Table of Contents

Réponses TP3 - Chiffrements

1. Rappel de la différence entre chiffrement symétrique et chiffrement asymétrique

Le **chiffrement symétrique** utilise une clé unique pour chiffrer et déchiffrer les données. Émetteur et récepteur partagent la même clé secrète. Les algorithmes symétriques sont généralement rapides et efficaces pour chiffrer de grandes quantités de données.

Exemples: AES, DES

Le **chiffrement asymétrique** utilise une paire de clés: une clé publique (pour chiffrer) et une clé privée (pour déchiffrer). La clé publique peut être partagée avec n'importe qui, tandis que la clé privée reste secrète. Le processus est plus lent mais résout le problème de la distribution sécurisée des clés.

Exemples: RSA, ECC

2. Relancer le chiffrement avec OpenSSL - même base64 ?

Non, on n'obtient pas le même base64 lors de la relance du chiffrement. C'est normal car:

1. OpenSSL utilise un sel (salt) aléatoire à chaque exécution
2. Ce sel est utilisé avec le mot de passe pour dériver la clé de chiffrement
3. L'IV (vecteur d'initialisation) est également différent à chaque exécution

En utilisant le paramètre `-p`, on peut voir que le sel, la clé et l'IV sont différents à chaque exécution, ce qui explique pourquoi le résultat chiffré (base64) est différent même pour le même message et le même mot de passe.

Commandes utilisées:

```
# Premier chiffrement
openssl enc -aes-256-ctr -salt -e -pbkdf2 -a
# Saisir le mot de passe et le message

# Deuxième chiffrement (avec les mêmes données)
```

```
openssl enc -aes-256-ctr -salt -e -pbkdf2 -a
# Saisir le même mot de passe et message

# Avec affichage du sel, clé et IV
openssl enc -aes-256-ctr -salt -e -pbkdf2 -a -p
# Saisir le mot de passe et le message
```

3. Sécurité de la transmission du mot de passe et du base64

Le mot de passe ainsi que le base64 ont été transmis à l'oral ainsi que sur une discussion Discord, ce qui n'est pas une bonne pratique.

La transmission du mot de passe et du base64 par des moyens non sécurisés (comme un email non chiffré, messagerie instantanée, etc.) n'est pas sécurisée pour plusieurs raisons:

1. Le mot de passe pourrait être intercepté par un attaquant sur le réseau
2. Si le mot de passe est compromis, l'attaquant peut déchiffrer le message
3. La transmission par canal non sécurisé expose les données à des risques d'interception

Une meilleure approche serait d'utiliser un canal de communication sécurisé.

Commandes utilisées (pour chiffrer et déchiffrer):

```
# Chiffrement (administrateur système)
openssl enc -aes-256-ctr -salt -e -pbkdf2 -a
# Saisir mot de passe et message

# Déchiffrement (développeur)
openssl enc -aes-256-ctr -salt -d -pbkdf2 -a
# Saisir le mot de passe et coller le texte en base64
```

4. Piratage de la clé privée

Oui, c'est extrêmement problématique si un attaquant obtient le fichier contenant la clé privée. La clé privée est l'élément qui permet de déchiffrer tous les messages chiffrés avec la clé publique correspondante.

Si un attaquant obtient la clé privée, il peut:

1. Déchiffrer tous les messages précédemment chiffrés avec la clé publique correspondante
2. Se faire passer pour le propriétaire légitime de la clé en signant des documents
3. Compromettre l'intégrité du système de chiffrement

C'est pour cette raison qu'il est recommandé de protéger la clé privée avec un mot de passe fort (comme dans l'exemple avec le paramètre `-aes256`).

Commandes utilisées:

```
# Génération de clé privée sans protection par mot de passe
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:4096

# Génération de clé privée protégée par mot de passe
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:4096 -aes256
```

5. Partage de la clé publique et risques

Le partage de la clé publique ne représente généralement pas un danger, car elle est conçue pour être distribuée librement. La sécurité du chiffrement asymétrique repose sur le fait qu'il est mathématiquement très difficile de déduire la clé privée à partir de la clé publique.

