AYI NEDJIMI Consultants

Disséquer l'Obscurité : Techniques Avancées de Déobfuscation

Catégorie : Retro-Ingenierie Lecture : 7 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Guide expert sur la déobfuscation statique de malwares polymorphes : exécution symbolique angr/Triton, Capstone, reconstruction CFG, YARA, Ghidra.

Disséquer l'Obscurité : Techniques Avancées de Déobfuscation constitue un enjeu majeur pour les professionnels de la sécurité informatique et les équipes techniques. Ce guide détaillé sur deobfuscation malwares polymorphes propose une méthodologie structurée, des outils éprouvés et des recommandations opérationnelles directement applicables. L'objectif est de fournir aux praticiens — consultants, ingénieurs sécurité, administrateurs systèmes — les connaissances et les techniques nécessaires pour aborder ce sujet avec rigueur. Chaque section s'appuie sur des retours d'expérience terrain et intègre les évolutions les plus récentes du domaine. Les recommandations présentées sont adaptées aux environnements d'entreprise et tiennent compte des contraintes opérationnelles réelles.

Table des matières

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

La **déobfuscation statique** se distingue de l'analyse dynamique par sa capacité à opérer sans exécuter le binaire suspect. Cette approche est indispensable dans plusieurs scénarios critiques : Pour approfondir, consultez notre article sur Fileless Malware Analyse Detection Memoire. Guide expert sur la déobfuscation statique de malwares polymorphes : exécution symbolique angr/Triton, Capstone, reconstruction CFG, YARA, Ghidra. La rétro-ingénierie est une discipline fondamentale en analyse de malware et en recherche de vulnérabilités. Disséquer l'Obscurité : Techniques Avancées de Déobfuscation couvre les techniques avancées utilisées par les analystes. Nous abordons notamment : analyse avec ghidra, questions frequentes et 10. conclusion : tendances émergentes et perspectives. 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.

- **Environnements air-gapped** où aucune sandbox n'est disponible
- **Malwares à déclenchement conditionnel** qui détectent les environnements virtualisés (anti-VM) et modifient leur comportement

- **Échantillons partiels** récupérés lors de forensics mémoire où seuls des fragments de code sont disponibles
- **Analyse à grande échelle** de milliers de variants nécessitant une automatisation par pipeline
- **Conformité légale** dans certaines juridictions interdisant l'exécution de code malveillant même en sandbox

Cet article détaille les techniques avancées permettant de déconstruire méthodiquement les couches d'obfuscation sans exécution, en s'appuyant sur des frameworks d'exécution symbolique (**angr**, **Triton**), de désassemblage programmatique (**Capstone**), et d'analyse automatisée (**YARA**, **Ghidra**, **IDA Pro**). Chaque section inclut du code fonctionnel directement applicable en contexte opérationnel. Pour approfondir, consultez notre article sur Reverse Engineering Dotnet Decompilation Analyse.

Avertissement juridique et éthique

Les techniques présentées dans cet article sont destinées exclusivement à la **défense et à la recherche en sécurité**. L'analyse de malwares doit être conduite dans un cadre légal approprié, idéalement sur des machines isolées dédiées. Toute utilisation offensive de ces connaissances est illégale au regard des articles 323-1 à 323-7 du Code pénal français et de la Convention de Budapest sur la cybercriminalité. Pour approfondir, consultez notre article sur Anti Retro Ingenierie Apt.

Notre avis d'expert

La rétro-ingénierie éthique est un pilier de la recherche en sécurité. Comprendre comment un exploit fonctionne au niveau assembleur permet de développer des détections plus robustes que celles basées sur de simples signatures. L'investissement dans les compétences reverse est stratégique.

