FONDAMENTAUX DE LA SÉCURITÉ
ET CRYPTOGRAPHIE
Cours et Travaux Pratiques - Niveau Ingénieur
Présenté par: Rachid Bouselama
Superviseur: Mr. Lahcen AIT IBOUREK

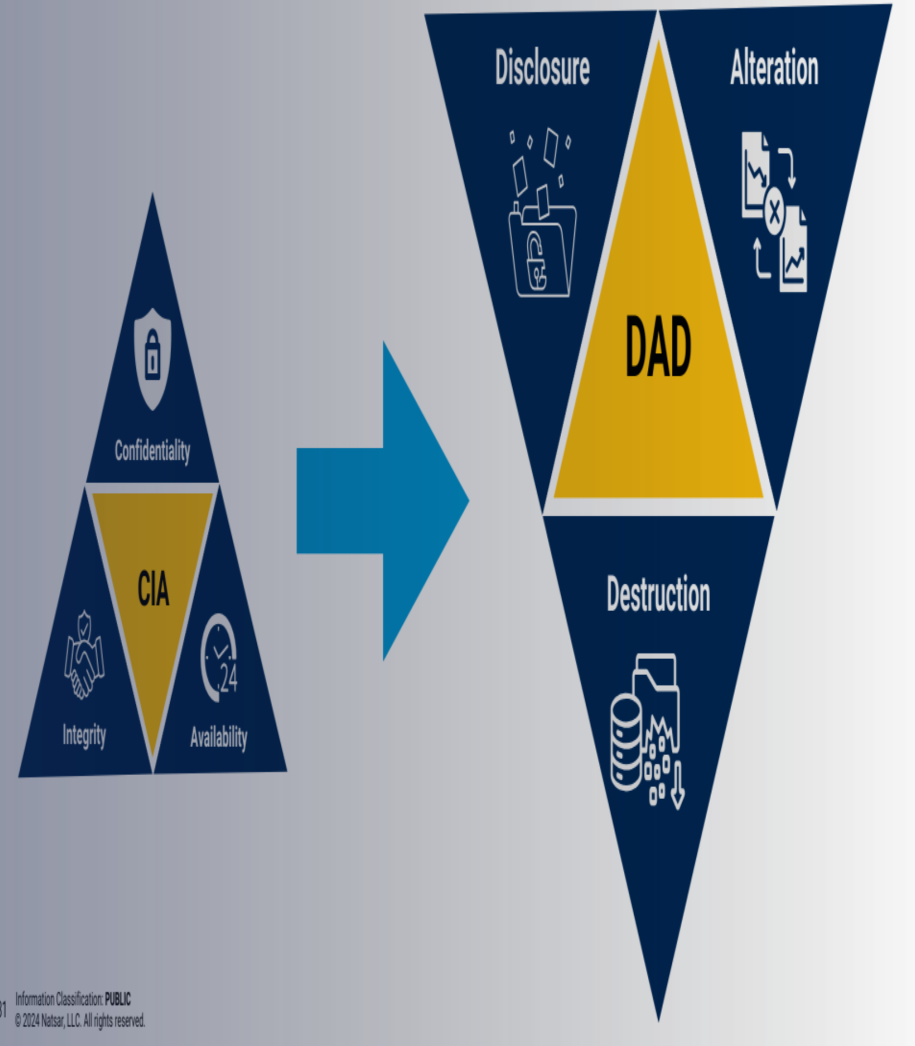
Fondamentaux de la Sécurité et Cryptographie

Partie A - Les Fondations et la Cryptographie Symétrique

Objectif : Comprendre pourquoi on n'invente pas son propre algorithme, maîtriser AES et l'importance critique des modes d'opération

The Foundation of Cybersecurity: CIA

CIA protects against DAD



La Triade CIA

Le socle de la cybersécurité

Confidentialité

Seules les personnes autorisées lisent les données

Ex : Chiffrement AES pour emails et communications

Intégrité

Les données n'ont pas été modifiées

Ex : Hash SHA-256, signatures numériques

Disponibilité

Le système fonctionne quand on en a besoin

Ex : Protection DDoS, sauvegardes redondantes, CDN

Bonus : Non-Répudiation

Empêche de nier avoir effectué une action

Ex : Signature électronique, journaux d'audit

Vocabulaire Clé et Règle d'Or

Les Termes Essentiels

Cryptographie
L'art de coder les messages

Cryptanalyse
L'art de casser un code sans clé

Plaintext
Message lisible (texte clair)

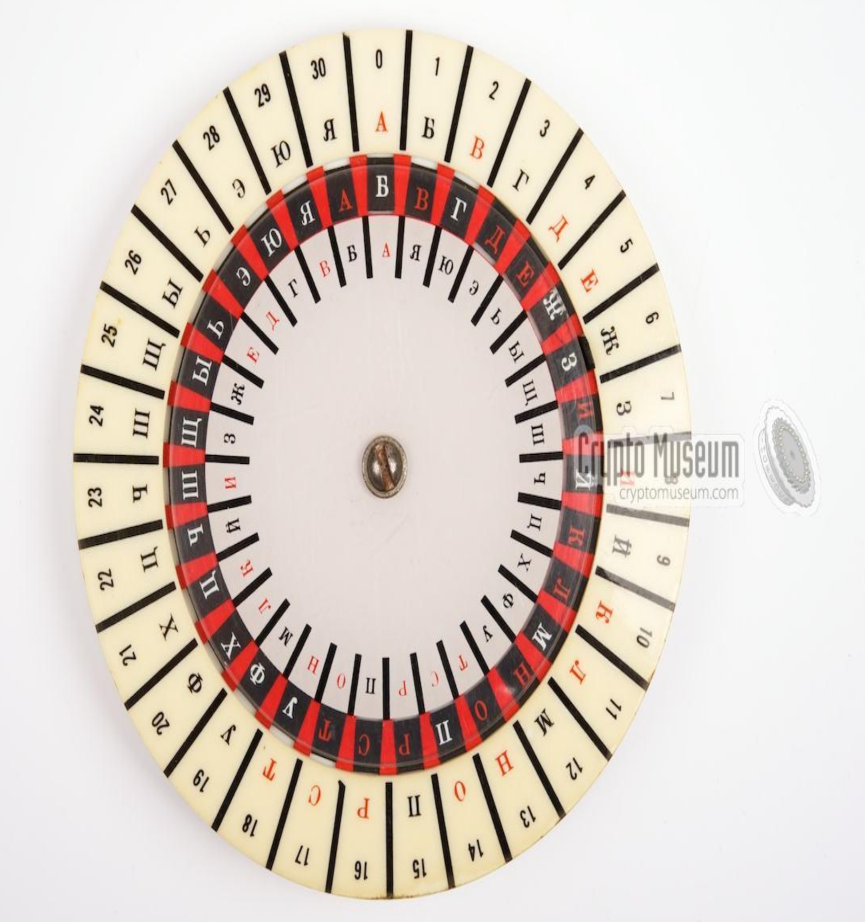
Ciphertext
Message illisible (chiffré)

LE PRINCIPE DE KERCKHOFFS (1883)

La sécurité ne doit reposer que sur le **secret de la clé**, pas sur celui de l'algorithme

→ L'algorithme doit être public, audité et standardisé
→ "Security by Obscurity" ne fonctionne jamais

alamy
Image ID: 2EEEK31
www.alamy.com



L'Histoire : César et Analyse Fréquentielle

Chiffre de César

Principe : Décalage de l'alphabet

Formule : C = (P + k) mod 26

⚠ **Problème :** Seulement 25 clés possibles → Cassable en **1 milliseconde**

Analyse Fréquentielle

Les langues ont une signature statistique

En français : E ≈ **15%** A ≈ **8%** S ≈ **7%**

Si un symbole apparaît 15% dans le chiffré → Probablement = E

LEÇON CRITIQUE

Un bon chiffrement doit **détruire** les statistiques du langage

Exemple : YHWL YLGL YLFL

Clé 3 → **VENI VIDI VICI**

La Cryptographie Symétrique

Moderne

Concept Fondamental

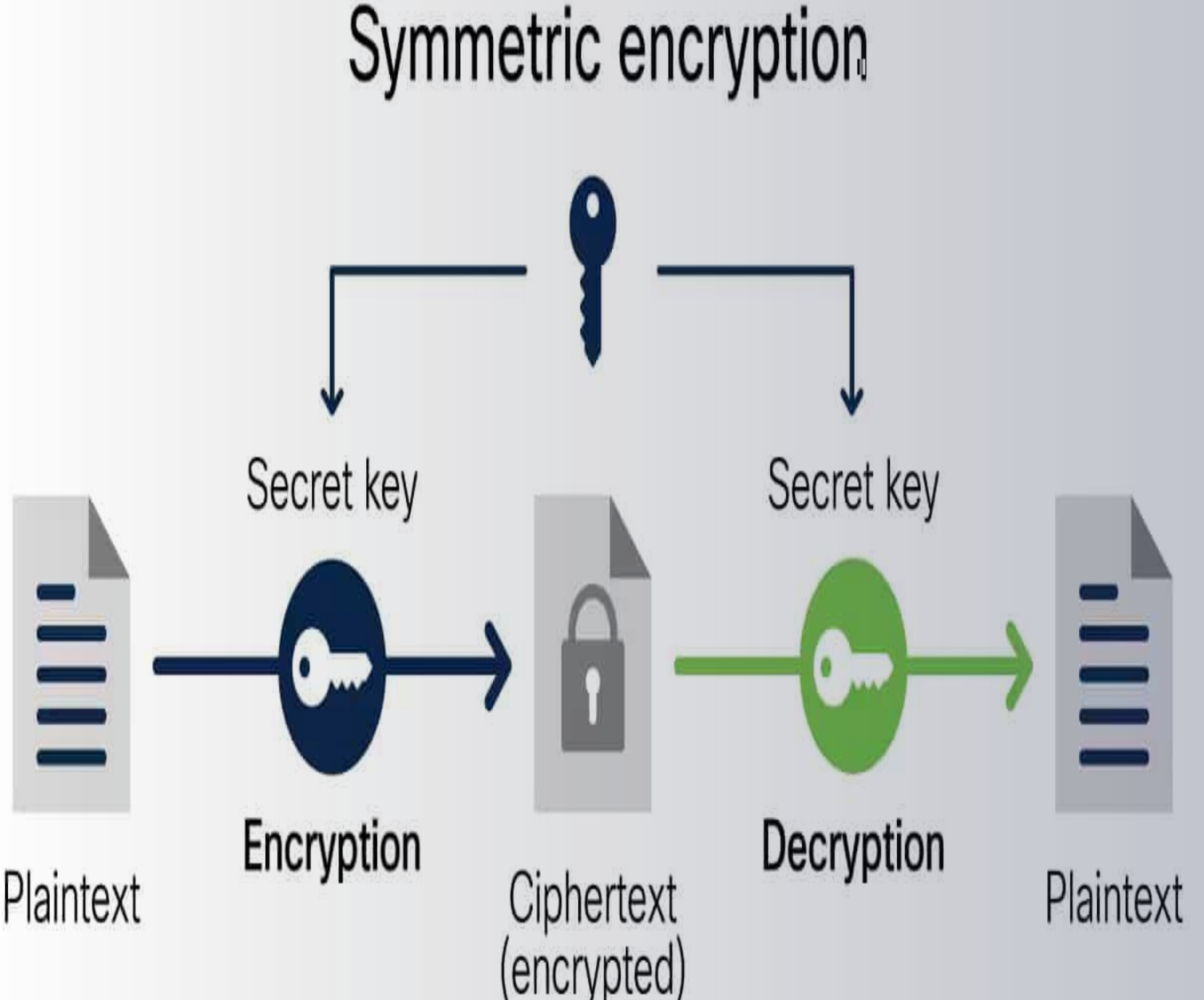
Une **SEULE clé** partagée pour chiffrer ET déchiffrer

Analogie Simple

La même clé verrouille et déverrouille votre porte

Problème Majeur

Comment **transmettre la clé** de manière sécurisée ?



