Compte rendu des activités du projet

Sommaire

1. Protection_AES — Chiffrement symétrique avec OpenSSL

1.1 Objectif

Comprendre, de manière concrète, comment chiffrer et déchiffrer un fichier avec AES-256-CBC via OpenSSL, en manipulant **explicitement** la clé et l'IV plutôt qu'en laissant OpenSSL dériver ces valeurs à partir d'un mot de passe.

1.2 Fichiers et rôle de chacun

- `Protection_AES/hello.txt` : message en clair de test.
- `Protection_AES/encrypt.sh` : script bash de **chiffrement**.
- `Protection_AES/decrypt.sh` : script bash de **déchiffrement**.
- `Protection_AES/hello.enc` : fichier chiffré obtenu.
- `Protection_AES/hello.dec.txt` : fichier déchiffré pour vérification.

Le README rappelle la commande de base OpenSSL :

```
openssl enc -aes-256-cbc -in hello.txt -out hello.enc -K
    <clé_hex_64_chars> -iv <iv_hex_32_chars>
```

Ici : - `-aes-256-cbc` : algorithme AES 256 bits en mode CBC ; - `-K` : clé **en hexadécimal**, 32 octets -> 64 caractères hex ; - `-iv` : vecteur d'initialisation (IV) en hexadécimal, 16 octets -> 32 caractères hex.

1.3 Étapes pas à pas

1. **Se placer dans le dossier de l'activité** :

```
cd Protection_AES
```

2. **Générer une clé et un IV aléatoires valides pour AES-256-CBC** :

```
# clé 256 bits (32 octets -> 64 caractères hex)
HEXKEY=$(openssl rand -hex 32)
# IV 128 bits (16 octets -> 32 caractères hex)
HEXIV=$(openssl rand -hex 16)

echo "Key: $HEXKEY"
echo "IV : $HEXIV"
```

3. **Chiffrer le message** en utilisant le script :

```
./encrypt.sh hello.txt hello.enc "$HEXKEY" "$HEXIV"
```

Le script : - vérifie que `$HEXKEY` fait bien 64 caractères, sinon il s'arrête ; - vérifie que `$HEXIV` fait 32 caractères ; - appelle ensuite `openssl enc -aes-256-cbc -in ... -out ... -K ... -iv ...`.

4. **Déchiffrer le message** :

```
./decrypt.sh hello.enc hello.dec.txt "$HEXKEY" "$HEXIV"
```

5. **Comparer fichier d'origine et fichier déchiffré** :

```
diff hello.txt hello.dec.txt
```

- s'il n'y a **aucune sortie**, les fichiers sont identiques ;
- sinon, le diff montre les différences (clé/IV incorrects, fichier corrompu, etc.).

1.4 Résultats observés

- `hello.enc` contient des données pseudo-aléatoires illisibles.
- `hello.dec.txt` est identique au texte d'origine.

1.5 Intérêt et points d'attention

- Comprendre la **taille exacte** attendue des clés et IV pour AES-256-CBC.
- Visualiser le rôle du couple (clé, IV) : même clé mais IV différent -> chiffré différent.
- Ne jamais réutiliser la même paire (clé, IV) sur plusieurs messages en mode CBC.
- En pratique, privilégier un mode authentifié (AES-GCM) et une gestion robuste des clés.

---

2. Protection en authenticité — Signatures numériques avec OpenSSL

2.1 Objectif

Apprendre à **signer** un message et à **vérifier** cette signature pour garantir l'authenticité et l'intégrité du contenu, en utilisant une paire de clés RSA (privée/publique).

2.2 Fichiers et structure

- Dossier : `Protection en authenticité/`.
- `make.sh` : script qui orchestre toute la démonstration :
  - création du message `hello.txt` ;
  - génération de la clé privée `private.key` ;
  - extraction de la clé publique `public.key` ;
  - signature du message dans `hello.sig` ;
  - vérification de la signature.
- `Makefile` :
  - `make` / `make all` : lance `make.sh` ;
  - `make verify` : relance seulement la vérification ;
  - `make clean` : supprime `hello.txt`, `hello.sig`, `private.key`, `public.key`.
- `README.md` : explique les commandes `openssl pkeyutl` et le cas des gros fichiers.