L'unpacking statique d'UPX est trivial grâce à l'outil natif, mais de nombreux malwares modifient les en-têtes UPX pour empêcher le déballage automatique :

```
#!/usr/bin/env python3
"""
Unpacker statique pour UPX modifié.
Restaure les magic bytes UPX altérés par les malwares
avant de procéder au déballage.
"""
import struct
import subprocess
import sys
import shutil
from pathlib import Path

# Signatures UPX connues et leurs altérations courantes
UPX_SIGNATURES = {
    b'UPX0': [b'UPX0', b'\x00PX0', b'UXP0', b'XUP0'],
    b'UPX1': [b'UPX1', b'\x00PX1', b'UXP1', b'XUP1'],
    b'UPX2': [b'UPX2', b'\x00PX2', b'UXP2', b'XUP2'],
    b'UPX!': [b'UPX!', b'\x00PX!', b'UXP!', b'XUP!'],
}

def fix_upx_headers(filepath: str) -> str:
    """Corrige les en-têtes UPX altérés et retourne le chemin du fichier corrigé."""
    data = bytearray(Path(filepath).read_bytes())
    fixed = False

    # Restaurer les noms de sections UPX
    for original, variants in UPX_SIGNATURES.items():
        for variant in variants[1:]:  # Ignorer l'original
            offset = 0
            while True:
                pos = data.find(variant, offset)
                if pos == -1:
                    break
                print(f"  [+] Correction à offset 0x{pos:08X}: "
                      f"{variant} -> {original}")
                data[pos:pos+4] = original
                fixed = True
                offset = pos + 4

    # Vérifier et restaurer le magic UPX en fin de fichier
    # Le magic "UPX!" se trouve généralement dans l'overlay
    for i in range(len(data) - 4, max(len(data) - 1024, 0), -1):
        if data[i:i+3] in [b'\x00PX', b'XUP', b'UXP']:
            data[i:i+3] = b'UPX'
            fixed = True
            print(f"  [+] Magic UPX restauré à offset 0x{i:08X}")
            break

    if fixed:
        output_path = filepath + ".fixed"
        Path(output_path).write_bytes(data)
        return output_path
    return filepath

def unpack_upx(filepath: str, output_dir: str) -> bool:
    """Tente le déballage UPX avec restauration automatique des en-têtes."""
    print(f"[*] Tentative d'unpacking UPX: {filepath}")

    output_path = Path(output_dir) / (Path(filepath).stem + "_unpacked.exe")

    # Tentative directe
    result = subprocess.run(
```

```
        ["upx", "-d", filepath, "-o", str(output_path)],
        capture_output=True, text=True
    )

    if result.returncode == 0:
        print(f"  [+] Déballage réussi: {output_path}")
        return True

    print("  [-] Échec direct, tentative avec correction d'en-têtes...")

    # Correction et nouvelle tentative
    fixed_path = fix_upx_headers(filepath)
    if fixed_path != filepath:
        result = subprocess.run(
            ["upx", "-d", fixed_path, "-o", str(output_path)],
            capture_output=True, text=True
        )
        Path(fixed_path).unlink(missing_ok=True)

        if result.returncode == 0:
            print(f"  [+] Déballage réussi après correction: {output_path}")
            return True

    print("  [!] Échec - packer non-standard ou UPX lourdement modifié")
    return False

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print(f"Usage: {sys.argv[0]}  ")
        sys.exit(1)
    unpack_upx(sys.argv[1], sys.argv[2])
```

Triton est un framework d'analyse binaire dynamique développé par Quarkslab qui offre des capacités d'exécution symbolique plus granulaires qu'angr, avec un accent sur la **taint analysis** (analyse de propagation de données) et la **simplification d'expressions**.

