CryptoSAS

DOCUMENTATION DÉVELOPPEUR

Guide d'intégration API

Chiffrement & Déchiffrement de fichiers

Comment chiffrer un fichier pour l'envoyer via l'API CryptoSAS

et comment déchiffrer un fichier reçu depuis l'API.

Exemples de code en Python.

Version API	v1
Algorithme	PBKDF2-SHA256 (600 000 it.) + AES-256-GCM
Bibliothèque Python	cryptography + requests
Mis à jour	07/08/2026
Base URL	https://cryptosas.newfutur.com/api/v1

1 Introduction et concepts

CryptoSAS est un coffre-fort numérique **zero-knowledge**. Les fichiers sont toujours stockés chiffrés sur le serveur. Deux modes d'intégration sont disponibles :

Critère	Mode client (recommandé)	Mode serveur (délégué)
Qui chiffre ?	Votre application	Le serveur CryptoSAS
Fichier vu par le serveur ?	✗ **Jamais**	⚠ **En mémoire seulement**
Clé persistée ?	✗ **Jamais**	✗ **Jamais**
Fichier stocké en clair ?	✗ **Jamais**	✗ **Jamais**
Zero-Knowledge ?	✓ **Complet**	⚠ **Partiel (transit TLS)**
Endpoint	`POST /api/v1/files`	`POST /api/v1/files/upload-plain`

Ce guide couvre les deux modes. Dans les deux cas, l'algorithme de chiffrement est identique : PBKDF2-SHA256 + AES-256-GCM. Les fichiers chiffrés par votre application peuvent être déchiffrés côté serveur et vice versa.

Algorithmes utilisés

Opération	Algorithme	Paramètres
Dérivation de clé	PBKDF2-SHA256	600 000 itérations, sel 16 bytes aléatoire, clé 32 bytes (256 bits)
Chiffrement	AES-256-GCM	IV 12 bytes aléatoire, tag d'authentification 16 bytes
Format blob stocké	ciphertext ‖ tag	Tag (16 bytes) concaténé à la fin — identique à Web Crypto API
Empreinte originale	SHA-256	Calculée sur le fichier original avant chiffrement

Authentification API

Toutes les requêtes nécessitent un header `Authorization` :

Méthode	Format	Obtention
JWT (Bearer)	`Authorization: Bearer <access_token>`	`POST /api/v1/auth/login`
Clé API	`Authorization: Bearer csk_<api_key>`	Profil → Clés API

2 Prérequis Python

Installation des bibliothèques

```bash
pip install cryptography requests
```

Imports communs à tous les exemples

```python — imports.py
import os
import hashlib
import urllib.parse
import requests
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

# ── Configuration ──────────────────────────────────────────────
API_URL  = "https://cryptosas.newfutur.com"
API_KEY  = "csk_votre_cle_api"   # Clé API générée depuis le profil CryptoSAS

HEADERS  = {"Authorization": f"Bearer {API_KEY}"}

KDF_ITERATIONS = 600_000   # PBKDF2-SHA256 — doit correspondre exactement à la valeur serveur
KEY_BYTES      = 32        # AES-256 = 256 bits = 32 bytes
```

Important : le nombre d'itérations PBKDF2 doit être exactement **600 000**. Toute autre valeur produit une clé différente rendant le déchiffrement impossible.

3 Mode client — Chiffrement local + upload

Mode recommandé — Zero-Knowledge complet. Votre application chiffre le fichier avant de l'envoyer. Le serveur ne voit jamais le contenu en clair.

Principe étape par étape

1. Lire le fichier original en mémoire
2. Calculer l'empreinte SHA-256 du fichier original (avant chiffrement)
3. Générer un sel aléatoire (16 bytes) et un IV aléatoire (12 bytes)
4. Dériver la clé AES-256 via PBKDF2-SHA256 (600 000 itérations)
5. Chiffrer avec AES-256-GCM → blob = ciphertext ‖ tag (16 bytes)
6. Envoyer le blob chiffré via `POST /api/v1/files` avec sel, IV, empreinte