2.3 Étapes pas à pas (cas d'un petit fichier)

1. **Se placer dans le dossier** :

```
cd "Protection en authenticité"
```

2. **Générer message, clés et signature avec `make`** :

```
make
```

Ce qui se passe dans `make.sh` :

- écriture du message :

```
printf 'Bonjour, ceci est un message a signer.\n' >
    hello.txt
```

- génération de la clé privée RSA 2048 bits :

```
openssl genpkey -algorithm RSA -out private.key
    -pkeyopt rsa_keygen_bits:2048
```

- extraction de la clé publique :

```
openssl rsa -in private.key -pubout -out public.key
```

- signature du message :

```
openssl pkeyutl -sign -in hello.txt -inkey private.key
    -out hello.sig
```

- vérification immédiate :

```
openssl pkeyutl -verify -in hello.txt -pubin -inkey
    public.key -sigfile hello.sig
```

3. **Rejouer uniquement la vérification après modification potentielle** :
   - si l'on modifie `hello.txt` (ou `hello.sig`), on peut tester :

```
make verify
```

   - la commande `pkeyutl -verify` renverra une erreur si la signature ne correspond plus au contenu.
4. **Nettoyer l'activité** :

```
make clean
```

2.4 Cas d'un fichier volumineux (signature via condensat)

Pour un gros fichier, `openssl pkeyutl -sign` direct n'est pas adapté. La bonne pratique :

1. **Côté expéditeur** :
   - calculer un hash SHA-256 du fichier et le signer en une seule commande :

```
openssl dgst -sha256 -sign private.key -out
    fichier.sig fichier.txt
```

   - envoyer `fichier.txt` et `fichier.sig` au destinataire (éventuellement aussi le hash à part si besoin).
2. **Côté destinataire** :
   - vérifier la signature :

```
openssl dgst -sha256 -verify public.key -signature
    fichier.sig fichier.txt
```

   - si la signature est valide, le hash recalculé côté destinataire est exactement celui qui a été signé par l'expéditeur.

2.5 Résultats observés

- Si la signature est correcte :
  - aucune sortie texte, mais code de retour 0 (`$? == 0`).
- Si le message a été modifié ou la clé publique ne correspond pas :
  - `openssl` affiche une erreur et renvoie un code != 0.

2.6 Intérêt et points d'attention

- Séparation claire des rôles :
  - **clé privée** : signer ;
  - **clé publique** : vérifier.
- La signature apporte **authenticité + intégrité** ; un hash seul n'apporte que l'intégrité.
- Bien protéger la clé privée (`private.key`) : elle ne doit jamais être transmise.

---

3. Vérifier un téléchargement — Intégrité avec SHA-256

Objectif : vérifier qu'un fichier téléchargé (`openssl-3.6.0.tar.gz`) n'a pas été corrompu ou modifié, à l'aide d'un fichier `.sha256`.

Implémentation : - Dossiers : - `Verifier_telechargement/fichier_integre/` : cas où le fichier est intact. - `Verifier_telechargement/fichier_modifie/` : cas où le fichier ne correspond plus au hash. - Dans chaque dossier : - `openssl-3.6.0.tar.gz.sha256` : hash attendu ; - `check.sh` : script qui : - vérifie la présence du tarball et du `.sha256` ; - appelle `sha256sum -c <fichier>.sha256` ; - affiche un message clair si l'intégrité est OK ou non. - Les README expliquent la commande : `bash sha256sum -c openssl-3.6.0.tar.gz.sha256` et rappellent la différence **intégrité** vs **authenticité**.

Comment l'utiliser :

```
cd Verifier_telechargement/fichier_integre
./check.sh   # doit afficher "OK"

cd ../fichier_modifie
./check.sh   # doit échouer si le tarball a été modifié
```

Résultats : - Cas intègre : `openssl-3.6.0.tar.gz: OK` puis message de succès. - Cas modifié : `FAILED` et message d'erreur.