Triton excelle dans la simplification de **Mixed Boolean-Arithmetic (MBA)** expressions, technique d'obfuscation très utilisée dans les malwares modernes qui combine opérations arithmétiques et logiques pour masquer des calculs simples :

```
#!/usr/bin/env python3
"""
Simplification d'expressions MBA obfusquées avec Triton.
Les expressions MBA (Mixed Boolean-Arithmetic) sont utilisées
par les obfuscateurs pour masquer des opérations triviales.

Exemple: (x ^ y) + 2*(x & y) est équivalent à x + y
"""
from triton import (
    TritonContext, ARCH, MODE, Instruction,
    SYMBOLIC_SIMPLIFICATION, AST_REPRESENTATION
)

def create_triton_ctx():
    """Initialise un contexte Triton pour x86-64."""
    ctx = TritonContext(ARCH.X86_64)
    ctx.setMode(MODE.ALIGNED_MEMORY, True)
    ctx.setMode(MODE.CONSTANT_FOLDING, True)
    ctx.setAstRepresentationMode(AST_REPRESENTATION.PYTHON)
    return ctx

def simplify_mba_expression(opcodes: bytes, base_addr: int = 0x400000) -> str:
    """
    Exécute symboliquement une séquence d'opcodes et simplifie
    l'expression résultante via le solveur Z3 intégré.

    Args:
        opcodes: bytes des instructions x86-64
        base_addr: adresse de base virtuelle

    Returns:
        Expression simplifiée sous forme de chaîne
    """
    ctx = create_triton_ctx()

    # Mapper les opcodes en mémoire
    for i, byte in enumerate(opcodes):
        ctx.setConcreteMemoryValue(base_addr + i, byte)

    # Rendre les registres d'entrée symboliques
    ctx.symbolizeRegister(ctx.registers.rax, "x")
    ctx.symbolizeRegister(ctx.registers.rbx, "y")

    # Exécuter instruction par instruction
    pc = base_addr
    executed = []
    while pc < base_addr + len(opcodes):
        inst = Instruction(pc, ctx.getConcreteMemoryAreaValue(pc, 16))
        if not ctx.processing(inst):
            break
        executed.append(f"0x{pc:x}: {inst.getDisassembly()}")
        pc = inst.getNextAddress()

    print("[*] Instructions exécutées:")
    for line in executed:
        print(f"    {line}")

    # Extraire l'expression symbolique du résultat (dans rax)
    rax_expr = ctx.getSymbolicRegister(ctx.registers.rax)
    if rax_expr is None:
        return "N/A (registre non modifié)"
```

```
    ast = rax_expr.getAst()
    simplified = ctx.simplify(ast, SYMBOLIC_SIMPLIFICATION.LLVM)

    print(f"\n[*] Expression brute:     {ast}")
    print(f"[+] Expression simplifiée: {simplified}")

    return str(simplified)

def deobfuscate_constant_unfolding(ctx, instructions_bytes, base=0x400000):
    """
    Résout le constant unfolding : technique où une constante simple
    est calculée par une longue série d'opérations.

    Exemple: mov rax, 0x1234 obfusqué en:
        mov rax, 0xDEADBEEF
        xor rax, 0xDEADBEEF ^ 0x1234  (= 0xDEADACDB)
        add rax, 0x5678
        sub rax, 0x5678
    """
    for i, b in enumerate(instructions_bytes):
        ctx.setConcreteMemoryValue(base + i, b)

    pc = base
    while pc < base + len(instructions_bytes):
        inst = Instruction(pc, ctx.getConcreteMemoryAreaValue(pc, 16))
        if not ctx.processing(inst):
            break
        pc = inst.getNextAddress()

    # Évaluer la valeur concrète résultante
    rax_val = ctx.getConcreteRegisterValue(ctx.registers.rax)
    return rax_val

# Exemple MBA courant dans les malwares obfusqués
# L'expression (x ^ y) + 2*(x & y) est équivalente à x + y
if __name__ == "__main__":
    # Opcodes x86-64 pour :
    #   mov rax, rdi        ; rax = x
    #   mov rbx, rsi        ; rbx = y
    #   mov rcx, rax
    #   xor rcx, rbx        ; rcx = x ^ y
    #   and rax, rbx        ; rax = x & y
    #   shl rax, 1          ; rax = 2 * (x & y)
    #   add rax, rcx        ; rax = (x ^ y) + 2*(x & y) = x + y
    mba_opcodes = (
        b'\x48\x89\xf8'       # mov rax, rdi
        b'\x48\x89\xf3'       # mov rbx, rsi
        b'\x48\x89\xc1'       # mov rcx, rax
        b'\x48\x31\xd9'       # xor rcx, rbx
        b'\x48\x21\xd8'       # and rax, rbx
        b'\x48\xd1\xe0'       # shl rax, 1
        b'\x48\x01\xc8'       # add rax, rcx
    )

    print("=" * 60)
    print("Simplification MBA avec Triton")
    print("=" * 60)
    simplify_mba_expression(mba_opcodes)
```