Code Python

python — encrypt_upload.py

```
def encrypt_and_upload(
    file_path: str,
    password: str,
    folder_id: str = None
) -> dict:
    """
    Chiffre un fichier localement et l'uploade via l'API CryptoSAS.
    Le serveur ne voit jamais le contenu en clair (Zero-Knowledge complet).

    Args:
        file_path : chemin vers le fichier à chiffrer
        password  : mot de passe du coffre-fort (jamais envoyé au serveur)
        folder_id : UUID du dossier de destination (optionnel)

    Returns:
        dict : métadonnées du fichier créé (id, filename, kdf_salt, file_iv, ...)
    """
    # ── 1. Lire le fichier original ──────────────────────────────
    with open(file_path, "rb") as f:
        plaintext = f.read()

    # ── 2. Empreinte SHA-256 du fichier original (avant chiffrement) ─────
    original_hash = hashlib.sha256(plaintext).hexdigest()

    # ── 3. Générer sel PBKDF2 (16 bytes) et IV AES-GCM (12 bytes) ────────
    kdf_salt = os.urandom(16)   # 128 bits — aléatoire, différent par fichier
    file_iv  = os.urandom(12)   # 96 bits  — aléatoire, différent par fichier
```

```
    # ── 4. Dériver la clé AES-256 via PBKDF2-SHA256 ──────────────────────
    kdf = PBKDF2HMAC(
        algorithm  = hashes.SHA256(),
        length     = KEY_BYTES,       # 32 bytes = 256 bits
        salt       = kdf_salt,
        iterations = KDF_ITERATIONS,  # 600 000
    )
    key = kdf.derive(password.encode("utf-8"))

    # ── 5. Chiffrer avec AES-256-GCM ─────────────────────────────────────
    #   AESGCM.encrypt() retourne : ciphertext + tag(16 bytes)
    #   Format identique à Web Crypto API — directement compatible CryptoSAS
    aesgcm = AESGCM(key)
    blob   = aesgcm.encrypt(file_iv, plaintext, None)  # None = pas d'AAD

    # Effacer le plaintext et la clé de la mémoire
    del plaintext
    key = bytes(len(key))

    # ── 6. Uploader le blob chiffré via l'API ────────────────────────────
    filename = os.path.basename(file_path)

    form_data = {
        "filename"        : filename,
        "kdf_salt"        : kdf_salt.hex(),   # 32 chars hex
        "file_iv"         : file_iv.hex(),    # 24 chars hex
        "size_original"   : str(len(blob) - 16),  # taille avant chiffrement (sans tag)
        "encryption_mode" : "per_file",
        "original_hash"   : original_hash,   # 64 chars hex SHA-256
    }
    if folder_id:
        form_data["folder_id"] = folder_id

    files    = {"blob": ("blob.enc", blob, "application/octet-stream")}
    response = requests.post(
        f"{API_URL}/api/v1/files",
        data    = form_data,
        files   = files,
        headers = HEADERS,
    )
    response.raise_for_status()
    return response.json()


# ── Utilisation ──────────────────────────────────────────────────────────
if __name__ == "__main__":
    result = encrypt_and_upload(
        file_path = "contrat.pdf",
        password  = "mon_mot_de_passe_coffre",
    )
    file_id = result["file"]["id"]
    print(f"Fichier uploadé — ID : {file_id}")
```

```
print(f"Empreinte SHA-256 : {result['file']['original_hash']}")
```

size_original doit être la taille du fichier original (avant chiffrement). Le blob envoyé est plus grand de 16 bytes (tag GCM). Si vous connaissez la taille du plaintext, passez-la directement plutôt que de la calculer depuis le blob.

4 Mode client — Téléchargement + déchiffrement local

Zero-Knowledge complet. Le serveur retourne le blob chiffré et les paramètres cryptographiques dans les headers HTTP. Votre application déchiffre localement — le serveur ne voit jamais le contenu.