Intérêt / points d'attention : - Vérifier systématiquement les hash fournis par l'éditeur pour éviter les fichiers corrompus. - L'intégrité seule ne prouve pas l'origine : pour l'**authenticité**, il faut vérifier une signature PGP ou équivalent.

---

4. mdp secure — Vérificateur de mot de passe hex avec temporisation

4.1 Objectif

Implémenter un programme C qui vérifie un mot de passe **hexadécimal** avec : - une **saisie masquée** ; - un format strict (0-9, A-F, longueur max 8) ; - une **temporisation** après 3 échecs pour limiter la force brute.

4.2 Fichiers et structure

- Dossier : `mdp secure/`.
- `src/main.c` : code source C du programme.
- `bin/checkpwd` : binaire généré.
- `Makefile` : compilation avec `gcc` et options de warning.
- `README.md` : rappel du comportement et de l'usage.

Le mot de passe attendu est `CAFECAFE` (8 caractères hex), comparaison insensible à la casse.

4.3 Étapes pas à pas

1. **Compilation du programme** :

```
cd "mdp secure"
make
```

   Le `Makefile` :

   - crée le répertoire `bin/` si nécessaire ;
   - compile `src/main.c` en `bin/checkpwd` avec `-Wall -Wextra -O2`.

2. **Lancement du vérificateur** :

```
./bin/checkpwd
```

   Le programme affiche :

```
Entrez le mot de passe (hexa, max 8):
```

   La saisie est **masquée** (rien ne s'affiche pendant que tu tapes).

3. **Cas d'un mot de passe invalide** (mauvaise valeur ou mauvais format) :

   - Exemple : `1234567G` (G n'est pas hexadécimal) ou `123456789` (> 8 chars).
   - Le programme répond immédiatement :

```
MDP KO
```

- Le compteur d'échecs `failures` est incrémenté.

4. **Cas du mot de passe correct** :
   - Entrez par exemple : `cafecafe` ou `CAFECAFE`.
   - Le programme :
     - convertit chaque caractère en majuscule (`toupper`) ;
     - compare à la constante `"CAFECAFE"` ;
     - affiche :

```
MDP OK
```

     - puis quitte avec code de retour 0.

5. **Temporisation après 3 échecs** :
   - Après trois `MDP KO`, la variable `failures` atteint le seuil `LOCK_THRESHOLD = 3`.
   - Le programme mémorise l'instant du dernier échec (`last_attempt`) et, à chaque nouvelle boucle, calcule :

```
next_allowed = last_attempt + LOCK_SECONDS; // 10 minutes
```

   - Tant que `time(NULL)< next_allowed`, il n'autorise pas une nouvelle saisie et affiche régulièrement :

```
Trop d'echecs (3). Nouvel essai possible dans X minute(s) Y seconde(s).
```

     puis `sleep(1);` pour éviter un busy loop.

4.4 Résultats observés

- Mot de passe correct immédiatement : `MDP OK`, fin du programme.
- Mot de passe erroné <= 3 fois : `MDP KO` à chaque fois, sans délai.
- À partir du 3e échec :
  - l'utilisateur voit un compte à rebours avant tout nouvel essai ;
  - ceci rend toute attaque par essais rapides beaucoup plus lente.

4.5 Intérêt et points d'attention

- Illustre l'usage des bibliothèques standard C :
  - `termios` pour manipuler le terminal (masquer la saisie) ;
  - `time` et `sleep` pour la temporisation ;
  - `ctype.h` pour le test/normalisation hex (`isxdigit`, `toupper`).
- Montre une contre-mesure simple contre la force brute : **ralentir** les essais.
- Limites pédagogiques : mot de passe en clair dans le binaire, état de blocage non persistant entre exécutions.

---

5. OpenSSL_dgst — Génération de haché SHA-256

5.1 Objectif

Montrer comment générer un fichier texte simple, puis calculer et afficher son **condensat SHA-256**, base de la vérification d'intégrité et des signatures.

5.2 Fichiers

- Dossier : `OpenSSL_dgst/`.
- `make_hello.sh` : script bash qui automatise la création du fichier et le calcul du hash.