Les Familles d'Algorithmes Symétriques

Stream Ciphers

Principe

Génère une suite pseudo-aléatoire (keystream) → XOR bit à bit avec le message

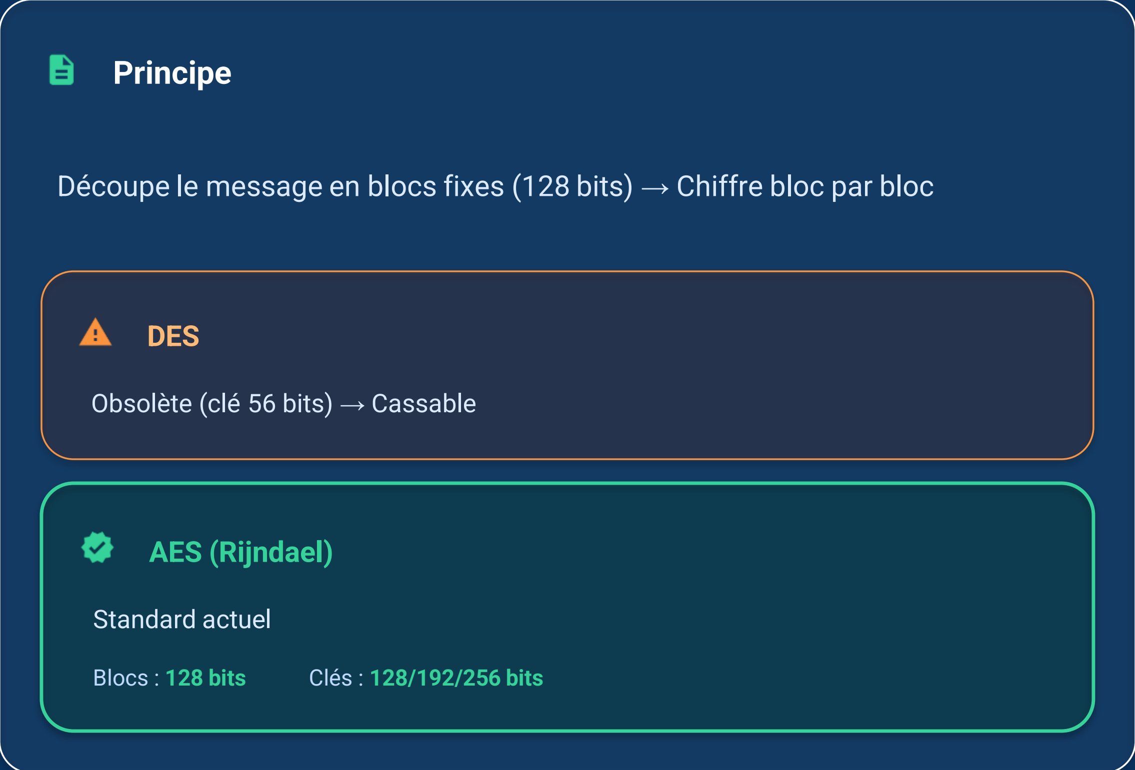
RC4

Cassé et interdit → Ne pas utiliser

ChaCha20

Moderne, rapide, sécurisé → Google, Android

Block Ciphers



ChaCha20 : Communications temps réel, performances mobile

AES : Stockage de fichiers, données structurées

Le Piège Critique

Les Modes d'Opération

Le Problème

AES seul chiffre **un seul bloc** de 128 bits

Pour des messages longs, il faut appliquer AES sur plusieurs blocs

! Même importance que l'algorithme

Un mauvais mode peut rendre **inutile** le meilleur algorithme

ECB (Electronic Code Book)

Chiffre chaque bloc **indépendamment**

Défaut : **Motifs visibles** dans le chiffré

Exemple : Pingouin BMP reste **reconnaissable**

CBC (Cipher Block Chaining)

Chaque bloc **chaîne** avec le précédent

Nécessite un **IV aléatoire** unique

Avantage : **Bonne diffusion**, un bloc change tout

GCM (Galois/Counter Mode)

Chiffrement + **Authentification** (AEAD)

Très rapide grâce à optimisations mathématiques

Fournit **confidentialité ET intégrité**

ECB vs CBC vs GCM

Comprendre la Différence

Le Test Pingouin

Chiffrer une image (ex: logo Tux) révèle la vérité

ECB

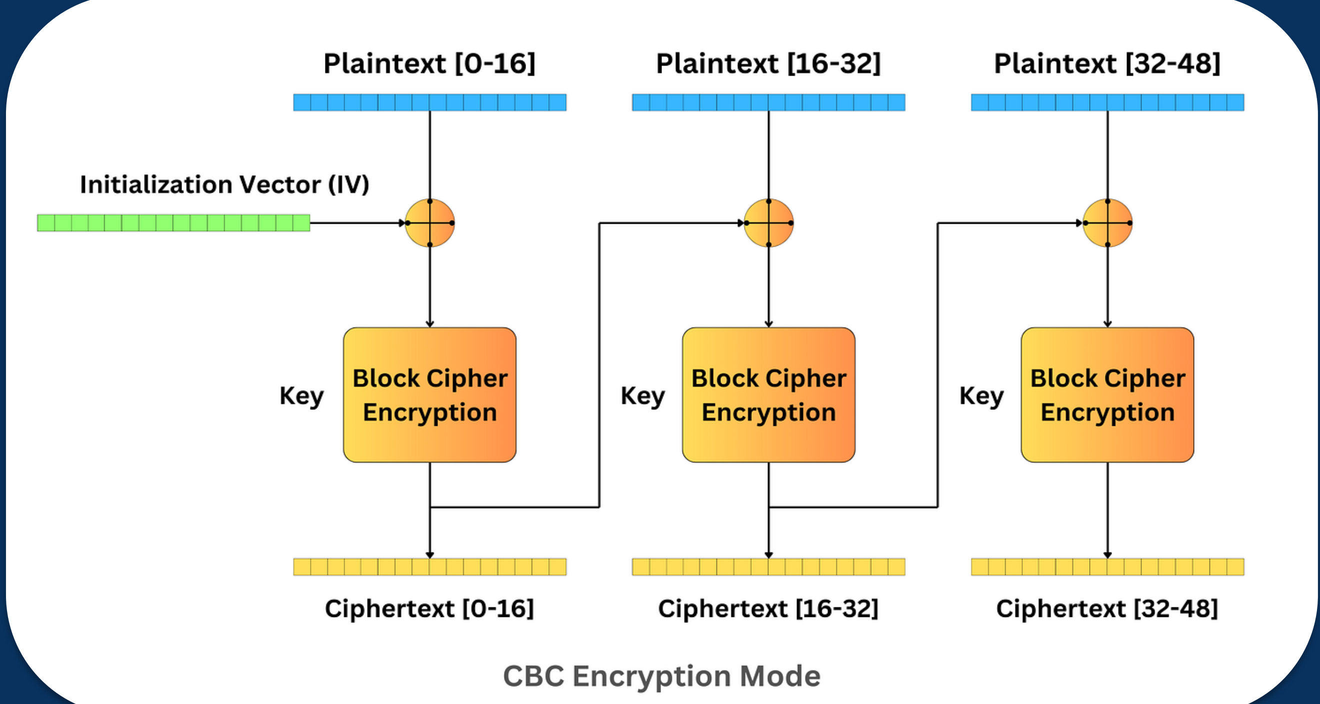
Contours visibles → Pas de sécurité

CBC

Bruit aléatoire → Bonne diffusion

GCM

Bruit + Authentification → Le meilleur



Conclusion Pratique

Jamais ECB - Motifs visibles, pas de sécurité réelle

CBC acceptable - Nécessite gestion rigoureuse des IVs

GCM recommandé - Chiffrement + authentification (AEAD)

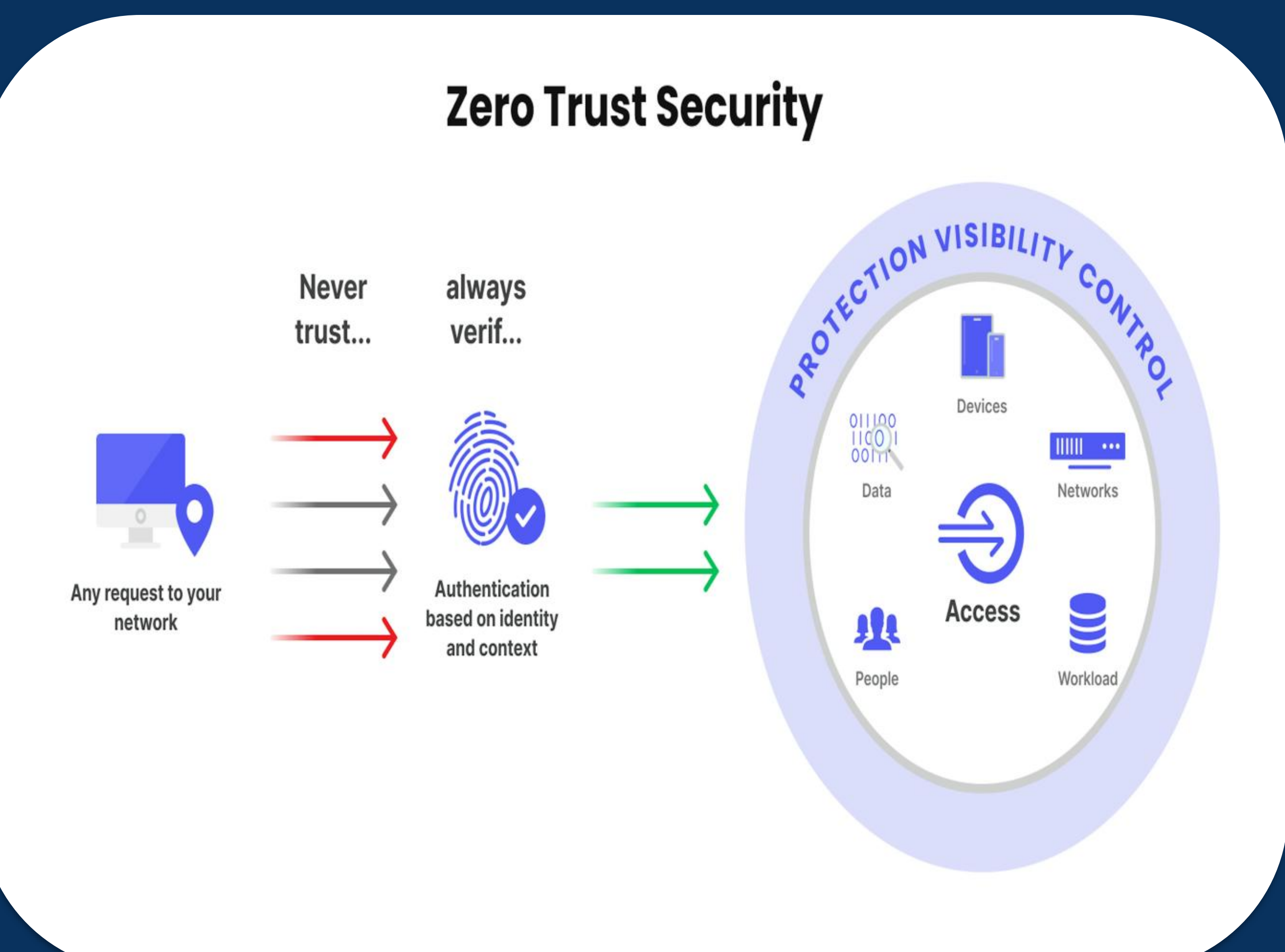
Bonnes Pratiques

À FAIRE

À ÉVITER

- **Standards audités** AES-GCM, ChaCha20-Poly1305
- **Mode authentifié** AEAD (GCM, ChaCha20-Poly1305)
- **IVs uniques** Génération sécurisée pour chaque chiffrage
- **Dérivation robuste** PBKDF2/Argon2 + salt
- **Bibliothèques éprouvées** OpenSSL, libsodium, cryptography

- Inventer votre algorithme
- Votre propre implémentation
- Utiliser **ECB**
- Réutiliser un IV
- Mots de passe comme clé
- Algorithmes obsolètes DES, RC4, MD5



Résumé

Points Clés à Retenir

Prochaine Étape

Distribution des clés & Cryptographie asymétrique

- → Diffie-Hellman
- → RSA

1 Triade CIA

Confidentialité, Intégrité, Disponibilité

2 Kerckhoffs

Clé secrète, algorithme public

3 Leçons Histoire

Force brute, analyse fréquentielle

4 AES Standard

Blocs 128 bits, clés 128/192/256

5 Mode Critique

ECB bannir, CBC OK, GCM recommandé

6 Bonnes Pratiques

AEAD, IVs uniques, PBKDF2/Argon2

Utilisez des standards testés

Et ajoutez toujours l'authentification (MAC/AEAD)

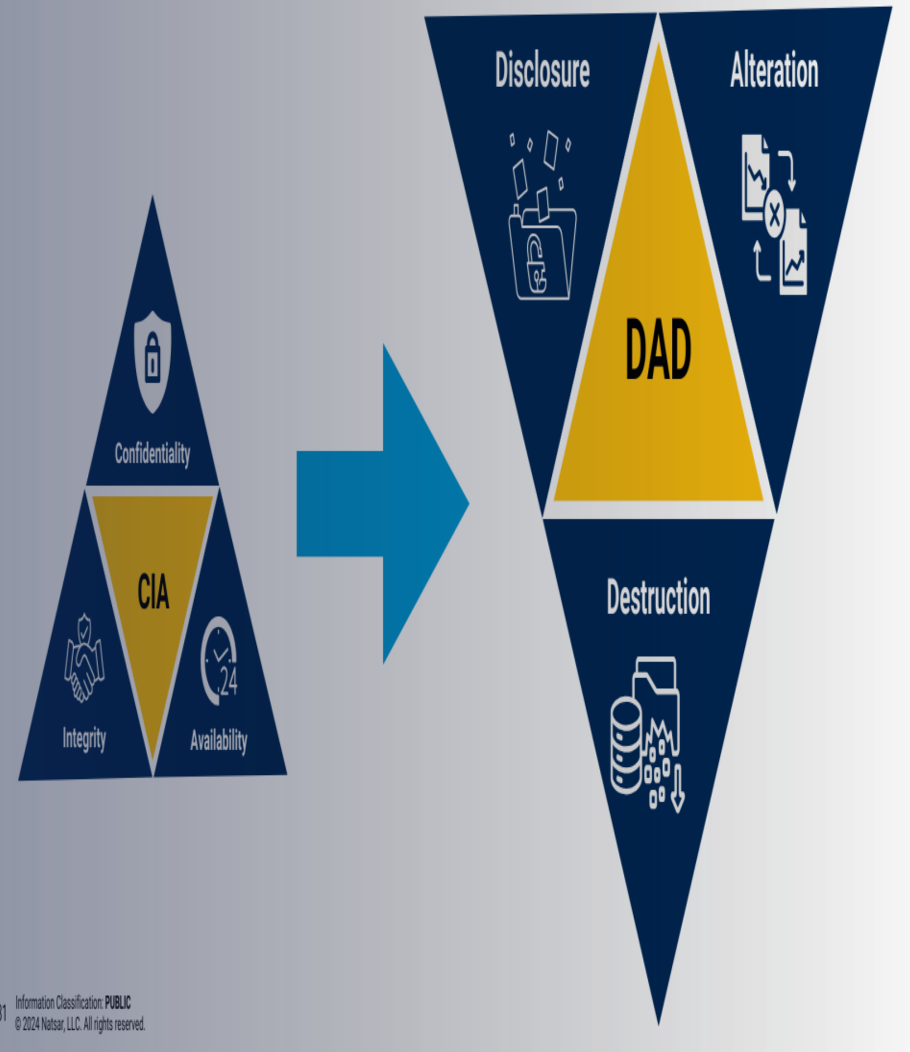
Fondamentaux de la Sécurité et Cryptographie

