?
Pour déchiffrer le message chiffré, on utilise la commande :

```
openssl enc -aes-256-cbc -d -in toto.chiffre -out toto.dechiffre
```

Et enfin pour la vérification :

```
diff toto toto.dechiffre
```

c. Commentez chacune des options des commandes utilisées pour le **chiffrement** et le **déchiffrement**.

d. affichez le contenu du chiffré :

```
xxd toto.chiffre
```

Qu'est-ce que c'est que cette en-tête ? Qu'est-ce que fait l'option `-nosalt` ?

Empreinte

Une **empreinte** est un résumé de taille fixe, souvent exprimée en hexadécimal, calculé pour tout fichier de taille variable donné en entrée. La particularité de cette empreinte est :

- de ne pas permettre de retrouver le fichier initial à partir de cette empreinte.
  On parle de « one-way » fonction ou de fonction non inversible.
- de fournir des valeurs très différentes pour des fichiers similaires (ne différant que d'un octet, voir même que d'un seul bit). *Ici, c'est de la notion d'absence de collision dont on parle.*

2 – Récupérez le fichier binaire de la bibliothèque OpenSSL (disponible actuellement en version 3.6.1). Vous le trouverez à l'URL : `http://www.openssl.org/source/`.
Afin de vous assurer que vous avez reçu correctement le logiciel, une *empreinte* au format SHA-256, « *Secure Hash Algorithm 256* » est donnée où vous avez trouvé le fichier à télécharger.

Vous pouvez alors calculer l'**empreinte** du fichier téléchargé à l'aide de la commande

```
openssl dgst -sha1 < openssl-3.6.1.tar.gz
```

a. Faites de même pour SHA-256, « *Secure Hash Algorithm 256* ».
b. En quoi, le calcul de cette empreinte permet d'établir la confiance dans l'archive téléchargée ?
c. Renommez l'archive précédente.
   Est-ce que cela change quelque chose dans le calcul de l'empreinte ?
d. Ajoutez un ou plusieurs caractères à l'archive précédente à l'aide de la commande :

```
echo 'toto' >> openssl-3.6.1.tar.gz
```

   Que se passe-t-il maintenant ?
e. Programmez un petit programme Python permettant :
   - de copier un fichier en supprimant un ou plusieurs caractères n'importe où ;
   - de modifier la valeur d'un caractère n'importe où (par exemple en ne modifiant qu'un seul bit de sa représentation binaire).

   Essayez ce programme sur l'archive originale.
   Quels sont les effets sur le calcul de l'empreinte ?

Rappels : Il est possible de contrôler le programme de calcul d'empreinte à l'aide d'un programme Python de la manière suivante :

```
import subprocess
commande_digest = subprocess.Popen(['openssl', 'dgst', '-sha1'],
            stdin=subprocess.PIPE, stdout=subprocess.PIPE)
commande_digest.stdin.write("Bonjour tout le monde")
commande_digest.stdin.close()

sortie = commande_digest.stdout.read()
print sortie
```

Chiffrement/déchiffrement

Exemple d'utilisation des commandes

Cryptographie symétrique :

- Chiffrement d'un fichier

```
openssl enc -aes256-cbc -in donneeclair.txt -out donneechiffre.enc -pbkdf2
-iter 100000
```

- Déchiffrement d'un fichier

```
openssl enc -aes256-cbc -d -in fichierchiffre.enc -out fichierclair.txt
```

Cryptographie asymétrique :

- Génère une clé privée

```
# RSA moderne (recommandé 3072 ou 4096 bits en 2026)
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out rsa_priv.pem
# ECDSA ou Ed25519 (plus efficace que RSA)
openssl genpkey -algorithm ED25519 -out ed25519_priv.pem
# ML-KEM-768 (Kyber-like, post-quantique)
openssl genpkey -algorithm ML-KEM-768 -out mlkem_priv.pem
# ML-DSA-65 (Dilithium niveau 3)
openssl genpkey -algorithm ML-DSA-65 -out mldsa_priv.pem
```

- Génère une clé publique dérivée d'une clé privée

```
openssl pkey -in algo_priv.pem -pubout -out algo_pub.pem
```

- Chiffre avec une **clé RSA publique** le fichier

```
openssl pkeyutl -encrypt -pubin -inkey rsaclefpublique.pem
  -in fichierclair.txt    -out fichierchiffre.enc
```

- Déchiffre avec la **clé RSA privée** le fichier

```
openssl pkeyutl -decrypt -inkey rsaclefprivee.pem -in fichierchiffre.enc
  -out fichierclair.txt
```

Depuis OpenSSL 3.x, privilégiez genpkey pour tous les algorithmes asymétriques (RSA, EC, EdDSA, ML-KEM, ML-DSA…). Les commandes spécifiques (genrsa, ecparam -genkey, etc.) sont « legacy *».*

3 – Allez sur la rubrique de l'UE sur `http://p-fb.net/` et récupérez le fichier de signature à l'extension `.vcf`.

a. À quoi ressemble le contenu du fichier ?
b. Comment « récupérer » le contenu image sous une forme utilisable ?

4 – Manipulation du chiffrement symétrique :

a. Chiffrez un fichier quelconque que vous aurez choisi, avec l'algorithme de votre choix et dans le mode de votre choix, puis déchiffrez-le.
b. Comparez les tailles des fichiers clairs et chiffrés.
Donnez une explication sur la différence de ces tailles.
c. Tentez de déchiffrer un cryptogramme en utilisant un mauvais mot de passe.
Comment réagit openSSL ?
d. Chiffrez avec le même mot de passe et même algorithme de chiffrement, un même fichier deux fois dans deux fichiers de sortie distincts. Comparez la taille et le contenu de ces deux fichiers obtenus. Expliquez ce que vous observez ?