5.3 Étapes pas à pas

1. **Se placer dans le dossier** :

```
cd OpenSSL_dgst
```

2. **Exécuter le script** :

```
./make_hello.sh
```

   Ce script :

   - écrit `Hello world` (sans saut de ligne) dans `hello.txt` :

```
printf 'Hello world' > hello.txt
```

   - calcule le hash SHA-256 du fichier :

```
sha256sum hello.txt
```

3. **Interpréter la sortie** :

   - La commande `sha256sum` affiche une ligne de la forme :

```
<64 caractères hex>  hello.txt
```

   - cette valeur à 64 caractères est le condensat SHA-256 du contenu **exact** de `hello.txt`.

5.4 Intérêt et points d'attention

- Comprendre qu'un **petit changement** dans le fichier (un caractère en plus, un saut de ligne, etc.) donnera un hash complètement différent.
- Cette étape est la brique de base des mécanismes de :
  - vérification de téléchargements (`sha256sum -c`) ;
  - signatures numériques (on signe un haché) ;
  - stockage de références de fichiers.

---

6. Analyse statique — Code C et assembleur, ELF et strip

6.1 Code C (`Code/AnalyseStatique/codeC`)

6.1.1 Objectif Écrire un programme C minimal, le compiler sur différentes architectures (x86_64, aarch64) et analyser la **structure ELF** du binaire résultant (sections, symboles, taille, strip).

6.1.2 Fichiers

- Dossier : `Code/AnalyseStatique/codeC`.
- `hello.c` :

```c
#include <stdio.h>

int main(void) {
          puts("Hello, world!");
          return 0;
}
```

- `hello` : exécutable compilé pour chaque architecture.
- `hello.stripped` : version du binaire après `strip`.
- `README.md` : contient les sorties de `readelf` et les conclusions.

6.1.3 Étapes pas à pas (x86_64)

1. **Compilation avec options de vérification** :

```
cd Code/AnalyseStatique/codeC
gcc -Wall -Wextra -O2 -o hello hello.c
```

2. **Exécution du programme** :

```
./hello
```

Sortie attendue :

```
Hello, world!
```

3. **Inspection de l'en-tête ELF** :

```
LANG=C readelf -h hello
```

Permet de voir :

- type (`Type: DYN (Position-Independent Executable file)`) ;
- architecture (`Machine: Advanced Micro Devices X86-64`) ;
- adresse d'entrée (`Entry point address`) ;

- nombre de segments et sections.

4. **Liste détaillée des sections** :

```
LANG=C readelf -S hello
```

Points marquants :

- 31 sections (voir `README.md`) ;
- sections de code (`.text`, `.plt`, `.init`, `.fini`),
- sections de données (`.data`, `.bss`, `.rodata`),
- métadonnées (`.dynsym`, `.dynstr`, `.rela.*`, `.eh_frame`, etc.).

5. **Création d'un binaire strippé** :

```
strip -o hello.stripped hello
stat -c "%n %s" hello hello.stripped
```

- taille de `hello` : environ 15 960 octets ;
- taille de `hello.stripped` : environ 14 472 octets.

6.1.4 Étapes pas à pas (aarch64 / ARM64) Sur la VM ARM64, la commande est similaire :

```
gcc -Wall -Wextra -O2 -o hello hello.c
LANG=C readelf -h hello
LANG=C readelf -S hello
```

Le `README.md` fournit la sortie détaillée : - type `DYN` (PIE) ; - sections et segments adaptés à l'architecture AArch64 ; - sections de relocalisation `.rela.dyn`, `.rela.plt`, etc.

6.1.5 Intérêt

- Comprendre la **structure interne** d'un binaire ELF généré par `gcc`.
- Voir l'impact du linking avec la glibc : beaucoup de sections et de symboles même pour un programme minimal.
- Observer l'effet de `strip` : réduction de la taille en supprimant symboles et infos de debug.

6.2 Code assembleur (`Code/AnalyseStatique/codeAssembler`)

6.2.1 Objectif Écrire directement un programme en **assembleur** (x86_64 et ARM64) qui affiche « Hello, World! » en utilisant les appels système Linux, puis comparer le binaire obtenu à celui en C.