Partie A - Les Fondations et la Cryptographie Symétrique

Objectif : Comprendre pourquoi on n'invente pas son propre algorithme, maîtriser AES et l'importance critique des modes d'opération

The Foundation of Cybersecurity: CIA

CIA protects against DAD



La Triade CIA

Le socle de la cybersécurité

Confidentialité

Seules les personnes autorisées lisent les données

Ex : Chiffrement AES pour emails et communications

Intégrité

Les données n'ont pas été modifiées

Ex : Hash SHA-256, signatures numériques

Disponibilité

Le système fonctionne quand on en a besoin

Ex : Protection DDoS, sauvegardes redondantes, CDN

Bonus : Non-Répudiation

Empêche de nier avoir effectué une action

Ex : Signature électronique, journaux d'audit

alamy
Image ID: 2EEEK31
www.alamy.com

Vocabulaire Clé et Règle d'Or

Les Termes Essentiels

Cryptographie
L'art de coder les messages

Cryptanalyse
L'art de casser un code sans clé

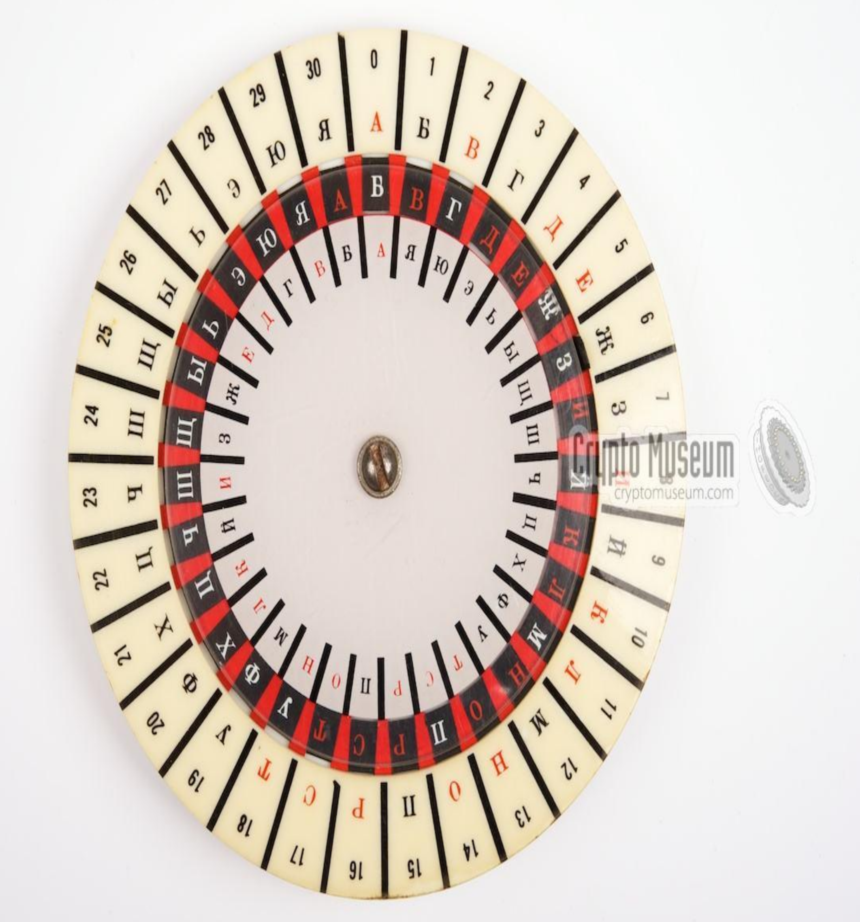
Plaintext
Message lisible (texte clair)

Ciphertext
Message illisible (chiffré)

LE PRINCIPE DE KERCKHOFFS (1883)

La sécurité ne doit reposer que sur le **secret de la clé**, pas sur celui de l'algorithme

→ L'algorithme doit être public, audité et standardisé
→ "Security by Obscurity" ne fonctionne jamais



L'Histoire : César et Analyse Fréquentielle

Chiffre de César

Principe : Décalage de l'alphabet

Formule : C = (P + k) mod 26

⚠ **Problème :** Seulement 25 clés possibles → Cassable en **1 milliseconde**

Analyse Fréquentielle

Les langues ont une signature statistique

En français : E ≈ **15%** A ≈ **8%** S ≈ **7%**

Si un symbole apparaît 15% dans le chiffré → Probablement = E

LEÇON CRITIQUE

Un bon chiffrement doit **détruire** les statistiques du langage

Exemple : YHWL YLGL YLFL

Clé 3 → **VENI VIDI VICI**

La Cryptographie Symétrique

Moderne

Concept Fondamental

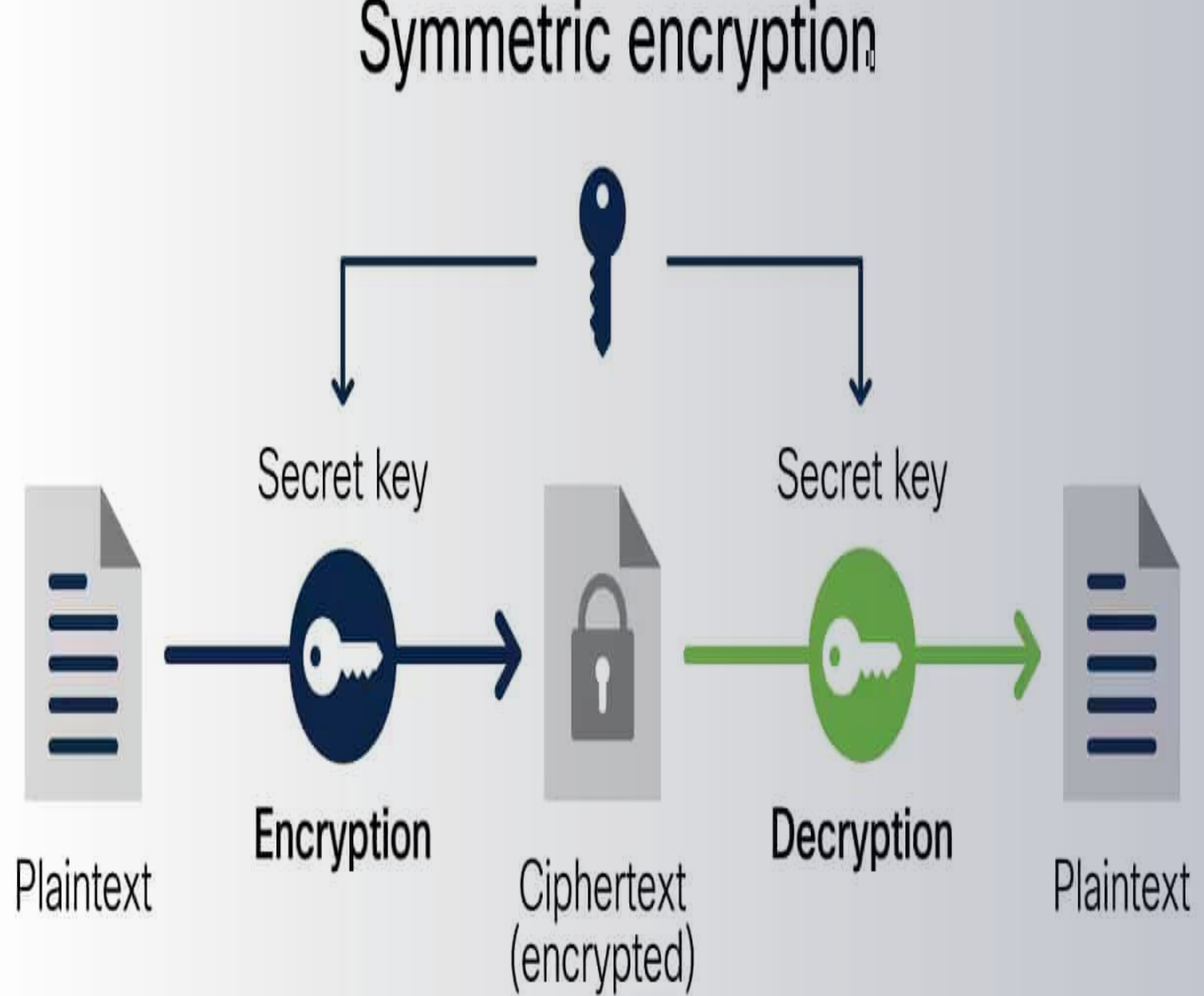
Une **SEULE clé** partagée pour chiffrer ET déchiffrer

Analogie Simple

La même clé verrouille et déverrouille votre porte

Problème Majeur

Comment **transmettre la clé** de manière sécurisée ?



Les Familles d'Algorithmes Symétriques

Stream Ciphers

Principe

Génère une suite pseudo-aléatoire (keystream) → XOR bit à bit avec le message

RC4

Cassé et interdit → Ne pas utiliser

ChaCha20

Moderne, rapide, sécurisé → Google, Android

Block Ciphers

ChaCha20 : Communications temps réel, performances mobile

AES : Stockage de fichiers, données structurées

Le Piège Critique

Les Modes d'Opération

Le Problème

AES seul chiffre **un seul bloc** de 128 bits

Pour des messages longs, il faut appliquer AES sur plusieurs blocs

! Même importance que l'algorithme

Un mauvais mode peut rendre **inutile** le meilleur algorithme

ECB (Electronic Code Book)

Chiffre chaque bloc **indépendamment**

Défaut : **Motifs visibles** dans le chiffré

Exemple : Pingouin BMP reste **reconnaissable**

CBC (Cipher Block Chaining)

Chaque bloc **chaîne** avec le précédent

Nécessite un **IV aléatoire** unique

Avantage : **Bonne diffusion**, un bloc change tout

GCM (Galois/Counter Mode)

Chiffrement + **Authentification** (AEAD)

Très rapide grâce à optimisations mathématiques

Fournit **confidentialité ET intégrité**

ECB vs CBC vs GCM

Comprendre la Différence

Le Test Pingouin

Chiffrer une image (ex: logo Tux) révèle la vérité

ECB

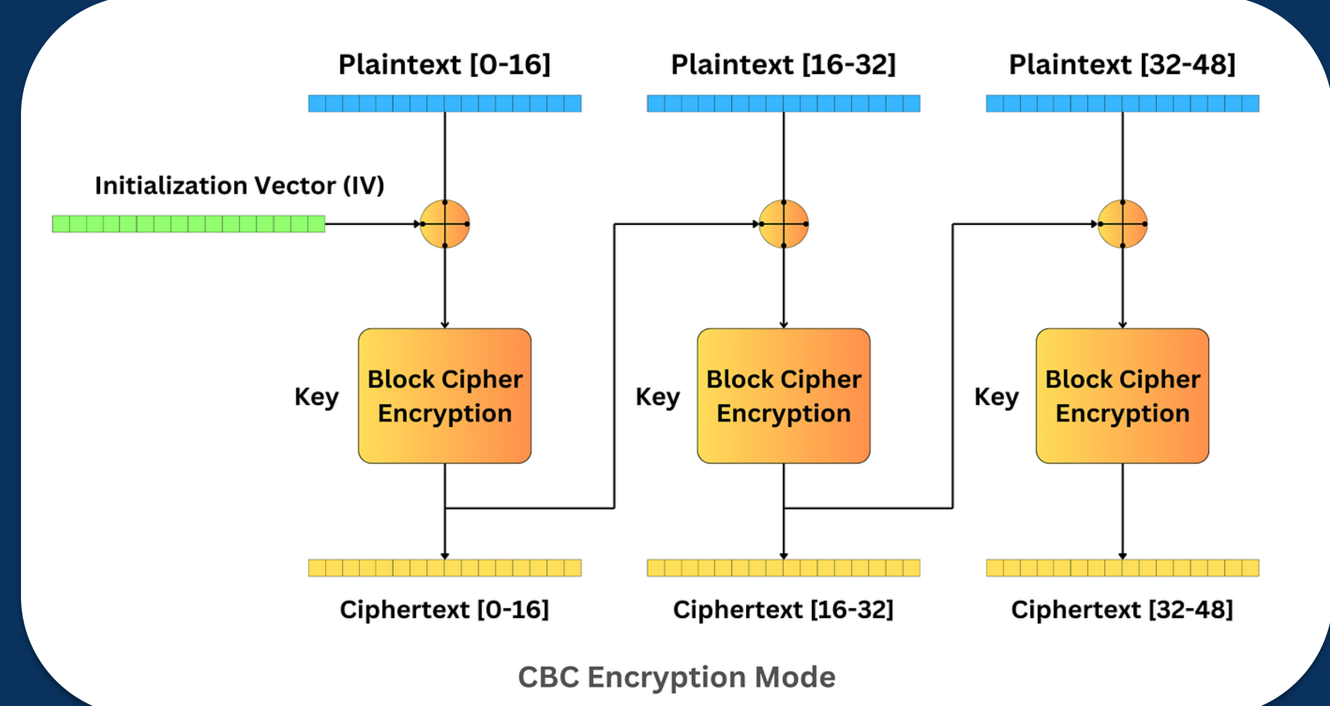
Contours visibles → Pas de sécurité

CBC

Bruit aléatoire → Bonne diffusion

GCM

Bruit + Authentification → Le meilleur



Conclusion Pratique

Jamais ECB - Motifs visibles, pas de sécurité réelle

CBC acceptable - Nécessite gestion rigoureuse des IVs

GCM recommandé - Chiffrement + authentification (AEAD)

