A02 - Cryptographic Failures

Intro

Cette catégorie regroupe les failles liées à une mauvaise utilisation ou absence de mécanismes cryptographiques, exposant ainsi des données sensibles. Anciennement nommée Sensitive Data Exposure, elle progresse à la deuxième place du classement car elle cible les erreurs fondamentales de chiffrement plutôt que leurs conséquences. Avec un taux d'incidence moyen de 4.49%, un total de 233 788 occurrences et 3 075 CVE, elle illustre la fréquence des erreurs de conception en cryptographie. Elle résulte souvent de mots de passe codés en dur ou de protocoles de chiffrement obsolètes qui compromettent la confidentialité des informations.

Scénario 1 : Encodages simples et Basic Auth

Attaque 1 : Basic Authentication (Base64)

Méthode / quand / pourquoi Base64 n'est pas du chiffrement. Il sert à encoder des octets lisibles (transport, headers). On l'utilise pour Basic Auth mais il faut TLS.

Code (clair → encodé) Bash :

```
# clair : "atelier:secret"
echo -n "atelier:secret" | base64
# sortie : YXRlbGllcjpzZWNyZXQ=
```

Chiffré (exemple) `Authorization: Basic YXRlbGllcjpzZWNyZXQ=`

Décodage / outils

- `echo 'YXRlbGllcjpzZWNyZXQ=' | base64 -d`
- Online: any Base64 decoder (ex: https://www.base64decode.org/) Facile, instantané.

Erreur commise Utiliser Base64 comme "protection" ou envoyer Basic Auth sans TLS.

Correction conceptuelle Ne jamais considérer Base64 comme secret. Toujours utiliser HTTPS. Préférer tokens signés (JWT) ou OAuth2. Ne pas logguer header Authorization.

Correction - code (exemples)

1. Forcer HTTPS (Nginx redirect) :

```
server {
  listen 80;
  server_name example.com;
  return 301 https://$host$request_uri;
}
```

2. Remplacer Basic par token (PHP pseudo) :

```
// reception d'un token Bearer
$auth = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
if (!preg_match('/Bearer\s+(\S+)/', $auth, $m)) { http_response_code(401); exit; }
$token = $m[1];
// valider token via DB / introspection / JWKS
```

Scénario 2 : Autres encodages et XOR obfuscation

Attaque 2 : XOR + Base64 (obfuscation reversible)

Méthode / quand / pourquoi XOR simple est utilisé comme obfuscation (config, keystore non protégé). C'est réversible si la clé est connue ou déduite.

Code (clair → chiffré) Python exemple (XOR single-byte puis base64) :

```
# clair -> xor(0x7F) -> base64
plain = b"DATABASEPASSWORD"
key = 0x7F
enc = bytes([b ^ key for b in plain])
import base64
print(base64.b64encode(enc).decode())
# sortie attendue : Oz4rPj0+LDovPiwsKDAtOw==
```

(ceci reproduit le format du challenge `{xor}Oz4rPj0+LDovPiwsKDAtOw==`)

Chiffré (exemple) `{xor}Oz4rPj0+LDovPiwsKDAtOw==`

Décodage / outils / méthode

- Base64 decode puis XOR avec la même clé.
- Script Python simple :

```
import base64
s = "Oz4rPj0+LDovPiwsKDAtOw=="
b = base64.b64decode(s)
key = 0x7F
print(bytes([x^key for x in b]).decode())
```

- Si la clé est inconnue on peut bruteforcer 256 valeurs.
- Outils : Python, small bruteforce script, binwalk not needed.

Erreur commise Obfuscation, pas de chiffrement. La clé est souvent embarquée ou trivialement déductible.

Correction conceptuelle Utiliser chiffrement authentifié (AES-GCM) avec gestion sécurisée de la clé. Ne pas stocker la clé dans le même endroit que le ciphertext. Pour mots de passe d'application, ne jamais stocker réversiblement.

Correction - code (chiffrement correct, Python cryptography)

```
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os, base64

key = os.urandom(32)           # stocker via KMS/secret manager
aesgcm = AESGCM(key)
nonce = os.urandom(12)
ct = aesgcm.encrypt(nonce, b"DATABASEPASSWORD", None)
print(base64.b64encode(nonce+ct).decode())  # stocker nonce+ct
```

Remarques déploiement : stocker `key` uniquement dans Vault/KMS, récupérer via API au runtime.