Lorsque de4dot ne parvient pas à résoudre automatiquement les chaînes protégées (versions modifiées de ConfuserEx), il est nécessaire de reproduire l'algorithme de déchiffrement manuellement. Voici l'implémentation du déchiffreur de chaînes ConfuserEx :

```
#!/usr/bin/env python3
"""
Déchiffreur de chaînes ConfuserEx pour RedLine Stealer.
Reproduit l'algorithme de déchiffrement des chaînes protégées
sans exécuter le binaire .NET.

ConfuserEx utilise typiquement:
1. Un tableau de bytes compressé (deflate) stocké en ressource
2. Un déchiffrement XOR avec clé dérivée du token de la méthode appelante
3. Optionnellement un chiffrement additionnel (RC4 ou mutation custom)
"""
import struct
import zlib
import hashlib
from typing import Optional

class ConfuserExStringDecryptor:
    """Déchiffre les chaînes protégées par ConfuserEx."""

    def __init__(self, encrypted_resource: bytes, module_key: int):
        """
        Args:
            encrypted_resource: contenu brut de la ressource .NET
                                contenant les chaînes chiffrées
            module_key: clé de module (typiquement le RID
                        de la méthode de déchiffrement)
        """
        self.module_key = module_key
        self.data = self._decompress(encrypted_resource)
        self.strings_cache = {}

    def _decompress(self, data: bytes) -> bytes:
        """Décompresse les données (deflate sans header zlib)."""
        try:
            return zlib.decompress(data, -15)  # Raw deflate
        except zlib.error:
            try:
                return zlib.decompress(data)   # Avec header zlib
            except zlib.error:
                return data  # Pas compressé

    def _derive_key(self, caller_token: int) -> int:
        """
        Dérive la clé XOR à partir du token de la méthode appelante.
        Reproduit l'algorithme de ConfuserEx:
          key = (caller_token * 0x5bd1e995) ^ module_key
        """
        key = (caller_token * 0x5BD1E995) & 0xFFFFFFFF
        key = key ^ self.module_key
        return key

    def decrypt_string(self, string_id: int,
                       caller_token: int) -> Optional[str]:
        """
        Déchiffre une chaîne par son identifiant et le token de
        la méthode appelante.

        Args:
            string_id: index dans le tableau de chaînes
            caller_token: metadata token de la méthode .NET appelante
                          (format 0x0600XXXX pour MethodDef)
```

```
        Returns:
            La chaîne déchiffrée ou None en cas d'erreur
        """
        cache_key = (string_id, caller_token)
        if cache_key in self.strings_cache:
            return self.strings_cache[cache_key]

        key = self._derive_key(caller_token)

        try:
            # Lire l'offset et la longueur depuis le header
            offset = string_id * 4
            if offset + 4 > len(self.data):
                return None

            str_offset = struct.unpack_from('= len(self.data):
                return None

            # Lire la longueur de la chaîne (encodée en 7-bit compact)
            pos = str_offset
            str_len = 0
            shift = 0
            while pos < len(self.data):
                b = self.data[pos]
                str_len |= (b & 0x7F) << shift
                pos += 1
                if (b & 0x80) == 0:
                    break
                shift += 7

            if pos + str_len * 2 > len(self.data):
                return None

            # Déchiffrer les caractères UTF-16LE
            chars = []
            xor_key = key
            for i in range(str_len):
                c = struct.unpack_from('> (8 * (i % 4))) & 0xFFFF
                chars.append(chr(c))
                # Rotation de la clé