Cependant, certains risques peuvent exister:

1. L'usurpation d'identité: quelqu'un pourrait prétendre que sa clé publique vous appartient
2. L'attaque de l'homme du milieu: un attaquant pourrait intercepter la communication et

remplacer votre clé publique par la sienne

Pour atténuer ces risques, il est important de vérifier l'authenticité de la clé publique reçue (par exemple, en comparant les empreintes digitales de la clé via un canal sécurisé).

Commandes utilisées:

```
# Dérivation de la clé publique à partir de la clé privée
openssl pkey -pubout -in private_key.pem -out public_key.pem

# Affichage du contenu des clés
cat private_key.pem
cat public_key.pem
```

6. Chiffrement symétrique vs asymétrique - cas d'usage

Utilisation préférentielle du chiffrement symétrique:

- Pour le chiffrement de grandes quantités de données (fichiers, disques durs)
- Quand la performance est critique (le chiffrement symétrique est beaucoup plus rapide)
- Dans des environnements contrôlés où l'échange sécurisé de clés est possible
- Pour les communications en temps réel nécessitant un chiffrement rapide

Utilisation préférentielle du chiffrement asymétrique:

- Pour l'échange sécurisé de clés dans un réseau non sécurisé
- Pour authentifier l'identité (signatures numériques)
- Quand il y a besoin de communiquer avec plusieurs parties sans partager de secret commun
- Pour établir un canal sécurisé initial avant de passer au chiffrement symétrique (comme dans TLS/SSL)

En pratique, les systèmes modernes utilisent souvent une approche hybride: le chiffrement asymétrique pour échanger une clé symétrique, puis le chiffrement symétrique pour les données.

Commandes utilisées:

```
# Chiffrement asymétrique
openssl rsautl -encrypt -inkey public_key.pem -pubin -out message.enc
# Saisir le message

# Déchiffrement asymétrique
openssl rsautl -decrypt -inkey private_key.pem -in message.enc
```

7. Différence GPG vs OpenSSL

Avec GPG, le message chiffré est automatiquement formaté avec un en-tête qui inclut des informations comme la version de GPG, l'algorithme utilisé, et d'autres métadonnées. Le format ressemble à:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v2
...
-----END PGP MESSAGE-----
```

Contrairement à OpenSSL qui produit une sortie brute encodée en base64, GPG encapsule le message dans un format défini qui fournit plus de contexte.

Le GPG gère automatiquement les clés dans un trousseau, contrairement à OpenSSL où les fichiers de clés doivent être gérés manuellement.

Le GPG est plus convivial et orienté utilisateur final, avec une meilleure intégration.

Commandes utilisées:

```
# Chiffrement symétrique avec GPG
gpg --symmetric --pinentry-mode=loopback -a
# Saisir le mot de passe et le message

# Déchiffrement avec GPG
```

```
gpg --decrypt --pinentry-mode=loopback
# Coller le message chiffré
```

8. Algorithme et longueur de clé par défaut dans GPG

En générant une nouvelle paire de clés avec GPG, j'ai pu observer que par défaut:

- L'algorithme utilisé est **RSA**
- La longueur de clé est de **3072 bits**
- La clé est valide pour une durée de **2 ans** (24 mois)

Ces valeurs par défaut sont considérées comme sécurisées selon les standards actuels. La longueur de 3072 bits offre un bon compromis entre sécurité et performance, tandis que la validité de 2 ans encourage les utilisateurs à renouveler régulièrement leurs clés.

Commandes utilisées:

```
# Générer une clé GPG avec les paramètres par défaut
gpg --gen-key --pinentry-mode=loopback

# Lister les clés pour voir leurs caractéristiques
gpg --list-secret-keys
gpg --list-keys
```

9. GPG plus facile que OpenSSL?

Oui, GPG est généralement considéré comme plus facile d'utilisation quotidienne qu'OpenSSL pour plusieurs raisons:

1. **Interface plus intuitive**: Les commandes GPG sont plus faciles à mémoriser et à utiliser
2. **Gestion des clés intégrée**: GPG maintient un trousseau de clés qui simplifie la gestion des clés publiques et privées
3. **Format standardisé**: Le format PGP est largement reconnu et standardisé

4. **Moins d'erreurs possibles**: GPG a des valeurs par défaut sécurisées et nécessite moins de paramètres techniques
5. **Plus d'automatisation**: Beaucoup d'opérations courantes sont automatisées

OpenSSL est plus puissant et polyvalent, mais cette flexibilité se traduit par une complexité accrue qui peut être source d'erreurs si on ne maîtrise pas bien l'outil.

Commandes comparées:

```
# Génération de clés avec GPG
gpg --gen-key --pinentry-mode=loopback

# vs génération de clés avec OpenSSL (plusieurs étapes)
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:4096
openssl pkey -pubout -in private_key.pem -out public_key.pem
```

10. Authenticité de l'expéditeur (administrateur système)

Non, il est impossible d'affirmer avec certitude que le message chiffré provient bien de l'administrateur système. Dans le cas du chiffrement asymétrique avec GPG, n'importe qui en possession de ma clé publique (le développeur) pourrait chiffrer un message et prétendre être l'administrateur système.

Exemple illustratif: Imaginons qu'un attaquant "Charlie" souhaite se faire passer pour l'administrateur système. Il pourrait:

1. Récupérer ma clé publique (celle du développeur) qui est disponible librement
2. Chiffrer un faux message avec ma clé publique (ex: "Voici le mot de passe de la base de données: MauvaisMotDePasse123")
3. M'envoyer ce message en prétendant être l'administrateur système

Comme je peux déchiffrer ce message avec ma clé privée, je n'ai aucun moyen technique de savoir s'il provient réellement de l'administrateur système ou de l'attaquant Charlie.

Pour garantir l'authenticité de l'expéditeur, l'administrateur système devrait:

- Signer le message avec sa clé privée en plus de le chiffrer avec ma clé publique
- Je pourrais alors vérifier la signature avec sa clé publique pour confirmer l'identité de l'expéditeur

La signature numérique résout ce problème en apportant la garantie d'identité que le simple chiffrement ne peut pas offrir.

Commandes utilisées:

```
# Simple chiffrement (sans authentification)
gpg -a --encrypt --recipient dev@target.lxc
# Saisir le message

# Chiffrement avec signature (garantit l'authenticité)
gpg -a --sign --encrypt --recipient dev@target.lxc
# Saisir le message

# Déchiffrement et vérification de signature
gpg --decrypt
# Coller le message chiffré
```

11. Altération du message signé - résultat de la vérification

Cette partie a été faite sur mon WSL car la VM a un peu galérée

Lorsqu'un message signé est altéré, même légèrement, la vérification de signature avec GPG échoue. Le système affiche généralement un message d'erreur explicite comme:

```
gpg: Signature made [date and time]
gpg: using RSA key [key ID]
gpg: BAD signature from "User Name <email@example.com>"
```

C'est exactement le but d'une signature numérique: garantir l'intégrité du message. La moindre modification du contenu après signature sera détectée lors de la vérification, car la fonction de hachage utilisée dans le processus de signature produira une valeur différente pour le message modifié.

Cette caractéristique est cruciale pour détecter toute tentative de falsification d'un message signé.

Commandes détaillées à exécuter:

Sur la machine du développeur (target-dev):

```
# 1. Créer un message signé (sans chiffrement)
gpg --clear-sign
# Saisir un message de test, par exemple: "Ceci est un message de test
important"
# Appuyer sur Ctrl+D pour terminer la saisie

# 2. Copier le message signé complet (y compris les en-têtes)
# Le message aura cette structure:
# -----BEGIN PGP SIGNED MESSAGE-----
# Hash: SHA256
#
# Ceci est un message de test important
# -----BEGIN PGP SIGNATURE-----
# ...signature...
# -----END PGP SIGNATURE-----
```

Sur la machine de l'administrateur système (target-admin):

```
# 3. Créer un fichier temporaire pour le message signé
nano message_signe.txt
# Coller le message signé complet, enregistrer et quitter (Ctrl+O,
Enter, Ctrl+X)

# 4. Vérifier la signature intacte
gpg --verify message_signe.txt
# Le résultat doit indiquer "Good signature"

# 5. Modifier le message
nano message_signe.txt
# Modifier le texte du message (par exemple, changer "important" en
"crucial")
# Sauvegarder les modifications (Ctrl+O, Enter, Ctrl+X)
```

```
# 6. Vérifier à nouveau la signature après modification
gpg --verify message_signe.txt
# Le résultat doit maintenant indiquer "BAD signature"
```

Cette séquence de commandes permet de démontrer concrètement comment GPG détecte immédiatement toute altération du message signé, même minime, garantissant ainsi l'intégrité du message original.