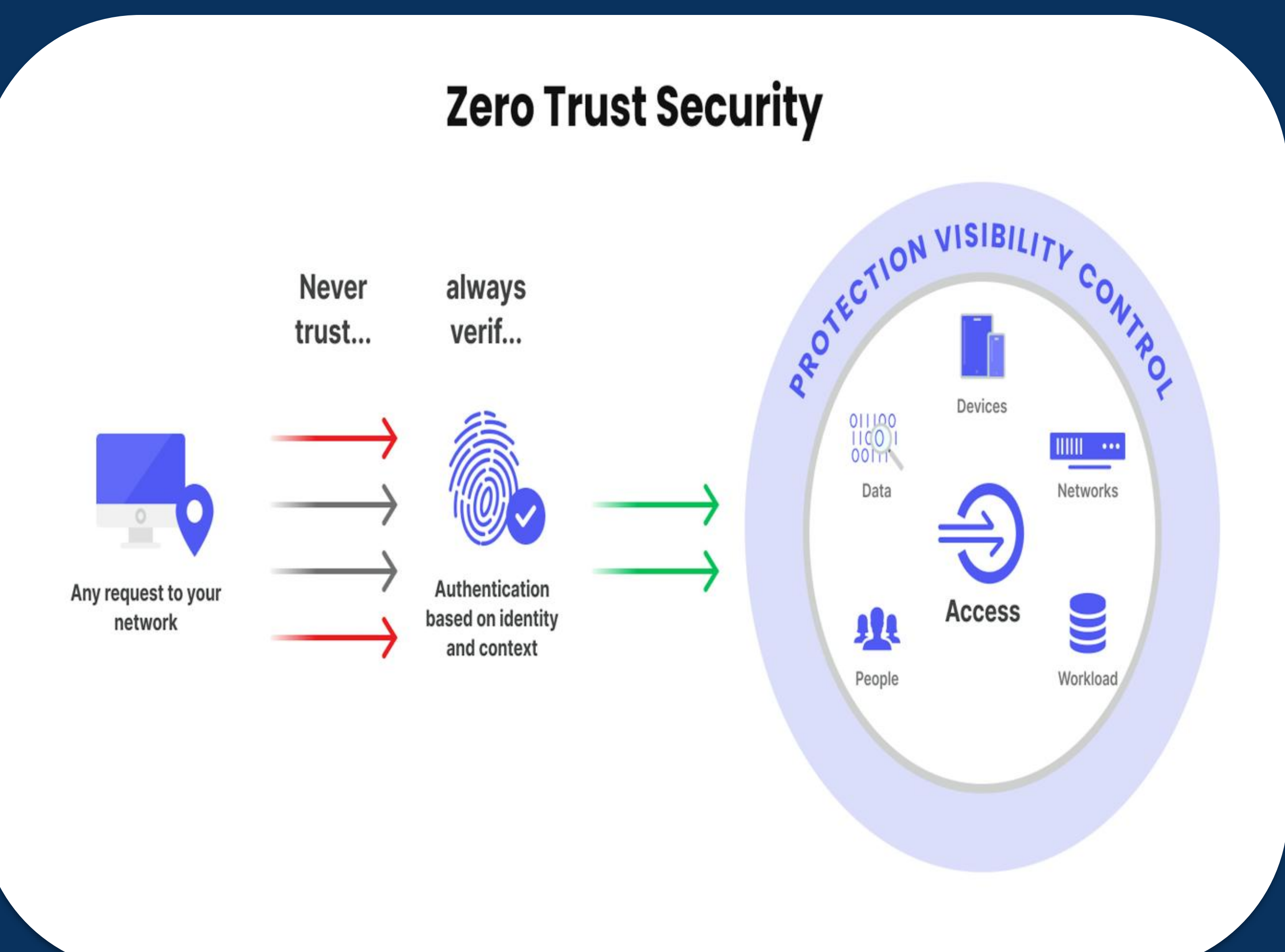
Bonnes Pratiques

À FAIRE

- **Standards audités**
  AES-GCM, ChaCha20-Poly1305
- **Mode authentifié**
  AEAD (GCM, ChaCha20-Poly1305)
- **IVs uniques**
  Génération sécurisée pour chaque chiffrage
- **Dérivation robuste**
  PBKDF2/Argon2 + salt
- **Bibliothèques éprouvées**
  OpenSSL, libsodium, cryptography

À ÉVITER

- Inventer votre algorithme
- Votre propre implémentation
- Utiliser **ECB**
- Réutiliser un IV
- Mots de passe comme clé
- Algorithmes obsolètes
  DES, RC4, MD5



Résumé

Points Clés à Retenir

Prochaine Étape

Distribution des clés & Cryptographie asymétrique

- → Diffie-Hellman
- → RSA

1

Triade CIA

Confidentialité, Intégrité, Disponibilité

2

Kerckhoffs

Clé secrète, algorithme public

3

Leçons Histoire

Force brute, analyse fréquentielle

4

AES Standard

Blocs 128 bits, clés 128/192/256

5

Mode Critique

ECB bannir, CBC OK, GCM recommandé

6

Bonnes Pratiques

AEAD, IVs uniques, PBKDF2/Argon2

Utilisez des standards testés

Et ajoutez toujours l'authentification (MAC/AEAD)

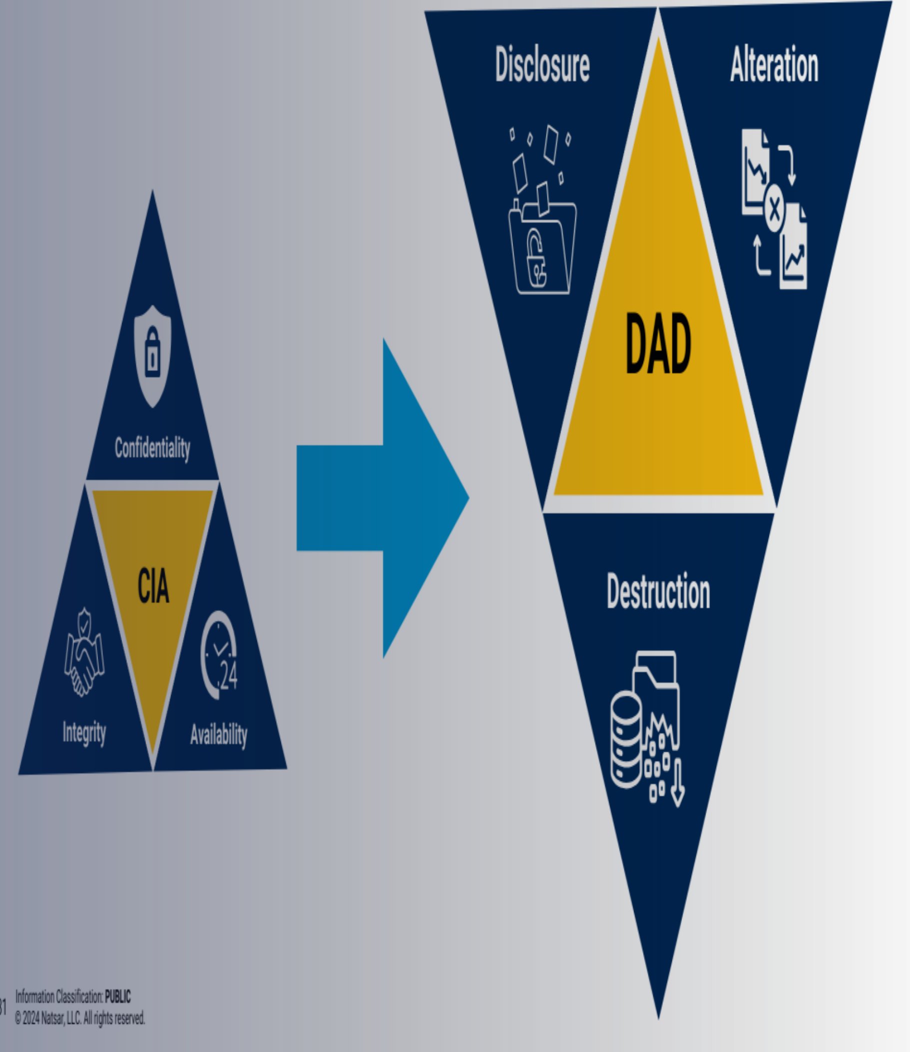
Fondamentaux de la Sécurité et Cryptographie

Partie A - Les Fondations et la Cryptographie Symétrique

Objectif : Comprendre pourquoi on n'invente pas son propre algorithme, maîtriser AES et l'importance critique des modes d'opération

The Foundation of Cybersecurity: CIA

CIA protects against DAD



La Triade CIA

Le socle de la cybersécurité

Confidentialité

Seules les personnes autorisées lisent les données

Ex : Chiffrement AES pour emails et communications

Intégrité

Les données n'ont pas été modifiées

Ex : Hash SHA-256, signatures numériques

Disponibilité

Le système fonctionne quand on en a besoin

Ex : Protection DDoS, sauvegardes redondantes, CDN

Bonus : Non-Répudiation

Empêche de nier avoir effectué une action

Ex : Signature électronique, journaux d'audit

alamy
Image ID: 2EEEK31
www.alamy.com

Vocabulaire Clé et Règle d'Or

Les Termes Essentiels

Cryptographie
L'art de coder les messages

Cryptanalyse
L'art de casser un code sans clé

Plaintext
Message lisible (texte clair)

Ciphertext
Message illisible (chiffré)

LE PRINCIPE DE KERCKHOFFS (1883)

La sécurité ne doit reposer que sur le **secret de la clé**, pas sur celui de l'algorithme

→ L'algorithme doit être public, audité et standardisé
→ "Security by Obscurity" ne fonctionne jamais

L'Histoire : César et Analyse Fréquentielle

Chiffre de César

Principe : Décalage de l'alphabet

Formule : C = (P + k) mod 26

⚠ **Problème :** Seulement 25 clés possibles → Cassable en **1 milliseconde**

Analyse Fréquentielle

Les langues ont une signature statistique

En français : E ≈ **15%** A ≈ **8%** S ≈ **7%**

Si un symbole apparaît 15% dans le chiffré → Probablement = E

LEÇON CRITIQUE

Un bon chiffrement doit **détruire** les statistiques du langage

Exemple : YHWL YLGL YLFL

Clé 3 → **VENI VIDI VICI**

La Cryptographie Symétrique

Moderne

Concept Fondamental

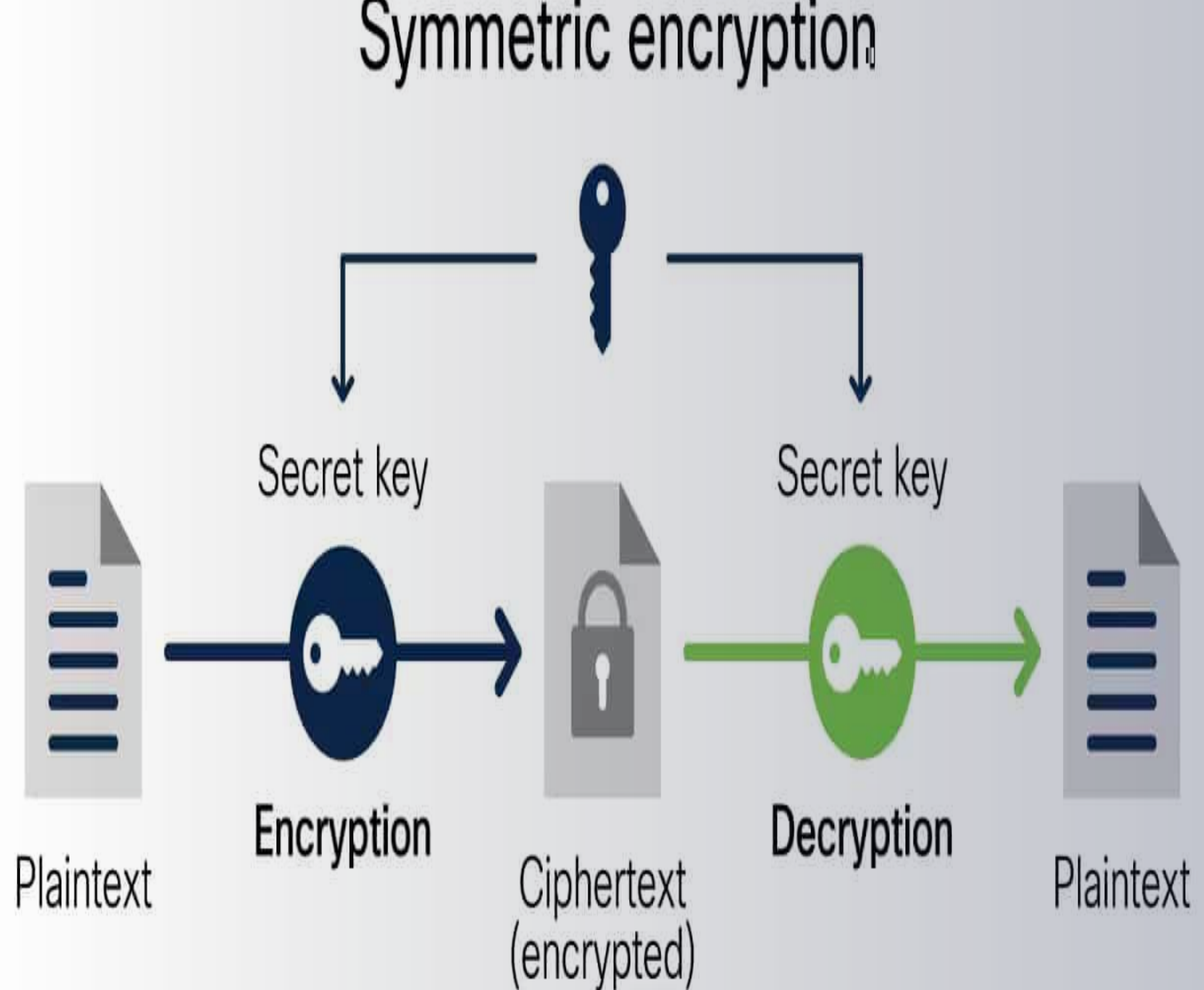
Une **SEULE clé** partagée pour chiffrer ET déchiffrer

Analogie Simple

La même clé verrouille et déverrouille votre porte

Problème Majeur

Comment **transmettre la clé** de manière sécurisée ?