                xor_key = ((xor_key >> 3) | (xor_key << 29)) & 0xFFFFFFFF

            result = ''.join(chars)
            self.strings_cache[cache_key] = result
            return result

        except (struct.error, IndexError, ValueError):
            return None

    def decrypt_all(self, method_tokens: list) -> dict:
        """
        Tente de déchiffrer toutes les chaînes pour une liste
        de tokens de méthodes.

        Returns:
            Dict mapping (string_id, token) -> chaîne déchiffrée
        """
        results = {}
        for token in method_tokens:
            for sid in range(1000):  # Heuristique: max 1000 chaînes
                s = self.decrypt_string(sid, token)
                if s and len(s) > 0 and all(
```

```
                    c.isprintable() or c in '\r\n\t' for c in s
                ):
                    results[(sid, hex(token))] = s
        return results

if __name__ == "__main__":
    # Exemple d'utilisation avec un échantillon RedLine
    # Les valeurs ci-dessous sont à adapter au sample analysé
    print("[*] ConfuserEx String Decryptor")
    print("    Adapter encrypted_resource et module_key au sample")

    # Simulation avec des données de test
    test_data = bytes(range(256)) * 4
    decryptor = ConfuserExStringDecryptor(
        encrypted_resource=test_data,
        module_key=0x1A2B3C4D
    )
    print(f"    Data size after decompression: {len(decryptor.data)}")
    print("    Prêt pour decrypt_string(id, caller_token)")
```

```
/*
 * Règles YARA avancées pour la détection de packing
 * et d'obfuscation dans les binaires PE.
 *
 * Auteur: Ayi NEDJIMI - ayinedjimi-consultants.fr
 * Date: 2026-02-05
 */

import "pe"
import "math"
import "hash"

rule Packed_High_Entropy_Sections {
    meta:
        description = "Détecte les PE avec sections à haute entropie (packing probable)"
        severity = "medium"
        category = "packer"

    condition:
        uint16(0) == 0x5A4D and
        for any section in pe.sections : (
            math.entropy(section.offset, section.size) > 7.2 and
            section.size > 1024
        )
}

rule Packed_UPX_Modified {
    meta:
        description = "Détecte UPX avec en-têtes modifiés (anti-unpacking)"
        severity = "high"
        category = "packer"

    strings:
        // Noms de sections UPX altérés (premier octet modifié)
        $s1 = { 00 50 58 30 }  // \x00PX0 au lieu de UPX0
        $s2 = { 00 50 58 31 }  // \x00PX1
        $s3 = { 55 58 50 30 }  // UXP0 (octets inversés)
        $s4 = { 55 58 50 31 }  // UXP1

        // Pattern du stub UPX même modifié
        $stub = { 60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 }

    condition:
        uint16(0) == 0x5A4D and
        (any of ($s*)) and
        $stub
}

rule VMProtect_Virtualized {
    meta:
        description = "Détecte la virtualisation VMProtect"
        severity = "critical"
        category = "protector"

    strings:
        // Nom de section VMProtect
        $vmp0 = ".vmp0"
        $vmp1 = ".vmp1"
        $vmp2 = ".vmp2"

        // Pattern d'entrée VM (push regs + jmp dispatcher)
        $vm_entry_x86 = {
```

```
            60                  // pushad
            9C                  // pushfd
            68 ?? ?? ?? ??      // push imm32
            E8 ?? ?? ?? ??      // call vm_dispatcher
        }

        $vm_entry_x64 = {
            50 51 52 53         // push rax,rcx,rdx,rbx
            55 56 57            // push rbp,rsi,rdi
            41 50 41 51         // push r8, r9
        }

    condition:
        uint16(0) == 0x5A4D and
        (any of ($vmp*)) and
        (any of ($vm_entry*))
}

rule Themida_Protected {
    meta:
        description = "Détecte la protection Themida/WinLicense"