Headers de réponse retournés par l'API

Header	Valeur	Description
`X-KDF-Salt`	hex 32 chars	Sel PBKDF2 pour re-dériver la clé
`X-File-IV`	hex 24 chars	IV AES-GCM pour le déchiffrement
`X-Filename`	URL-encoded	Nom original du fichier
`X-Mimetype`	URL-encoded	Type MIME original
`X-Size-Original`	integer	Taille originale en octets
`X-Original-Hash`	hex 64 chars	SHA-256 du fichier original (si disponible)
`X-Encryption-Mode`	string	Mode : unified, per_file ou server

Principe étape par étape

1. Télécharger le blob chiffré via `GET /api/v1/files/{id}`
2. Lire `X-KDF-Salt` et `X-File-IV` depuis les headers HTTP
3. Dériver la clé AES-256 via PBKDF2-SHA256 avec le sel et le mot de passe
4. Déchiffrer avec AES-256-GCM (le tag est les 16 derniers bytes du blob)
5. Vérifier l'empreinte SHA-256 si `X-Original-Hash` est présent
6. Sauvegarder ou utiliser le fichier déchiffré

Code Python

python — download_decrypt.py

```
def download_and_decrypt(
    file_id     : str,
    password    : str,
    output_path : str = None,
) -> bytes:
    """
    Télécharge un blob chiffré depuis l'API CryptoSAS et le déchiffre localement.
    Zero-Knowledge complet : le serveur ne voit jamais le mot de passe ni la clé.

    Args:
        file_id     : UUID du fichier CryptoSAS
```

```
    password    : mot de passe du coffre-fort
    output_path : chemin de sauvegarde (optionnel)

Returns:
    bytes : contenu déchiffré du fichier original
Raises:
    ValueError : si le mot de passe est incorrect ou l'intégrité compromise
"""
# ── 1. Télécharger le blob chiffré ──────────────────────────────────────
response = requests.get(
    f"{API_URL}/api/v1/files/{file_id}",
    headers = HEADERS,
)
response.raise_for_status()

# ── 2. Lire les paramètres crypto dans les headers HTTP ─────────────────
kdf_salt_hex  = response.headers["X-KDF-Salt"]      # hex 32 chars
file_iv_hex   = response.headers["X-File-IV"]       # hex 24 chars
original_hash = response.headers.get("X-Original-Hash", "")
original_name = urllib.parse.unquote(
    response.headers.get("X-Filename", "fichier")
)

kdf_salt = bytes.fromhex(kdf_salt_hex)
file_iv  = bytes.fromhex(file_iv_hex)
blob     = response.content   # format : ciphertext || tag(16 bytes)

# ── 3. Dériver la même clé AES-256 via PBKDF2-SHA256 ────────────────────
kdf = PBKDF2HMAC(
    algorithm  = hashes.SHA256(),
    length     = KEY_BYTES,
    salt       = kdf_salt,
    iterations = KDF_ITERATIONS,
)
key = kdf.derive(password.encode("utf-8"))

# ── 4. Déchiffrer avec AES-256-GCM ──────────────────────────────────────
#   AESGCM.decrypt() attend : ciphertext || tag(16 bytes)
#   Lève une exception cryptography.exceptions.InvalidTag si le mot de passe
#   est incorrect ou si le blob est corrompu
aesgcm = AESGCM(key)
try:
    plaintext = aesgcm.decrypt(file_iv, blob, None)
except Exception:
    raise ValueError(
        "Déchiffrement échoué : mot de passe incorrect ou fichier corrompu."
    )

# Effacer la clé de la mémoire
key = bytes(len(key))

# ── 5. Vérifier l'empreinte SHA-256 ─────────────────────────────────────
```

```
    if original_hash:
        computed = hashlib.sha256(plaintext).hexdigest()
        if computed != original_hash:
            raise ValueError(
                f"Intégrité compromise !\n"
                f"  Attendu  : {original_hash}\n"
                f"  Calculé  : {computed}"
            )
        print(f"✓ Intégrité vérifiée — SHA-256 : {original_hash}")

    # ── 6. Sauvegarder si un chemin est fourni ──────────────────────────
    if output_path:
        with open(output_path, "wb") as f:
            f.write(plaintext)
        print(f"✓ Fichier '{original_name}' sauvegardé dans '{output_path}'")

    return plaintext


# ── Utilisation ──────────────────────────────────────────────────────────
if __name__ == "__main__":
    contenu = download_and_decrypt(
        file_id     = "550e8400-e29b-41d4-a716-446655440000",
        password    = "mon_mot_de_passe_coffre",
        output_path = "contrat_dechiffre.pdf",
    )
    print(f"Taille du fichier déchiffré : {len(contenu)} octets")
```

5 Mode serveur — Chiffrement et déchiffrement délégués

Zero-Knowledge partiel. Le serveur chiffre/déchiffre le fichier en mémoire. Le fichier transite en clair via TLS uniquement. Les fichiers sont toujours stockés chiffrés — le serveur ne peut pas lire leur contenu au repos. À utiliser lorsque votre application ne peut pas effectuer le chiffrement elle-même.