Les Familles d'Algorithmes Symétriques

Stream Ciphers

Principe

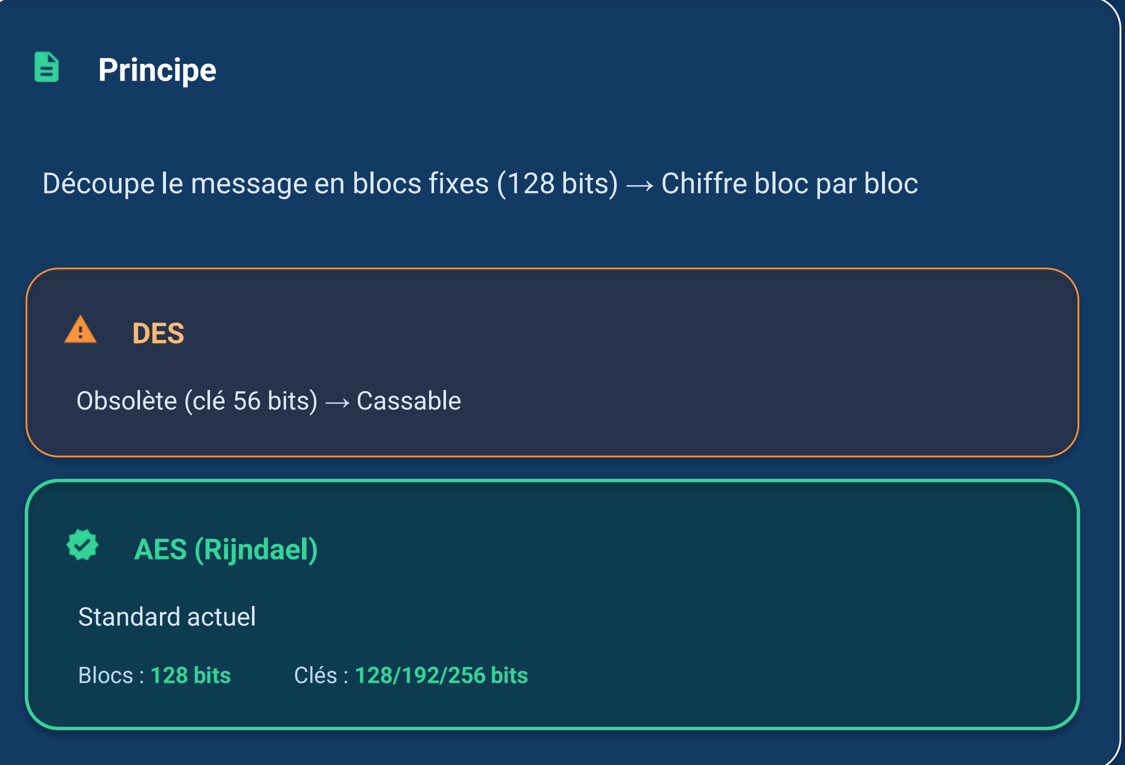
Génère une suite pseudo-aléatoire (keystream) → XOR bit à bit avec le message

RC4

Cassé et interdit → Ne pas utiliser

ChaCha20

Moderne, rapide, sécurisé → Google, Android



ChaCha20 : Communications temps réel, performances mobile

AES : Stockage de fichiers, données structurées

Le Piège Critique

Les Modes d'Opération

Le Problème

AES seul chiffre **un seul bloc** de 128 bits

Pour des messages longs, il faut appliquer AES sur plusieurs blocs

! Même importance que l'algorithme

Un mauvais mode peut rendre **inutile** le meilleur algorithme

ECB (Electronic Code Book)

Chiffre chaque bloc **indépendamment**

Défaut : **Motifs visibles** dans le chiffré

Exemple : Pingouin BMP reste **reconnaissable**

CBC (Cipher Block Chaining)

Chaque bloc **chaîne** avec le précédent

Nécessite un **IV aléatoire** unique

Avantage : **Bonne diffusion**, un bloc change tout

GCM (Galois/Counter Mode)

Chiffrement + **Authentification** (AEAD)

Très rapide grâce à optimisations mathématiques

Fournit **confidentialité ET intégrité**

ECB vs CBC vs GCM

Comprendre la Différence

Le Test Pingouin

Chiffrer une image (ex: logo Tux) révèle la vérité

ECB

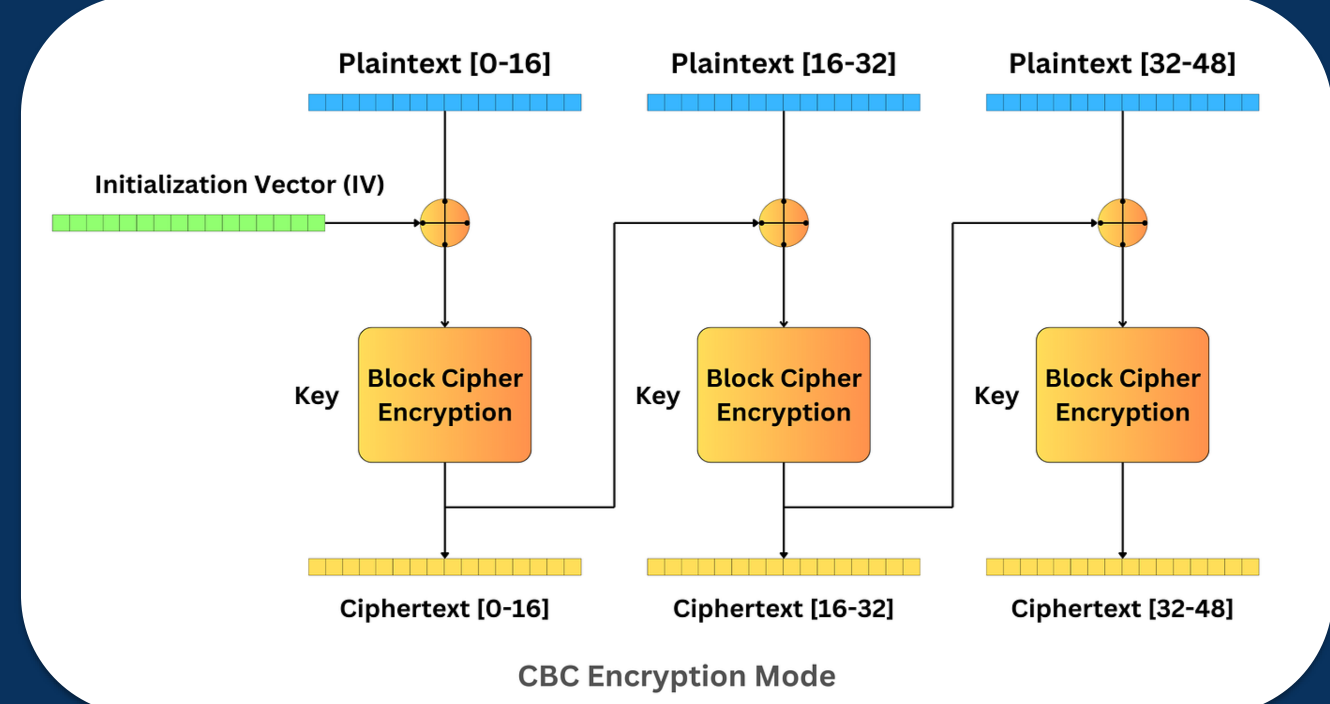
Contours visibles → Pas de sécurité

CBC

Bruit aléatoire → Bonne diffusion

GCM

Bruit + Authentification → Le meilleur



Conclusion Pratique

Jamais ECB - Motifs visibles, pas de sécurité réelle

CBC acceptable - Nécessite gestion rigoureuse des IVs

GCM recommandé - Chiffrement + authentification (AEAD)

Bonnes Pratiques

À FAIRE

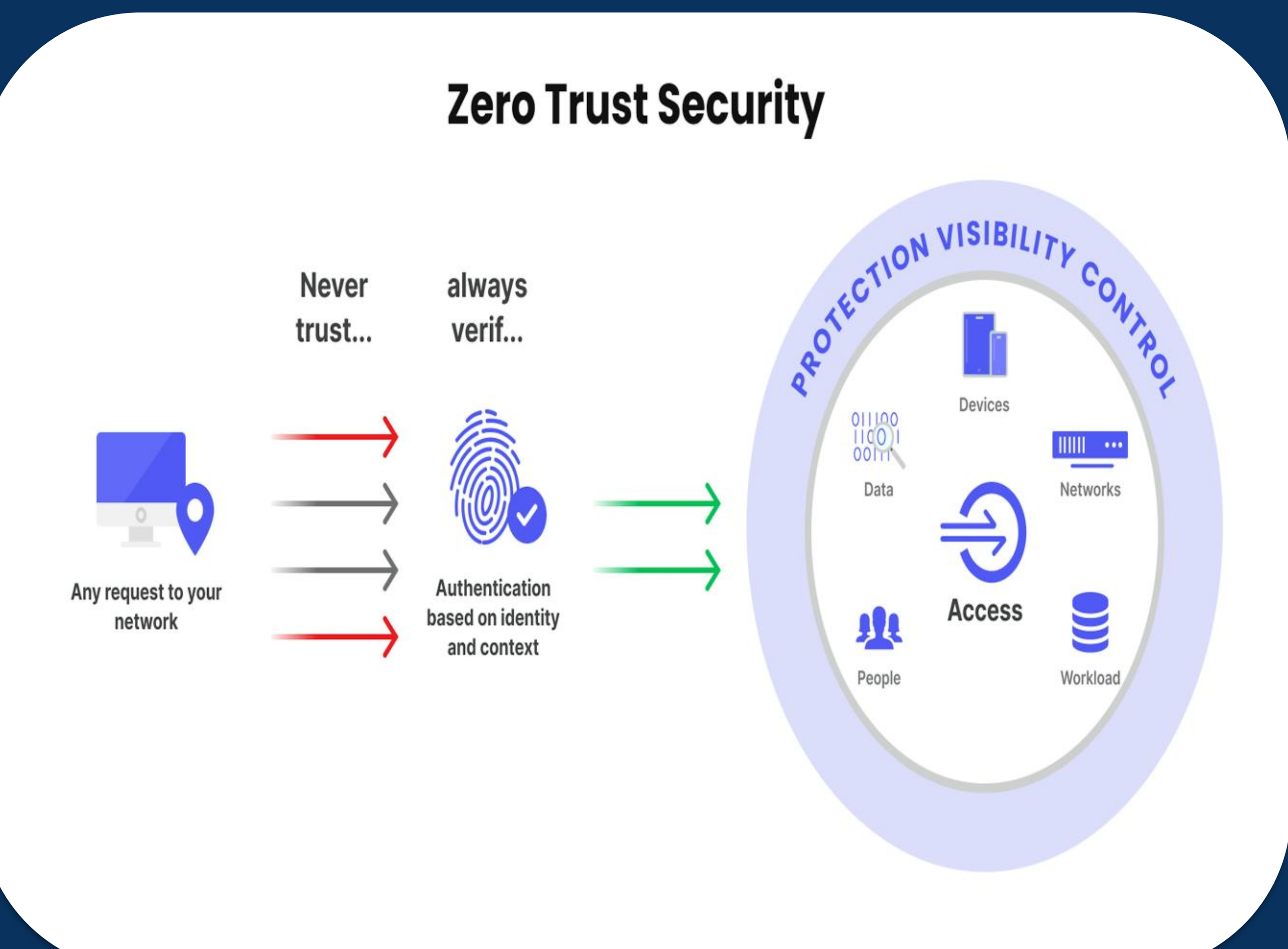
À ÉVITER

À FAIRE

- **Standards audités**
  AES-GCM, ChaCha20-Poly1305
- **Mode authentifié**
  AEAD (GCM, ChaCha20-Poly1305)
- **IVs uniques**
  Génération sécurisée pour chaque chiffrage
- **Dérivation robuste**
  PBKDF2/Argon2 + salt
- **Bibliothèques éprouvées**
  OpenSSL, libsodium, cryptography