        severity = "critical"
        category = "protector"

    strings:
        $s1 = ".themida" ascii
        $s2 = ".winlice" ascii
        $s3 = "THEMIDA" ascii wide
        // Anti-debug check pattern
        $anti_dbg = {
            64 A1 30 00 00 00   // mov eax, fs:[0x30] (PEB)
            0F B6 40 02         // movzx eax, byte [eax+2] (BeingDebugged)
            85 C0               // test eax, eax
        }

    condition:
        uint16(0) == 0x5A4D and
        (any of ($s*)) and
        $anti_dbg
}

rule ConfuserEx_NET_Obfuscated {
    meta:
        description = "Détecte l'obfuscation ConfuserEx sur binaires .NET"
        severity = "high"
        category = "obfuscator"
        family = "confuserex"

    strings:
        // Markers ConfuserEx dans les métadonnées .NET
        $marker1 = "ConfuserEx" ascii wide nocase
        $marker2 = "Confuser.Core" ascii
        // Pattern de protection de chaînes (delegate call)
        $str_prot = {
            7E ?? ?? ?? 04      // ldsfld
            28 ?? ?? ?? 06      // call
            72 ?? ?? ?? 70      // ldstr ""
        }
        // Anti-tamper module initializer
        $tamper = {
            28 ?? ?? ?? 0A      // call
            2A                  // ret
```

```
            00                    // padding
            13 30                 // .method header
        }

    condition:
        uint16(0) == 0x5A4D and
        (any of ($marker*) or $str_prot) and
        pe.imports("mscoree.dll", "_CorExeMain")
}

rule Polymorphic_XOR_Decrypt_Stub {
    meta:
        description = "Détecte les stubs de déchiffrement XOR polymorphes"
        severity = "high"
        category = "polymorphic"

    strings:
        // Pattern: boucle XOR avec registre index + compteur
        $xor_loop_1 = {
            30 (1? | 0?) [0-2]    // xor byte [reg+idx], reg
            (40 | 41 | 42 | 43 | 46 | 47 | FF C? | 83 C? 01)  // inc reg
            (48 | 49 | 4A | 4B | 4E | 4F | FF C? | 83 E? 01)  // dec reg
            (75 | 0F 85) ??       // jnz loop
        }

        // Pattern: boucle XOR avec LOOP instruction
        $xor_loop_2 = {
            30 [1-3]              // xor byte [mem], reg
            [0-4]                 // possible inc/lea
            E2 ??                 // loop
        }

        // Variante: XOR word/dword
        $xor_loop_3 = {
            (31 | 33) [1-3]       // xor dword [mem], reg
            (83 C? 04 | 83 E? 04) // add/sub reg, 4
            (3B | 39 | 3D) [1-5]  // cmp
            (72 | 76 | 7C | 0F 82 | 0F 86) ??  // jb/jbe/jl
        }

    condition:
        uint16(0) == 0x5A4D and
        any of ($xor_loop_*) and
        for any section in pe.sections : (
            math.entropy(section.offset, section.size) > 6.8
        )
}

rule RedLine_Stealer_Deobfuscated {
    meta:
        description = "Détecte RedLine Stealer après déobfuscation"
        severity = "critical"
        category = "infostealer"
        family = "redline"
        mitre = "T1555, T1539, T1552"

    strings:
        $func1 = "ScanBrowsers" ascii wide
        $func2 = "ScanFTP" ascii wide
        $func3 = "ScanWallets" ascii wide
        $func4 = "GrabInfo" ascii wide
        $func5 = "ScanScreen" ascii wide
```

```
        $func6 = "ScanTelegram" ascii wide
        $func7 = "ScanDiscord" ascii wide
        $func8 = "ScanSteam" ascii wide

        $net1 = "Authorization" ascii wide
        $net2 = "Content-Type" ascii wide
        $net3 = "application/soap+xml" ascii wide

        $cfg1 = /\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}:\d{2,5}/ ascii

    condition:
        uint16(0) == 0x5A4D and
        pe.imports("mscoree.dll", "_CorExeMain") and
        (3 of ($func*)) and
        (2 of ($net*)) and
        $cfg1
}
```