6.2.2 Fichiers

- Dossier : `Code/AnalyseStatique/codeAssembler`.
- `hello.asm` (x86_64, NASM).
- `hello` : exécutable assemblé et lié.
- `README.md` : code assembleur x86_64, exemple ARM64, sortie de `readelf` pour ARM.

6.2.3 Étapes pas à pas (x86_64)

1. **Contenu du fichier assembleur** :
   - Définit la chaîne `"Hello, World!\n"` dans `.data` ;
   - `_start` effectue :
     - appel système `write(1, msg, len)` ;
     - appel système `exit(0)`.
2. **Assemblage et édition de liens** :

```
cd Code/AnalyseStatique/codeAssembler
nasm -f elf64 -o hello.o hello.asm
ld -o hello hello.o
```

3. **Exécution** :

```
./hello
```

   Résultat :

```
Hello, World!
```

4. **Analyse du binaire** :

```
LANG=C readelf -h hello
LANG=C readelf -S hello
```

   On observe un binaire très simple, avec peu de sections.

6.2.4 Étapes pas à pas (ARM64) Le `README.md` fournit un exemple d'assembleur ARM64 : - usage de `x0` (stdout), `x1` (adresse du message), `x2` (longueur), `x8` (numéro d'appel système) ; - `svc #0` pour déclencher l'appel système.

Les sorties `readelf` pour l'exécutable ARM montrent : - très peu de sections (6) ; - deux segments LOAD (code + données) ; - absence de sections de relocalisation dynamiques.

6.2.5 Intérêt

- Comprendre le lien direct entre **instructions assembleur** et ABI Linux.
- Voir la différence entre un binaire pur assembleur et un binaire C lié à la glibc.
- Approcher la structure ELF la plus minimaliste possible.

---

7. Analyse dynamique — Prise en main de GDB (ARM64)

7.1 Objectif

Se familiariser avec **GDB** sur une architecture ARM64 pour des exercices de débordement de pile : compréhension des registres, de la pile, du mapping mémoire et du comportement du programme compilé avec des protections désactivées.

7.2 Fichiers

- Dossier : `Code/AnalyseDynamique/Ex5/`.
- `MacOS.md` : mémo de commandes GDB et d'options de compilation.
- (fichier `overflow.c` non fourni dans le dépôt mais mentionné dans la doc).

7.3 Compilation typique d'un binaire vulnérable

Dans `MacOS.md` :

```
gcc -g -fno-stack-protector -z execstack -no-pie overflow.c -o
    overflow
```

Signification des options : - `-g` : inclut les symboles de debug (noms de fonctions, lignes source) pour GDB. - `-fno-stack-protector` : désactive les **canaris de pile** (protection contre les overflows). - `-z execstack` : rend la pile exécutable (dangereux en production, utile pour tester les exploits avec shellcode). - `-no-pie` : binaire non PIE -> adresses virtuelles **fixes**, plus simples à analyser et exploiter.

7.4 Correspondance des registres x86_64 / ARM64

Le mémo rappelle :

x86_64	ARM64	Rôle
`rbp`	`x29`	Frame pointer
`rsp`	`sp`	Stack pointer
`rip`	`pc`	Instruction ptr

Cela permet de retrouver les mêmes concepts (pointeur de pile, pointeur de base, compteur de programme) sur ARM64.

7.5 Étapes pas à pas avec GDB

1. **Lancer GDB sur le binaire** :

```
gdb ./overflow
```

2. **Placer un breakpoint sur `main`** :

```
(gdb) break main
```

3. **Démarrer le programme** :

```
(gdb) run
```

4. **Afficher les registres généraux** :

```
(gdb) info registers
```

- permet de voir `x29`, `sp`, `pc`, etc.

5. **Inspecter un registre particulier** (par ex. `x29`) :

```
(gdb) info registers x29
```

6. **Afficher la carte mémoire du processus** (mappings) :

```
(gdb) info proc mappings
```

- montre les segments texte, data, heap, stack, bibliothèques partagées, etc.