À ÉVITER

- Inventer votre algorithme
- Votre propre implémentation
- Utiliser **ECB**
- Réutiliser un IV
- Mots de passe comme clé
- Algorithmes obsolètes
  DES, RC4, MD5



Résumé

Points Clés à Retenir

Prochaine Étape

Distribution des clés & Cryptographie asymétrique

- → Diffie-Hellman
- → RSA

1 Triade CIA

Confidentialité, Intégrité, Disponibilité

2 Kerckhoffs

Clé secrète, algorithme public

3 Leçons Histoire

Force brute, analyse fréquentielle

4 AES Standard

Blocs 128 bits, clés 128/192/256

5 Mode Critique

ECB bannir, CBC OK, GCM recommandé

6 Bonnes Pratiques

AEAD, IVs uniques, PBKDF2/Argon2

Utilisez des standards testés

Et ajoutez toujours l'authentification (MAC/AEAD)

Travaux Pratiques

Sécurité et Cryptographie

Fondamentaux de la Cryptographie

dreamstime.com

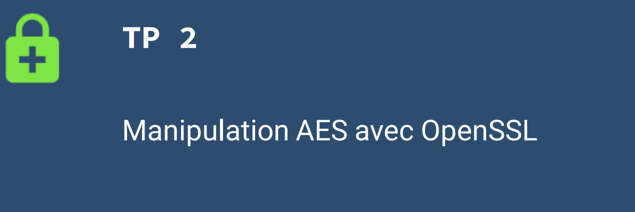
ID 396402075 © Helga Panji Setiawan

Introduction aux Travaux Pratiques

Les 4 Exercices Pratiques

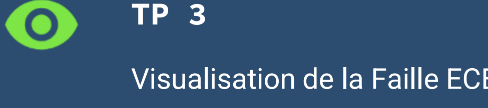
TP 1

Attaque par Force Brute (César)



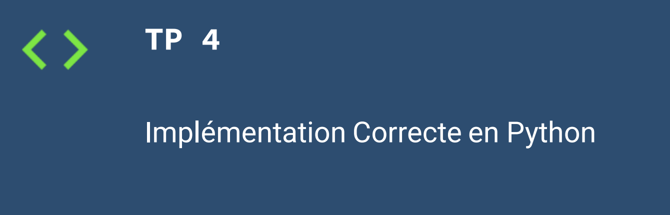
TP 2

Manipulation AES avec OpenSSL



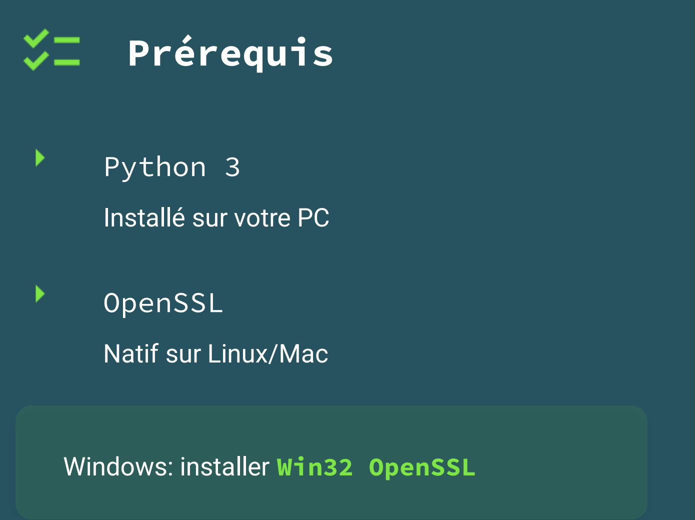
TP 3

Visualisation de la Faille ECB



TP 4

Implémentation Correcte en Python



Prérequis

- `Python 3`
  Installé sur votre PC
- `OpenSSL`
  Natif sur Linux/Mac

Windows: installer `Win32 OpenSSL`

🔍 TP 1 : Attaque par Force Brute (Python)

Scénario

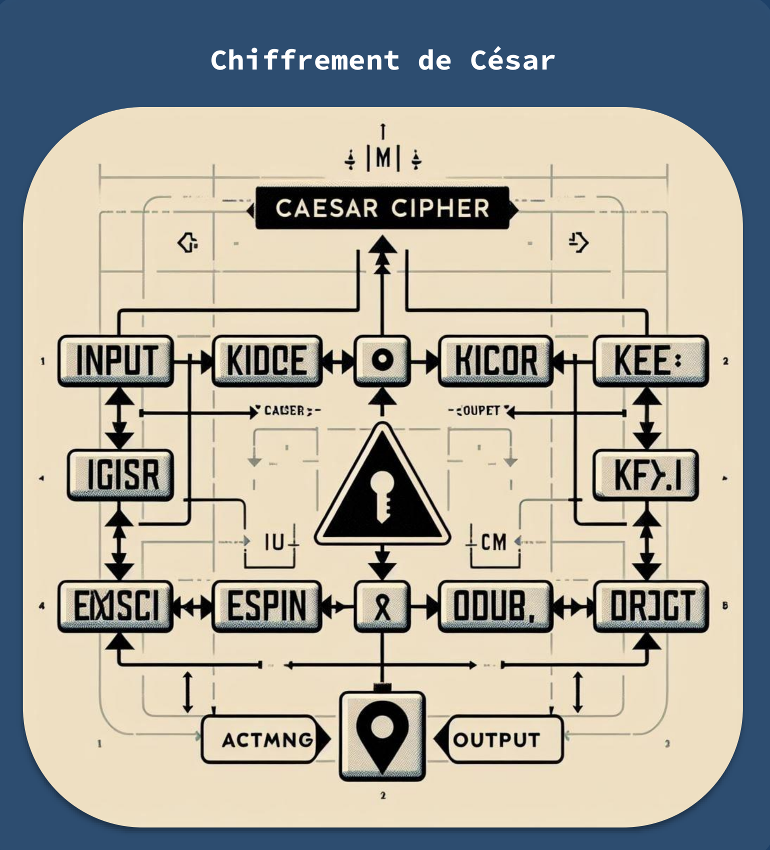
Vous interceptez un message militaire romain chiffré par César : **"YHWL YLGL YLFL"**

Retrouvez le texte clair en testant les 26 décalages possibles.

Code Python (squelette)

```
# Message chiffré
message = "YHWL YLGL YLFL"
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
for cle in range(1, 26):
resultat = ""
for lettre in message:
if lettre in ALPHABET:
# TODO : Appliquer la formule inverse
# index = (ALPHABET.index(lettre) - cle) % 26
# resultat += ALPHABET[index]
else:
resultat += lettre
print(f"Clé {cle}: {resultat}")
```

Chiffrement de César



Formule de déchiffrement

$P = (C - k) \bmod 26$

Solution TP 1

Code Python Complet

```
# Message chiffré
message = "YHWL YLGL YLFL"
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
for cle in range(1, 26):
resultat = ""
for lettre in message:
if lettre in ALPHABET:
index = (ALPHABET.index(lettre) - cle) % 26
resultat += ALPHABET[index]
else:
resultat += lettre
print(f"Clé {cle}: {resultat}")
```

L'attaque par force brute teste systématiquement **toutes les clés possibles** (25 décalages en français)

Résultat Attendu

Clé 3 :

VENI VIDI VICI

"Je suis venu, j'ai vu, j'ai vaincu"

— Jules César

25

Décalages possibles

1

Solution unique

Windows PowerShell

GNU nano 8.4 caesar_brute_force.py

```
message = "YHWL YLGL YLFL"
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

for key in range(1, 26):
    result = ""
    for letter in message:
        if letter in ALPHABET:
            index = (ALPHABET.index(letter) - key) % 26
            result += ALPHABET[index]
        else:
            result += letter
    print(f"Key {key}: {result}")
```

[Lecture de 12 lignes]

^G Aide ^O Écrire ^F Chercher ^K Couper ^T Exécuter ^C Emplacement M-U Annuler
^X Quitter ^R Lire fich. ^\ Remplacer ^U Coller ^J Justifier ^/ Aller ligne M-E Refaire

Windows PowerShell

```
Successfully installed cffi-2.0.0 cryptography-46.0.3 pycparser-2.23
(venv) root@Debian11-bouselama:~# nano caesar_brute_force.py
(venv) root@Debian11-bouselama:~# python caesar_brute_force.py
Key 1: XGVK XKFK XKEK
Key 2: WFUJ WJEJ WJDJ
Key 3: VETI VIDI VICI
Key 4: UDSH UHCH UHBH
Key 5: TCRG TGBG TGAG
Key 6: SBQF SFAF SFZF
Key 7: RAPE REZE REYE
Key 8: QZOD QDYD QDXD
Key 9: PYNC PCXC PCWC
Key 10: OXMB OBWB OBVB
Key 11: NWLA NAVA NAUA
Key 12: MVKZ MZUZ MZTZ
Key 13: LUJY LYTY LYSY
Key 14: KTIX KXSX KXRX
Key 15: JSHW JWRW JWQW
Key 16: IRGV IVQV IVPV
Key 17: HQFU HUPU HUOU
Key 18: GPET GTOT GTNT
Key 19: FODS FSNS FSMS
Key 20: ENCR ERMR ERLR
Key 21: DMBQ DQLQ DQKQ
Key 22: CLAP CPKP CPJP
Key 23: BKZO BOJO BOIO
Key 24: AJYN ANIN ANHN
Key 25: ZIXM ZMHM ZMGM
```

TP 2 : Le Secret du Salage

Expérience

Chiffrer **deux fois** le même fichier avec le **même mot de passe**

Étape 1 : Chiffrer le premier fichier

```
openssl enc -aes-256-cbc -in secret.txt -out secret1.enc -pbkdf2
```

Étape 2 : Chiffrer le deuxième fichier (même password)

```
openssl enc -aes-256-cbc -in secret.txt -out secret2.enc -pbkdf2
```

Étape 3 : Comparer les fichiers

```
# Comparaison avec md5sum ou diff
md5sum secret1.enc secret2.enc
ou
diff secret1.enc secret2.enc
```

Observation

Les fichiers sont DIFFÉRENTS !

Malgré le même mot de passe

Explication

OpenSSL ajoute automatiquement un **salage aléatoire** (Salt) à chaque chiffrement

Protection

Rainbow Tables

Aléatoire

À chaque fois

```
                                              uc◆◆2cat: ds: Aucun fichier ou dossier de ce nom
(venv) root@Debian11-bouselama:~# diff secret.enc secret2.enc
1c1
< Salted__#*#'◆◆aL◆v◆◆ʕ◆:◆:'g◆◆PRõB◆◆%◆◆C◆◆◆u
\ Pas de fin de ligne à la fin du fichier
---
> Salted__◆◆◆{◆◆:=mWk:j◆◆>◆X-◆#^)◆◆G◆$◆
                                        uc◆◆2
\ Pas de fin de ligne à la fin du fichier
(venv) root@Debian11-bouselama:~# |
```

Windows PowerShell

```
Verify failure
bad password read
405744A8CC7F0000:error:1400006B:UI routines:UI_process:processing error:../crypto/ui/ui_lib.c:552:while
 reading strings
(venv) root@Debian11-bouselama:~# openssl enc -aes-256-cbc -in secret.txt -out secret.enc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
(venv) root@Debian11-bouselama:~# ls
caesar_brute_force.py           King-infra.lan.+008+21898.private  secret.txt
King-infra.lan.+008+21898.key  secret.enc                          security-tps
(venv) root@Debian11-bouselama:~# cat secret.enc
Salted__#*#'◆◆aL◆v◆◆ʕ◆:◆:'g◆◆PRõB◆◆%◆◆C◆◆◆u(venv) root@Debian11-bouselama:~# openssl enc -d -aes-256-cb
c -in secret.enc -out decrypt.txt openssl enc -d -aes-256-cbc -in secret.enc -out decrypt.txt -pbkdf2
enter AES-256-CBC decryption password:
(venv) root@Debian11-bouselama:~# ls
caesar_brute_force.py           King-infra.lan.+008+21898.private  secret.txt
King-infra.lan.+008+21898.key  secret.enc                          security-tps
(venv) root@Debian11-bouselama:~# cat secret.enc
Salted__#*#'◆◆aL◆v◆◆ʕ◆:◆:'g◆◆PRõB◆◆%◆◆C◆◆◆u(venv) root@Debian11-bouselama:~# openssl enc -d -aes-256-cb
c -in secret.enc -out decrypt.txt -pbkdf2
enter AES-256-CBC decryption password:
bad password read
(venv) root@Debian11-bouselama:~# openssl enc -d -aes-256-cbc -in secret.enc -out decrypt.txt -pbkdf2
enter AES-256-CBC decryption password:
(venv) root@Debian11-bouselama:~# ls
caesar_brute_force.py  King-infra.lan.+008+21898.key      secret.enc  security-tps
decrypt.txt            King-infra.lan.+008+21898.private  secret.txt
(venv) root@Debian11-bouselama:~# cat decrypt.txt
My bank secret is 1234
(venv) root@Debian11-bouselama:~#
```