5.1 — Upload d'un fichier en clair (chiffrement délégué au serveur)

python — server_upload_plain.py

```
def upload_plain(
    file_path    : str,
    vault_password: str,
    folder_id    : str = None,
) -> dict:
    """
    Envoie un fichier en clair au serveur qui se charge de le chiffrer.
    Utile pour les clients légers qui ne peuvent pas effectuer le chiffrement.

    ⚠  Le contenu du fichier est visible par le serveur en mémoire pendant l'opération.
       Utilisez le mode client (section 3) pour un Zero-Knowledge complet.
    """
    with open(file_path, "rb") as f:
        file_content = f.read()

    form_data = {"vault_password": vault_password}
    if folder_id:
        form_data["folder_id"] = folder_id

    # Le champ 'filename' est optionnel — le serveur utilise le nom de l'upload
    files    = {"file": (os.path.basename(file_path), file_content, "application/octet-stream")}
    response = requests.post(
        f"{API_URL}/api/v1/files/upload-plain",
        data    = form_data,
        files   = files,
        headers = HEADERS,
    )
    response.raise_for_status()
    result = response.json()
    print(f"Fichier uploadé — ID : {result['file']['id']}")
    print(f"Mode : {result['file']['encryption_mode']}")   # affiche 'server'
    return result
```

5.2 — Téléchargement d'un fichier déchiffré par le serveur

python — server_decrypt.py

```python
def download_decrypted(
    file_id       : str,
    vault_password: str,
    output_path   : str = None,
) -> bytes:
    """
    Demande au serveur de déchiffrer le fichier et retourne le contenu original.
    Fonctionne pour tous les modes : unified, per_file et server.

    ⚠  Le mot de passe est transmis dans le corps de la requête (protégé par TLS).
       Il n'est jamais stocké par le serveur.
    """
    response = requests.post(
        f"{API_URL}/api/v1/files/{file_id}/decrypt",
        json    = {"vault_password": vault_password},  # JSON body
        headers = HEADERS,
    )

    # 401 = mot de passe incorrect
    if response.status_code == 401:
        raise ValueError("Mot de passe incorrect ou fichier corrompu.")
    response.raise_for_status()

    plaintext = response.content

    # Vérifier l'intégrité si le header X-Original-Hash est présent
    original_hash = response.headers.get("X-Original-Hash", "")
    if original_hash:
        computed = hashlib.sha256(plaintext).hexdigest()
        if computed != original_hash:
            raise ValueError("Intégrité compromise !")
        print(f"✓ Intégrité vérifiée — SHA-256 : {original_hash}")

    if output_path:
        with open(output_path, "wb") as f:
            f.write(plaintext)
        print(f"✓ Fichier déchiffré sauvegardé dans '{output_path}'")

    return plaintext
```

6 Script complet — Flux de bout en bout

Ce script illustre un cycle complet : chiffrement local, upload, re-téléchargement et déchiffrement.