7.6 Intérêt et points d'attention

- Comprendre visuellement l'impact des options de compilation sur la surface d'attaque.
- Sur ARM64, se repérer dans les registres (`x0`–`x30`, `sp`, `pc`, `x29` frame pointer).
- `-z execstack` + `-no-pie` ne doivent servir qu'en environnement de **lab** pour tester des exploits, jamais en production.

---

8. Exercice PS4 — Périmètre EBIOS (modélisation périmètre métier/technique)

8.1 Objectif

Modéliser le **périmètre métier et technique** d'un système (ici une PS4 et ses périphériques) en vue d'une analyse de risque EBIOS : quels composants, quelles liaisons, quels points d'entrée ?

8.2 Fichiers

- Dossier : `Exercice PS4/Atelier 1/`.
- `perimetre.puml` : diagramme PlantUML décrivant les composants et leurs relations.

8.3 Contenu du diagramme

Le fichier définit des composants techniques :

- `PS4`, `TV`, `Sono`, `CD`, `Disque dur interne`, `Disque dur externe`, `Box`, `Internet`, `Clavier`, `Caméra`, `Casque`, `Clé USB`, `Manette`…

Et les **liaisons** :

- PS4 <=> TV : HDMI
- PS4 <=> Sono : audio (PiF)
- PS4 <=> Box : WiFi, Ethernet
- PS4 <=> Manette / Clavier / Caméra / USB / Casque / HDD externe : USB
- PS4 <=> Casque / Manette : Bluetooth
- Box <=> Internet : connexion réseau.

8.4 Génération du diagramme

Avec PlantUML (VS Code + extension ou outil en ligne) :

1. Ouvrir `perimetre.puml`.
2. Lancer la prévisualisation PlantUML (dans VS Code, via la commande de l'extension) ou utiliser un site PlantUML pour coller le contenu.
3. Exporter en image (PNG/SVG) si besoin pour les supports.

8.5 Intérêt et points d'attention

- C'est la **première étape** d'une analyse de risque : délimiter le périmètre technique.
- Permet d'identifier :
  - tous les liens physiques/logiques (ports USB, réseau, stockage externe, sans-fil) ;
  - les potentiels points d'entrée et de sortie de données ;
  - les dépendances (Box, Internet, périphériques tiers).

Cette vision globale servira ensuite de base aux études de menaces et scénarios EBIOS.

---

9. Synthèse globale du projet

Ce projet couvre plusieurs briques fondamentales de la sécurité des systèmes d'exploitation :

- **Chiffrement symétrique (Protection_AES)** : garantir la confidentialité des données avec AES-256-CBC, en comprenant les notions de clé, IV et bonnes pratiques d'usage.
- **Authenticité et signatures (Protection en authenticité)** : assurer que les données proviennent bien du bon émetteur et n'ont pas été altérées, grâce aux signatures asymétriques.
- **Intégrité des téléchargements (Verifier_telechargement, OpenSSL_dgst)** : vérifier qu'un fichier n'a pas été modifié par accident ou de manière malveillante à l'aide de hachés SHA-256.
- **Protection des mots de passe (mdp secure)** : illustrer la mise en œuvre d'un contrôle de mot de passe robuste côté client, avec format restreint, saisie masquée et temporisation anti-force-brute.
- **Analyse statique (code C / assembleur)** : comprendre la structure interne des exécutables ELF, le rôle des sections et symboles, et l'impact du stripping.
- **Analyse dynamique (GDB)** : apprendre à instrumenter un binaire (notamment vulnérable) pour observer registres, pile et mémoire, en tenant compte des spécificités ARM64.
- **Modélisation du périmètre (Exercice PS4)** : poser le contexte métier et technique avant une analyse de risque, en matérialisant les composants et leurs interactions.

Points d'attention transverses : - Toujours distinguer **confidentialité**, **intégrité** et **authenticité** : chaque exercice illustre une facette différente. - Comprendre les options de compilation/d'exécution qui renforcent ou affaiblissent la sécurité (canaris, NX, PIE, etc.). - Garder à l'esprit que les exemples sont pédagogiques : en production, il faut ajouter gestion sérieuse des clés, stockage sécurisé des secrets, journalisation, limitation fine des droits, etc.