Windows PowerShell

```
Key 20: ENCR ERMR ERLR
Key 21: DMBQ DQLQ DQKQ
Key 22: CLAP CPKP CPJP
Key 23: BKZO BOJO BOIO
Key 24: AJYN ANIN ANHN
Key 25: ZIXM ZMHM ZMGM
(venv) root@Debian11-bouselama:~# nano caesar_brute_force.py
(venv) root@Debian11-bouselama:~# echo "My bank secret is 1234" > secret.txt
(venv) root@Debian11-bouselama:~# openssl enc -aes-256-cbc -in secret.txt -out secret.enc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
Verify failure
bad password read
40376737897F0000:error:1400006B:UI routines:UI_process:processing error:../crypto/ui/ui_lib.c:552:while
 reading strings
(venv) root@Debian11-bouselama:~# openssl enc -aes-256-cbc -in secret.txt -out secret.enc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
Verify failure
bad password read
405744A8CC7F0000:error:1400006B:UI routines:UI_process:processing error:../crypto/ui/ui_lib.c:552:while
 reading strings
(venv) root@Debian11-bouselama:~# openssl enc -aes-256-cbc -in secret.txt -out secret.enc -pbkdf2
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
(venv) root@Debian11-bouselama:~# ls
caesar_brute_force.py           King-infra.lan.+008+21898.private  secret.txt
King-infra.lan.+008+21898.key  secret.enc                          security-tps
(venv) root@Debian11-bouselama:~# cat secret.enc
Salted__#*#'◆◆aL◆v◆◆ʕ◆:◆:'g◆◆PRõB◆◆%◆◆C◆◆◆u(venv) root@Debian11-bouselama:~#
```

TP 3 : Visualisation de la Faille ECB

Objectif

"Voir" pourquoi le mode **ECB** est dangereux

1. Extraire l'en-tête BMP (54 octets)

```
head -c 54 image.bmp > header.bin
```

2. Extraire le corps de l'image

```
tail -c +55 image.bmp > body.bin
```

3. Chiffrer le corps en AES-ECB

```
# Clé hexadécimale : "1234567890123456"
openssl enc -aes-128-ecb -in body.bin -out body_ecb.enc
-K $(echo -n '1234567890123456' | xxd -p) -nosalt
```

4. Reconstruire l'image chiffrée

```
cat header.bin body_ecb.enc > image_ecb.bmp
```

! Résultat Attendu

Contours de l'image **toujours visibles** malgré le chiffrement

Pourquoi BMP ?

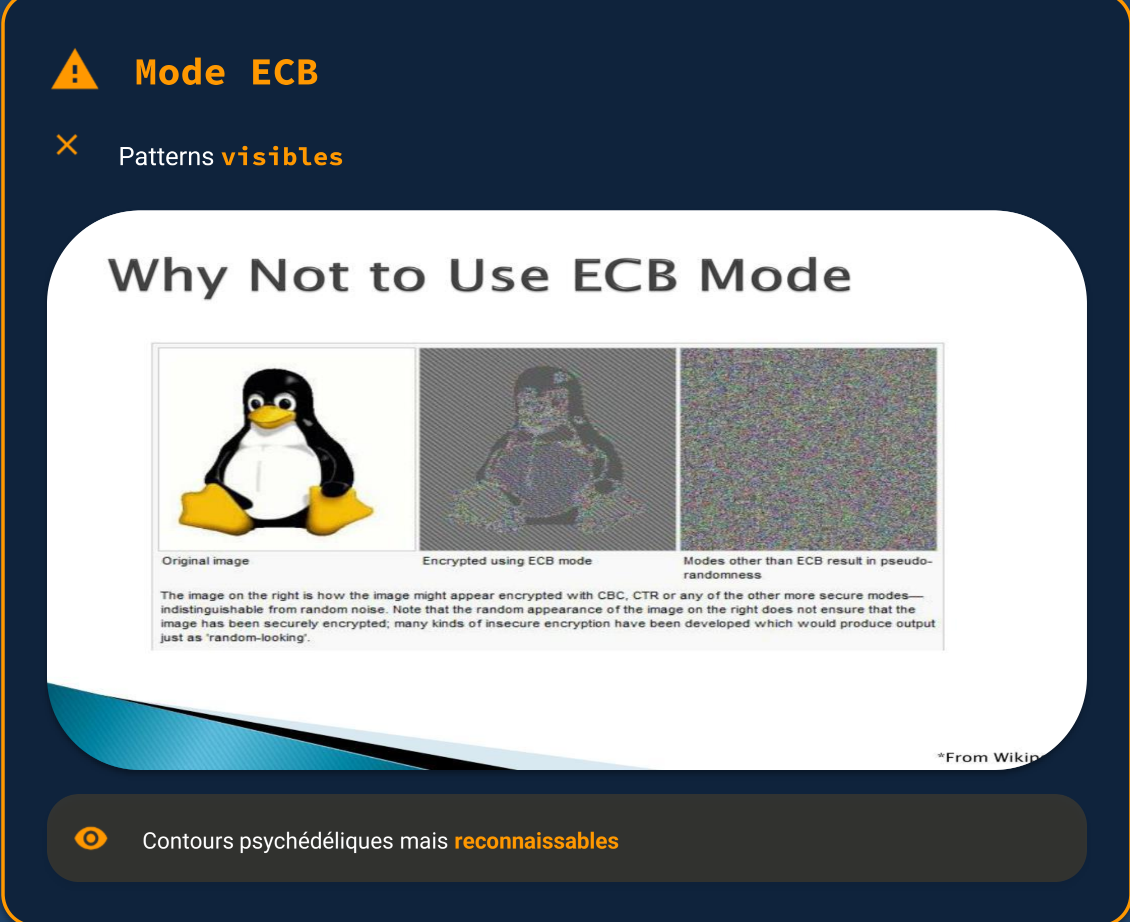
Format non compressé, les zones identiques ont les mêmes couleurs

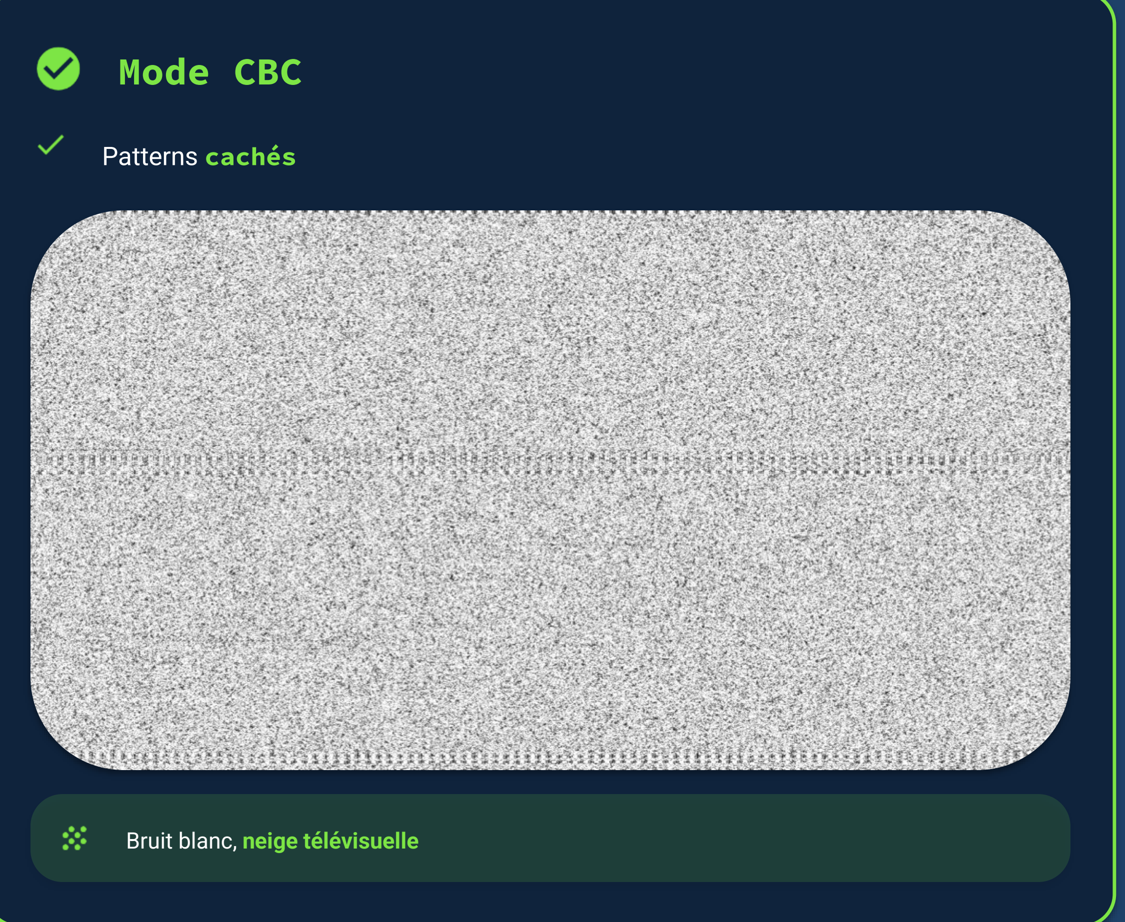
En-tête	Corps
54	**?**
octets	octets

Comparaison

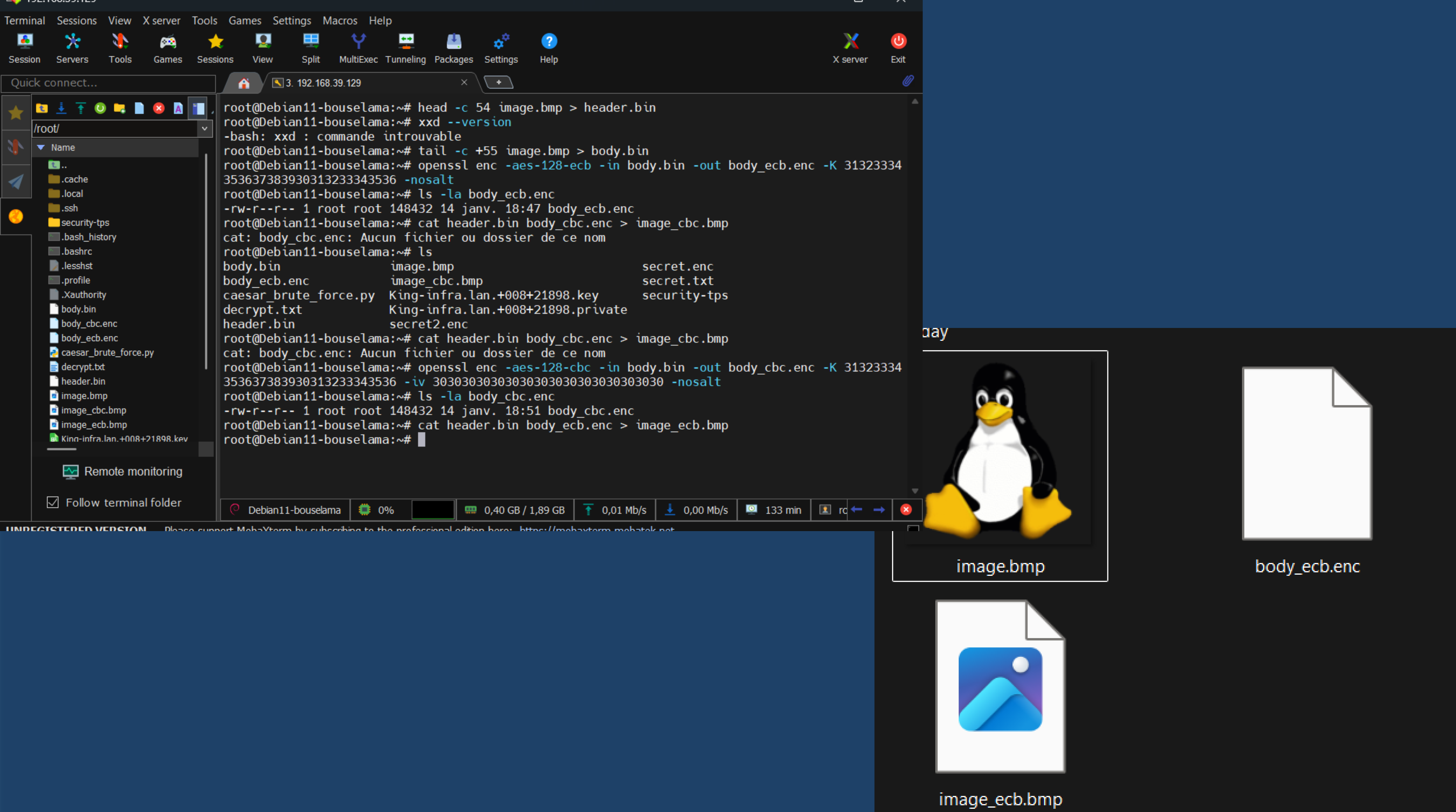
Refaites la manip avec CBC pour voir la différence !