Scénario 3 : Hachage simple (MD5 / SHA-1 / SHA-256 non salés)

Attaque 3 : Hashs non salés ou algos faibles

Méthode / quand / pourquoi Hachage = intégrité. Mauvais pour mots de passe si sans sel et avec algo rapide. On peut inverser via tables rainbow ou brute force.

Exemples fournis

- `5F4DCC3B5AA765D61D8327DEB882CF99` → MD5 → `password`
- `8F0E2F76E22B43E2855189877E7DC1E1E7D98C226C95DB247CD1D547928334A9` → SHA-256 → `passw0rd`

Comment générer (clair→hash) Bash :

```
# MD5
echo -n "password" | md5sum
# SHA-256
echo -n "passw0rd" | sha256sum
```

Décodage / outils / méthodes

- Lookup in online DB (hashes.org, crackstation).
- `john` or `hashcat` with wordlist:

```
# john
john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-md5 hashfile
# hashcat (GPU)
hashcat -m 0 hash.txt /path/rockyou.txt
```

Erreur commise Usage d'algorithmes rapides sans sel. Stockage de hachage réversible via brute force.

Correction conceptuelle Utiliser des algorithmes de dérivation lente et mémoire-difficile : Argon2id, bcrypt, scrypt. Toujours saler (unique par entrée) et utiliser paramètres adaptés.

Correction - code (PHP & Python) PHP (recommandé) :

```
// stockage
$hash = password_hash($password, PASSWORD_ARGON2ID);
// vérification
if (password_verify($input, $hash)) { /* ok */ }
```

Python (argon2-cffi) :

```
from argon2 import PasswordHasher
ph = PasswordHasher()
h = ph.hash("mysecret")
ph.verify(h, "mysecret")
```

Remarques : définir coûts (time, memory, parallelism) en fonction de capacité serveur.

Scénario 4 : Chiffrement symétrique et TLS (utilisation inappropriée)

Attaque 4 : Utilisation incorrecte d'AES / mots de passe dans repos / chiffrement par passphrase

Méthode / quand / pourquoi AES est bon. Risque si :

- clé dérivée de mot de passe faible,
- utilisation de mode non authentifié (CBC sans HMAC),
- réutilisation d'IV.

Code (clair → chiffré) avec OpenSSL (exemple)

```
# chiffrement AES-256-CBC par passphrase (exemple pédagogique)
echo -n "SECRET_MESSAGE" > secret.txt
openssl enc -aes-256-cbc -salt -pbkdf2 -iter 100000 -in secret.txt -out secret.enc
-pass pass:MyS3cr3t
# base64:
base64 -w0 secret.enc
```

Décodage / outils `openssl enc -aes-256-cbc -d -pbkdf2 -iter 100000 -in secret.enc -out plain.txt -pass pass:MyS3cr3t` Si passphrase faible, on peut brute-forcer via `openssl` loop or `hashcat` against PBKDF2-derived keys.

Erreur commise

- PBKDF2 params faibles.
- Mode CBC sans authentification.
- Stockage de passphrase dans image/config.

Correction conceptuelle

- Utiliser chiffrement authentifié (AES-GCM).
- Utiliser KDFs robustes et store de clés séparé (KMS/HSM).
- Ne pas gérer clés dans le même repo.

Correction - code (Python AES-GCM)

```
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os, base64

key = os.urandom(32)  # from KMS in prod
aesgcm = AESGCM(key)
nonce = os.urandom(12)
ct = aesgcm.encrypt(nonce, b"SECRET_MESSAGE", None)
print(base64.b64encode(nonce+ct).decode())
```

Scénario 5 : Asymmetric / Signatures (RSA) — extraction de modulus et signature

Attaque 5 : Manipulation d'une clé privée fournie (WebGoat exercise)

Méthode / quand / pourquoi Avec la clé privée on peut : extraire le module, signer des messages, usurper identité si clé réelle.

Commandes utiles (extraire modulus, signer)

```
# extraire modulus hex depuis privkey.pem
openssl rsa -in priv.pem -noout -text | sed -n '/modulus:/,/publicExponent/p'

# exporter modulus en hex propre
openssl rsa -in priv.pem -noout -modulus | sed 's/Modulus=//'

# signer une chaîne (sha256) et base64
echo -n "<message>" | openssl dgst -sha256 -sign priv.pem | base64 -w0
```

Exemple chiffré / signature

- Message à signer : hex(modulus) (exercice WebGoat).
- Signature générée par la commande ci-dessus.