8.2 Pipeline d'automatisation Python + YARA

L'intégration de YARA dans un pipeline de triage automatisé permet de classer rapidement les échantillons selon leur niveau d'obfuscation et d'orienter l'analyse vers les outils appropriés :

9.1 Ghidra scripting pour la déobfuscation

Ghidra, le framework de rétro-ingénierie open-source développé par la NSA, offre un environnement de scripting puissant via Java et Python (Jython). Son décompilateur intégré et son moteur d'analyse P-Code en font un outil particulièrement adapté à la déobfuscation automatisée.

Le script suivant utilise l'API Ghidra pour identifier et résoudre automatiquement les prédicats opaques dans une fonction analysée :

IDA Pro reste l'outil de référence pour l'analyse de binaires complexes. Son API IDAPython permet d'automatiser des tâches de déobfuscation avancées. Voici un script pour la détection et la reconstruction des appels indirects obfusqués :

```
"""
IDAPython: Résolution d'appels indirects obfusqués.
Identifie les patterns call [reg] où le registre est
calculé par une séquence d'opérations obfusquée, et
résout la cible réelle par émulation partielle.
"""
import ida_bytes
import ida_funcs
import ida_ua
import ida_idp
import idautils
import idc

def resolve_indirect_calls(func_ea):
    """
    Parcourt une fonction et résout les appels indirects
    dont la cible est calculée par constant folding.
    """
    func = ida_funcs.get_func(func_ea)
    if not func:
        print(f"[-] Pas de fonction à 0x{func_ea:x}")
        return

    resolved = 0

    for head in idautils.Heads(func.start_ea, func.end_ea):
        insn = ida_ua.insn_t()
        if ida_ua.decode_insn(insn, head) == 0:
            continue

        # Chercher les CALL indirects (call reg ou call [mem])
        if insn.itype not in [ida_idp.NN_call, ida_idp.NN_callfi,
                              ida_idp.NN_callni]:
            continue

        op = insn.ops[0]
        if op.type == ida_ua.o_reg:
            # call reg -> remonter pour trouver la valeur
            reg_name = ida_idp.get_reg_name(op.reg, 8)
            target = _trace_register_value(head, op.reg, func)

            if target and target != 0:
                print(f"  [+] 0x{head:x}: call {reg_name} "
                      f"-> 0x{target:x}")
                # Ajouter un commentaire et une xref
                idc.set_cmt(head, f"Résolu: call 0x{target:x}", False)
                ida_bytes.add_cref(head, target, 0)  # Code xref
                resolved += 1

    print(f"[*] {resolved} appels indirects résolus")
    return resolved

def _trace_register_value(call_addr, reg, func):
    """
    Remonte les instructions depuis call_addr pour
    déterminer la valeur du registre par propagation
    de constantes.
    """
    # Parcourir les instructions précédentes (max 20)
    addr = call_addr
    value = None
    operations = []
```

```
    for _ in range(20):
        addr = idc.prev_head(addr, func.start_ea)
        if addr == idc.BADADDR or addr < func.start_ea:
            break

        insn = ida_ua.insn_t()
        if ida_ua.decode_insn(insn, addr) == 0:
            continue

        # MOV reg, imm -> valeur directe
        if (insn.itype == ida_idp.NN_mov and
            insn.ops[0].type == ida_ua.o_reg and
            insn.ops[0].reg == reg and
            insn.ops[1].type == ida_ua.o_imm):
            value = insn.ops[1].value
            break

        # ADD reg, imm
        if (insn.itype == ida_idp.NN_add and
            insn.ops[0].type == ida_ua.o_reg and
            insn.ops[0].reg == reg and
            insn.ops[1].type == ida_ua.o_imm):
            operations.append(('add', insn.ops[1].value))

        # SUB reg, imm
        if (insn.itype == ida_idp.NN_sub and
            insn.ops[0].type == ida_ua.o_reg and
            insn.ops[0].reg == reg and
            insn.ops[1].type == ida_ua.o_imm):
            operations.append(('sub', insn.ops[1].value))

        # XOR reg, imm
        if (insn.itype == ida_idp.NN_xor and
            insn.ops[0].type == ida_ua.o_reg and
            insn.ops[0].reg == reg and
            insn.ops[1].type == ida_ua.o_imm):
            operations.append(('xor', insn.ops[1].value))

    # Appliquer les opérations en ordre inverse
    if value is not None:
        for op, imm in reversed(operations):
            if op == 'add':
                value = (value + imm) & 0xFFFFFFFFFFFFFFFF
            elif op == 'sub':
                value = (value - imm) & 0xFFFFFFFFFFFFFFFF
            elif op == 'xor':
                value = value ^ imm
        return value

    return None
```

Analyse avec Ghidra

9.3 Comparaison des outils de reverse engineering

Critère	Ghidra	IDA Pro	Binary Ninja	radare2/rizin
Licence	Open-source (Apache 2.0)	Commercial (~1700 EUR/an)	Commercial (~350 USD)	Open-source (LGPL3)
Décompilateur	Intégré (P-Code)	Hex-Rays (addon payant)	Intégré (HLIL/MLIL)	Via r2ghidra ou r2dec
Scripting	Java, Python (Jython)	Python (IDAPython)	Python, C++, Rust	r2pipe (Python, JS, etc.)
Architectures	~30+ (x86, ARM, MIPS, PPC...)	~20+ avec plugins	~15+ officielles	~30+ (très extensible)
Analyse headless	analyzeHeadless (excellent)	idat (limité)	binaryninja.MainThreadAction	Natif (CLI-first)
Déobfuscation	P-Code simplifié, bon CFG	microcode, le plus mature	BNIL, SSA, bon pour CFG	ESIL, basique mais extensible
Intégration pipeline	Excellente (Ghidrathon)	Bonne (IDAPython batch)	Très bonne (API Python)	Excellente (r2pipe)
Forces pour déobfuscation	P-Code normalise le code, décompilateur gratuit robuste	Microcode et Hex-Rays le plus puissant, écosystème de plugins	IL multi-niveaux (LLIL, MLIL, HLIL), API très propre	Léger, rapide, excellent pour l'automatisation CLI