Résultats et Comparaison



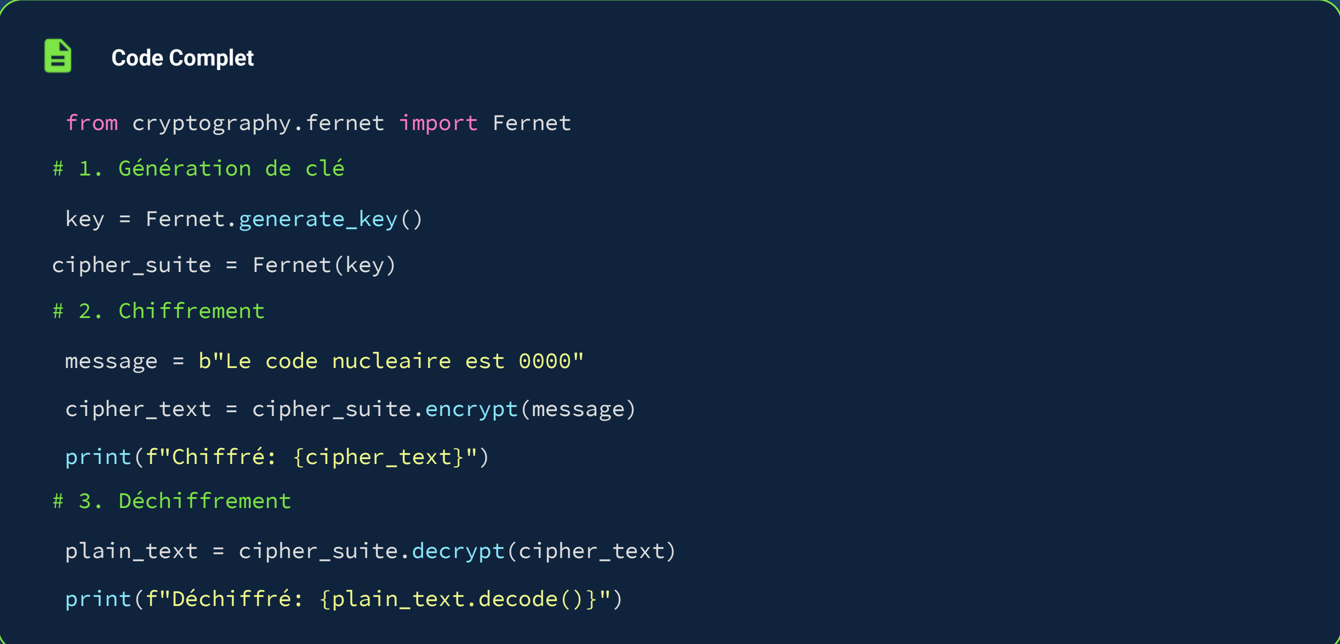


Le mode **CBC** utilise un vecteur d'initialisation (IV) aléatoire pour chaîner les blocs, détruisant ainsi les patterns visibles



192.168.39.129
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
Quick connect...
3. 192.168.39.129
/root/
Name
.cache
.local
.ssh
security-tps
.bash_history
.bashrc
.lesshst
.profile
.Xauthority
body.bin
body_cbc.enc
body_ecb.enc
caesar_brute_force.py
decrypt.txt
header.bin
image.bmp
image_cbc.bmp
image_ecb.bmp
root@Debian11-bouselama:~# head -c 54 image.bmp > header.bin
root@Debian11-bouselama:~# xxd --version
-bash: xxd : commande introuvable
root@Debian11-bouselama:~# tail -c +55 image.bmp > body.bin
root@Debian11-bouselama:~# openssl enc -aes-128-ecb -in body.bin -out body_ecb.enc -K 31323334 35363738393031323334353536 -nosalt
root@Debian11-bouselama:~# ls -la body_ecb.enc
-rw-r--r-- 1 root root 148432 14 janv. 18:47 body_ecb.enc
root@Debian11-bouselama:~# cat header.bin body_cbc.enc > image_cbc.bmp
cat: body_cbc.enc: Aucun fichier ou dossier de ce nom
root@Debian11-bouselama:~# ls
body.bin image.bmp secret.enc
body_ecb.enc image_cbc.bmp secret.txt
caesar_brute_force.py King-infra.lan.+008+21898.key security-tps
decrypt.txt King-infra.lan.+008+21898.private
header.bin secret2.enc
root@Debian11-bouselama:~# cat header.bin body_cbc.enc > image_cbc.bmp
cat: body_cbc.enc: Aucun fichier ou dossier de ce nom
root@Debian11-bouselama:~# openssl enc -aes-128-cbc -in body.bin -out body_cbc.enc -K 31323334 35363738393031323334353536 -iv 30303030303030303030303030303030 -nosalt
root@Debian11-bouselama:~# ls -la body_cbc.enc
-rw-r--r-- 1 root root 148432 14 janv. 18:51 body_cbc.enc
root@Debian11-bouselama:~# cat header.bin body_ecb.enc > image_ecb.bmp
root@Debian11-bouselama:~#
Remote monitoring
Follow terminal folder
Debian11-bouselama 0% 0,40 GB / 1,89 GB 0,01 Mb/s 0,00 Mb/s 133 min
image.bmp
body_ecb.enc
image_ecb.bmp

<> TP 4 : Implémentation Correcte en Python



Code Complet

```python
from cryptography.fernet import Fernet
# 1. Génération de clé
key = Fernet.generate_key()
cipher_suite = Fernet(key)
# 2. Chiffrement
message = b"Le code nucleaire est 0000"
cipher_text = cipher_suite.encrypt(message)
print(f"Chiffré: {cipher_text}")
# 3. Déchiffrement
plain_text = cipher_suite.decrypt(cipher_text)
print(f"Déchiffré: {plain_text.decode()}")
```

Génération

Automatic

Chiffrement

AES-128-CBC

Déchiffrement

HMAC inclus

Recommandation

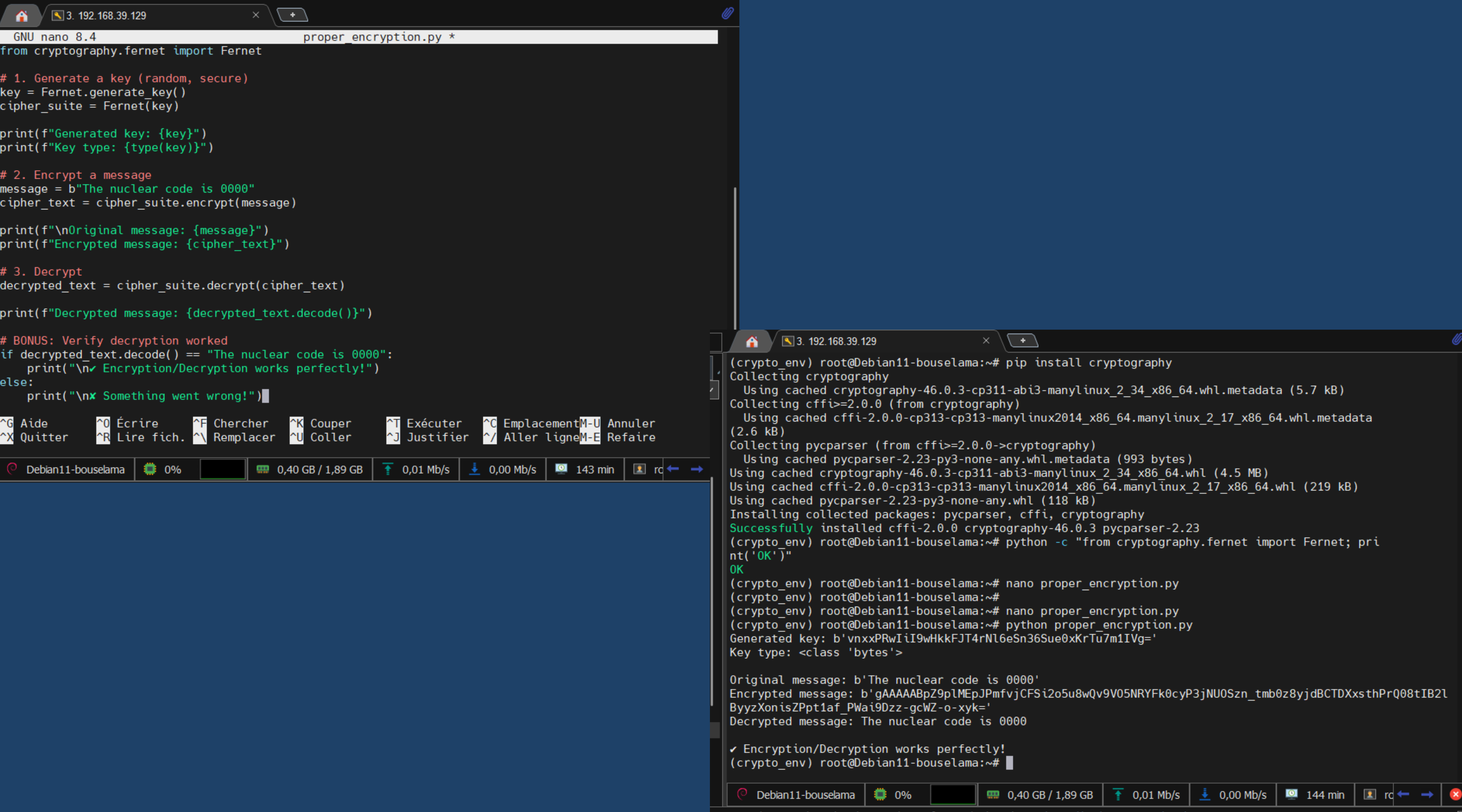
Utilisez une **bibliothèque éprouvée** plutôt que de coder l'algorithme vous-même

Bibliothèque cryptography

- ✓ Audité et maintenu par l'équipe de Python
- ✓ Gère le salage et HMAC automatiquement
- ✓ Protection contre les erreurs courantes

À éviter

NE JAMAIS utiliser votre propre implémentation AES en production



3. 192.168.39.129
GNU nano 8.4 proper_encryption.py *
from cryptography.fernet import Fernet
# 1. Generate a key (random, secure)
key = Fernet.generate_key()
cipher_suite = Fernet(key)
print(f"Generated key: {key}")
print(f"Key type: {type(key)}")
# 2. Encrypt a message
message = b"The nuclear code is 0000"
cipher_text = cipher_suite.encrypt(message)
print(f"\nOriginal message: {message}")
print(f"Encrypted message: {cipher_text}")
# 3. Decrypt
decrypted_text = cipher_suite.decrypt(cipher_text)
print(f"Decrypted message: {decrypted_text.decode()}")
# BONUS: Verify decryption worked
if decrypted_text.decode() == "The nuclear code is 0000":
print("\n✔ Encryption/Decryption works perfectly!")
else:
print("\n✘ Something went wrong!")
^G Aide ^O Écrire ^F Chercher ^K Couper ^T Exécuter ^C Emplacement M-U Annuler
^X Quitter ^R Lire fich. ^\ Remplacer ^U Coller ^J Justifier ^/ Aller ligne M-E Refaire
Debian11-bouselama 0% 0,40 GB / 1,89 GB 0,01 Mb/s 0,00 Mb/s 143 min
(crypto_env) root@Debian11-bouselama:~# pip install cryptography
Collecting cryptography
Using cached cryptography-46.0.3-cp311-abi3-manylinux_2_34_x86_64.whl.metadata (5.7 kB)
Collecting cffi>=2.0.0 (from cryptography)
Using cached cffi-2.0.0-cp313-cp313-manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (2.6 kB)
Collecting pycparser (from cffi>=2.0.0->cryptography)
Using cached pycparser-2.23-py3-none-any.whl.metadata (993 bytes)
Using cached cryptography-46.0.3-cp311-abi3-manylinux_2_34_x86_64.whl (4.5 MB)
Using cached cffi-2.0.0-cp313-cp313-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (219 kB)
Using cached pycparser-2.23-py3-none-any.whl (118 kB)
Installing collected packages: pycparser, cffi, cryptography
Successfully installed cffi-2.0.0 cryptography-46.0.3 pycparser-2.23
(crypto_env) root@Debian11-bouselama:~# python -c "from cryptography.fernet import Fernet; print('OK')"
OK
(crypto_env) root@Debian11-bouselama:~# nano proper_encryption.py
(crypto_env) root@Debian11-bouselama:~#
(crypto_env) root@Debian11-bouselama:~# nano proper_encryption.py
(crypto_env) root@Debian11-bouselama:~# python proper_encryption.py
Generated key: b'vnxxPRwIiI9wHkkFJT4rNl6eSn36Sue0xKrTu7m1IVg='
Key type: <class 'bytes'>
Original message: b'The nuclear code is 0000'
Encrypted message: b'gAAAAABpZ9plMEpJPmfvjCFSi2o5u8wQv9VO5NRYFk0cyP3jNUOSzn_tmb0z8yjdBCTDXxsthPrQ08tIB2lByyzXonisZPpt1af_PWai9Dzz-gcWZ-o-xyk='
Decrypted message: The nuclear code is 0000
✔ Encryption/Decryption works perfectly!
(crypto_env) root@Debian11-bouselama:~#
Debian11-bouselama 0% 0,40 GB / 1,89 GB 0,01 Mb/s 0,00 Mb/s 144 min
upport MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net

Devoir et Prochaines Étapes

Devoir

Pour la semaine prochaine

Chercher ce qu'est la **Distribution des Clés**

Question clé

Pourquoi cela a mené à l'invention de la **cryptographie asymétrique** (Diffie-Hellman)

Points Clés Appris

1. **Jamais votre propre algorithme**
   Utilisez des standards audités (AES, RSA)
2. **ECB est dangereux**
   Préférez CBC ou GCM pour masquer les patterns
3. **Bibliothèques éprouvées**
   Utilisez cryptography, OpenSSL, etc.
4. **Salage essentiel**
   Prévient les attaques rainbow table

Prochaine séance : **Cryptographie Asymétrique**