> Remarque : dans ce document nous fournissons la procédure. La génération effective de la signature requiert exécution locale d'OpenSSL. Je ne fournis pas la signature b64 ici car je ne peux pas exécuter OpenSSL.

Décodage / vérification

```
# vérifier signature
echo -n "<message>" > m.bin
echo "<base64sig>" | base64 -d > sig.bin
openssl dgst -sha256 -verify <(openssl rsa -in priv.pem -pubout) -signature
sig.bin m.bin
```

Erreur commise

- Clé privée partagée ou stockée en clair.
- Absence de gestion des droits sur fichiers PEM.
- Utilisation de clés dépassées (taille <2048 bits).

Correction conceptuelle

- Ne jamais distribuer la clé privée.
- Stocker dans HSM/KMS.
- Restreindre permissions (chmod 600) et accès.
- Utiliser PKI, révocation et rotation.

Correction - code / config

- Restreindre fichier :

```
chmod 600 /etc/ssl/private/priv.pem
chown root:ssl-cert /etc/ssl/private/priv.pem
```

- Utiliser signer via KMS (pseudo) :

```
# appel API KMS pour signer instead of local private key
# signer_request = kms.sign(key_id, digest)
```

Scénario 6 : Keystore / secrets in images (Docker) — récupération et déchiffrement

Attaque 6 : Secret laissé dans image + openssl decrypt (WebGoat challenge)

Méthode Extraire fichier de secret depuis l'image ou container et l'utiliser comme clé pour déchiffrer un ciphertext (exercice `findthesecret`).

Commandes (simulation)

```
docker run -d --name findthesecret webgoat/assignments:findthesecret
docker cp findthesecret:/root/default_secret ./default_secret
cat default_secret                          # montre la clé
echo "U2FsdGVkX1...D7as=" | openssl enc -aes-256-cbc -d -a -kfile default_secret
```

Chiffré donné (exemple WebGoat)
`U2FsdGVkX199jgh5oANElFdtCxIEvdEvciLi+v+5loE+VCuy6Ii0b+5byb5DXp32RPmT02Ek1pf55ctQN+DHbwCPiVRfFQamDmbHBUpD7as=`

Décodage / outils `openssl enc -aes-256-cbc -d -a -kfile default_secret` Facile si clé accessible.

Erreur commise

- Secrets baked into image.
- Clé et ciphertext co-localisés.

Correction conceptuelle

- Externaliser secrets (Docker secrets / Kubernetes secrets / Vault / cloud KMS).
- Build-time ≠ runtime secrets. Scanner images.

Correction - code (Docker Compose using secret) `docker-compose.yml` snippet:

```
services:
  app:
    image: myapp:latest
    secrets:
      - db_key
secrets:
  db_key:
    file: ./secrets/db_key.txt
```

CI pipeline example (scan with truffleHog / git-secrets) :

```
# fail build if secret found
trufflehog --json file://. > secrets.json || true
if jq '.results | length' secrets.json | grep -qv '^0$'; then
  echo "Secrets found, block build"
  exit 1
fi
```

Scénario 7 : Defaults, keystores & post-quantum note

Attaque 7 : Defaults et mots de passe par défaut

Méthode Trouver defaults (cacerts, keystore password default, id_rsa non chiffré) et les utiliser.

Exemples d'erreur

- `cacerts` non protégés.
- `id_rsa` privé sans passphrase dans un partage cloud.
- Keystore avec mot de passe public.

Correction

- Mettre des mots de passe non triviaux.
- Chiffrer clés privées par passphrase ou stocker en HSM.
- Forcer rotation, inventaire des keystores, audits.

Scénario 8 : Post-Quantum (note)

Méthode / risque Capture passive aujourd'hui + décryptage futur. Les données chiffrées avec algos vulnérables (RSA-2048, ECC) peuvent être lisibles plus tard.

Correction

- Commencer plan de migration PQC (hybride) pour données à conserver longtemps.
- Classer données sensibles par durée de confidentialité requise.

Conclusion

Checklist rapide pour tout projet Web vulnérable :

- Ne pas stocker secrets dans images ou repo.
- Utiliser chiffrement authentifié (AES-GCM), KMS/HSM pour clés.
- Pour mots de passe utilisateurs, utiliser Argon2id/Bcrypt/Scrypt.
- Remplacer BasicAuth sans TLS.
- Scanner images/commits pour secrets.
- Appliquer politique de rotation et monitoring d'accès aux clés.
- Pour signatures et clés privées, utiliser HSM/KMS et restreindre permissions.