python — full_example.py

```python
#!/usr/bin/env python3
"""
CryptoSAS — Exemple de flux complet (mode client Zero-Knowledge)
Dépendances : pip install cryptography requests
"""
import os, hashlib, urllib.parse, requests
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

API_URL        = "https://cryptosas.newfutur.com"
API_KEY        = "csk_votre_cle_api"
HEADERS        = {"Authorization": f"Bearer {API_KEY}"}
KDF_ITERATIONS = 600_000
KEY_BYTES      = 32

def derive_key(password: str, kdf_salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(hashes.SHA256(), KEY_BYTES, kdf_salt, KDF_ITERATIONS)
    return kdf.derive(password.encode("utf-8"))

def encrypt_file(plaintext: bytes, password: str) -> tuple[bytes, bytes, bytes, str]:
    """Retourne : blob, kdf_salt, file_iv, original_hash"""
    kdf_salt = os.urandom(16)
    file_iv  = os.urandom(12)
    key      = derive_key(password, kdf_salt)
    blob     = AESGCM(key).encrypt(file_iv, plaintext, None)
    orig_hash = hashlib.sha256(plaintext).hexdigest()
    return blob, kdf_salt, file_iv, orig_hash

def decrypt_blob(blob: bytes, password: str, kdf_salt: bytes, file_iv: bytes) -> bytes:
    key = derive_key(password, kdf_salt)
    return AESGCM(key).decrypt(file_iv, blob, None)

# ── 1. Chiffrer et uploader ──────────────────────────────────
print("=== UPLOAD ===")
with open("document.pdf", "rb") as f:
    plaintext = f.read()

blob, kdf_salt, file_iv, orig_hash = encrypt_file(plaintext, "mon_mot_de_passe")

resp = requests.post(f"{API_URL}/api/v1/files",
    data  = {"filename": "document.pdf", "kdf_salt": kdf_salt.hex(),
             "file_iv": file_iv.hex(), "size_original": str(len(plaintext)),
             "encryption_mode": "per_file", "original_hash": orig_hash},
    files = {"blob": ("blob.enc", blob, "application/octet-stream")},
```

```
    headers = HEADERS)
resp.raise_for_status()
file_id = resp.json()["file"]["id"]
print(f"✓ Uploadé — ID : {file_id}")

# ── 2. Télécharger et déchiffrer ──────────────────────────────
print("\n=== DOWNLOAD ===")
resp = requests.get(f"{API_URL}/api/v1/files/{file_id}", headers=HEADERS)
resp.raise_for_status()

remote_kdf_salt = bytes.fromhex(resp.headers["X-KDF-Salt"])
remote_file_iv  = bytes.fromhex(resp.headers["X-File-IV"])
remote_hash     = resp.headers.get("X-Original-Hash", "")

recovered = decrypt_blob(resp.content, "mon_mot_de_passe", remote_kdf_salt, remote_file_iv)

if remote_hash and hashlib.sha256(recovered).hexdigest() == remote_hash:
    print(f"✓ Intégrité vérifiée")

with open("document_recovered.pdf", "wb") as f:
    f.write(recovered)
print(f"✓ Fichier récupéré ({len(recovered)} octets)")
assert recovered == plaintext, "ERREUR : les fichiers sont différents !"
print("✓ Fichier identique à l'original — cycle complet réussi")
```

7 Référence rapide

Endpoints fichiers

Méthode	Endpoint	Description
GET	`/api/v1/files`	Lister les fichiers
POST	`/api/v1/files`	Upload blob chiffré côté client
GET	`/api/v1/files/{id}`	Télécharger blob chiffré
DELETE	`/api/v1/files/{id}`	Supprimer un fichier
PATCH	`/api/v1/files/{id}`	Renommer / déplacer
POST	`/api/v1/files/upload-plain`	⚠ Upload fichier en clair (chiffrement serveur)
POST	`/api/v1/files/{id}/decrypt`	⚠ Télécharger fichier déchiffré par le serveur

Codes d'erreur fréquents

Code	Cause	Solution
400	Paramètre manquant ou invalide	Vérifier kdf_salt (32 chars hex), file_iv (24 chars hex)
401	Token invalide / expiré / mot de passe incorrect	Vérifier le token ou le mot de passe
403	Email non vérifié	Vérifier l'email avant la première connexion
422	Fichier trop volumineux (mode serveur > 100 Mo)	Utiliser le mode client pour les gros fichiers
429	Rate limit atteint	Attendre avant de réessayer

Rappel des contraintes de format

Paramètre	Format attendu	Exemple
`kdf_salt`	hex lowercase, exactement 32 chars	`a1b2c3...d4e5f6` (16 bytes)
`file_iv`	hex lowercase, exactement 24 chars	`a1b2c3...d4e5f6` (12 bytes)
`original_hash`	hex lowercase, exactement 64 chars	SHA-256 du fichier original
`size_original`	integer (string dans le formulaire)	`"102400"`
`encryption_mode`	"unified" | "per_file"	`"per_file"` recommandé via API