5 – Manipulation du chiffrement asymétrique avec RSA :

a. Générez une clé privée de taille 4096. Sous quelle forme la clé est-elle fournie ?
b. Comment les deux clés (publique et privée) sont liées ?
c. Étudiez le contenu de la clé privée à l'aide de la commande :

```
openssl rsa -in cle_privee.pem -text
```

Comparez au codage de l'information au format ASN.1 (regarder dans Wikipedia sa définition) :

```
openssl asn1parse -in cle_privee.pem
```

Étudiez le contenu de la clé publique avec la commande :

```
openssl rsa -inform PEM -pubin -in rsaclefpublique.pem -text
```

et :

```
openssl asn1parse -in rsaclefpublique.pem
```

Que pouvez vous en dire ?
d. Chiffrez un document et échangez le avec un collègue. Comment peut-il le déchiffrer ?
e. Pouvez vous générez la même clé que celle de l'un de vos collègues ?
f. Quelles sont les avantages du chiffrement asymétrique par rapport au chiffrement symétrique lors de l'échange de document confidentiel entre deux interlocuteurs ? Trois et plus ?
g. Ce système est-il facilement généralisable à tous les utilisateurs d'Internet qui voudrait bénéficier de la confidentialité dans leurs échanges ? Pourquoi ?

Signature

- Signe, avec la clé privée, le fichier `fichier.txt` en `signature.sig`

```
openssl pkeyutl -sign -inkey rsaclefprivee.pem -in fichier.txt -out signature.sig
```

- Vérifie, avec la clé publique, la signature et sortie dans fichier.txt

```
openssl pkeyutl -verify -pubin -inkey rsaclefpublique.pem -in signature.sig -out fichier.txt
```

6 – Pouvez vous vérifiez l'archive d'openssl récupéré précédemment avec `pgp` :

a. récupérez la signature (fichier d'extension `.asc`) sur la page web d'openssl ;
b. récupérez la clé publique à la fin de la page web (extension `.asc`) ;
c. utilisez les commandes suivantes :

```
$ gpg --no-default-keyring --keyring ./pubkeys.gpg --trust-model always
  --import pubkeys.asc
$ gpg --no-default-keyring --keyring ./pubkeys.gpg --trust-model always
  --verify openssl-3.6.1.tar.gz.asc openssl-3.6.1.tar.gz
```

7 – Est-ce qu'il serait possible de faire de la signature à l'aide d'un chiffrement symétrique ?
Proposez une méthodologie.

Accès SSH sécurisé par clé asymétrique

Pour utiliser une machine à distance, on utilise la commande `ssh`, « secure shell », qui permet de chiffrer les données échangées entre le poste local et la machine distante.

Cette commande réalise :

- ⋆ une connexion à la machine distante ;
- ⋆ négocie l'utilisation d'algorithmes de chiffrement/authentification ;
- ⋆ authentifie l'utilisateur auprès du serveur :
  - ◇ par l'utilisation de login/mot de passe ;
  - ◇ par l'utilisation du chiffrement asymétrique.

Pour utiliser l'authentification par chiffrement asymétrique, il est nécessaire de créer un couple de clés (publique/privée), puis de mettre :

- ▷ la clé publique sur les machines sur lesquelles on veut se connecter ;
- ▷ la clé privée sur la machine depuis laquelle on veut se connecter.

Le répertoire où mettre les clés est « `~/.ssh/` ».
La commande `ssh-keygen` permet de créer les clés dans le format accepté par openssh (il est également possible de « traduire » une clé donnée en format d'openSSL).

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/toto/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/toto/.ssh/id_rsa.
Your public key has been saved in /home/toto/.ssh/id_rsa.pub.
The key fingerprint is:
62:18:c8:e2:27:13:e0:e9:0e:23:15:c3:a7:47:1f:7c
```

Vous pouvez remarquer que l'outil vous fournit une empreinte de la clé afin de pouvoir l'identifier humainement plus facilement.

8 – Vous récupérerez l'empreinte de la clé du serveur `p-fb.net` (ligne contenant par `Server Host Key`) :

```
$ ssh -v toto@p-fb.net
```

Qu'elle est l'empreinte de la clé ?
Est-elle identique à : « `SHA256:ofMSZZk2EOVjAbiAJXaM7ZMX7CEFfj2su5N0+4DlGgg` »

Vous essaierez de vous connecter sur la machine `agate.unilim.fr` (ne fonctionne que depuis le réseau de la FST) :

```
$ ssh -v votre_nom_de_compte@agate.unilim.fr
```

Vous pouvez également utiliser l'option « `-vv` » à la place de « `-v` ».

À quelle clé est liée cette empreinte « `SHA256:BHRaT02VNUUPq+ZVyweAkt069euENhvxk1F3nLpWHq0` » ?

9 – « Comment gâcher son entropie » ou « le jeu de la vie avec ssh » :

a. Essayez la commande suivante :

```
ssh-keygen -t rsa -f /tmp/ma_cle_temp -N "" | tail -n 11 ; rm /tmp/ma_cle_temp
```

Que fait-elle ?

b. Sachant que la commande « `/bin/echo -e "\x1Bc"` » efface l'écran de sortie, programmez un programme réalisant en boucle la génération d'une clé puis qui efface l'écran et recommence…

c. ...et d'ailleurs quels liens avec le « jeu de la vie » de John Conway ?

d. ...et enfin c'est quoi cette « entropie » ?