9.4 radare2/rizin : automatisation CLI

radare2 (et son fork **rizin**) excelle dans l'automatisation en ligne de commande et l'intégration dans des scripts shell. Voici un exemple d'analyse automatisée via `r2pipe` :

```
#!/usr/bin/env python3
"""
Analyse automatisée de malware obfusqué via r2pipe (radare2).
Extrait les chaînes, identifie les fonctions suspectes et
détecte les patterns d'obfuscation courants.
"""
import r2pipe
import json
from typing import Dict, List

def analyze_with_radare2(filepath: str) -> Dict:
    """Analyse complète d'un binaire avec radare2."""

    r2 = r2pipe.open(filepath, flags=['-2'])  # -2 = no stderr
    r2.cmd('aaa')  # Analyse complète

    results = {
        'info': json.loads(r2.cmd('ij')),
        'sections': json.loads(r2.cmd('iSj')),
        'imports': json.loads(r2.cmd('iij')),
        'strings': [],
        'suspicious_functions': [],
        'crypto_constants': [],
        'obfuscation_indicators': []
    }

    # Chercher les constantes cryptographiques
    # (AES S-Box, RC4 init, etc.)
    crypto_search = r2.cmd('/cr')
    if crypto_search:
        results['crypto_constants'] = crypto_search.strip().split('\n')

    # Analyser l'entropie par section
    for section in results['sections']:
        name = section.get('name', '')
        size = section.get('size', 0)
        if size > 0:
            paddr = section.get('paddr', 0)
            entropy_cmd = f'ph entropy {size} @ {paddr}'
            try:
                entropy = float(r2.cmd(entropy_cmd).strip())
                section['entropy'] = entropy
                if entropy > 7.0:
                    results['obfuscation_indicators'].append({
                        'type': 'high_entropy_section',
                        'section': name,
                        'entropy': entropy
                    })
            except (ValueError, TypeError):
                pass

    # Lister les fonctions et identifier les suspectes
    functions = json.loads(r2.cmd('aflj'))
    for func in functions:
        fname = func.get('name', '')
        fsize = func.get('size', 0)
        nbbs = func.get('nbbs', 0)  # Nombre de basic blocks

        # Heuristiques de détection d'obfuscation
        if nbbs > 50 and fsize < 500:
            # Beaucoup de blocs pour une petite fonction = CFF probable
            results['obfuscation_indicators'].append({
```

```
                'type': 'possible_cff',
                'function': fname,
                'blocks': nbbs,
                'size': fsize
            })

        # Fonctions avec noms suspects (anti-debug, crypto)
        suspicious_keywords = [
            'IsDebuggerPresent', 'NtQueryInformation',
            'CheckRemoteDebugger', 'OutputDebugString',
            'VirtualProtect', 'VirtualAlloc',
            'Crypt', 'Encrypt', 'Decrypt'
        ]
        for kw in suspicious_keywords:
            if kw.lower() in fname.lower():
                results['suspicious_functions'].append({
                    'name': fname,
                    'address': hex(func.get('offset', 0)),
                    'keyword': kw
                })

    # Extraire les chaînes intéressantes
    strings_json = json.loads(r2.cmd('izj'))
    for s in strings_json:
        content = s.get('string', '')
        if len(content) > 6:
            results['strings'].append({
                'value': content,
                'address': hex(s.get('vaddr', 0)),
                'section': s.get('section', ''),
                'type': s.get('type', '')
            })

    r2.quit()
    return results

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

    results = analyze_with_radare2(sys.argv[1])
    print(json.dumps(results, indent=2, default=str))
```

Questions frequentes

Comment mettre en place Disséquer l'Obscurité dans un environnement de production ?

La mise en place de Disséquer l'Obscurité en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Pourquoi Disséquer l'Obscurité est-il essentiel pour la securite des systemes d'information ?

Disséquer l'Obscurité constitue un element fondamental de la securite des systemes d'information car il permet de reduire significativement la surface d'attaque, d'ameliorer la detection des menaces et de renforcer la posture globale de securite de l'organisation face aux cybermenaces actuelles.

Quelles sont les bonnes pratiques pour Disséquer l'Obscurité en 2026 ?

Les bonnes pratiques pour Disséquer l'Obscurité en 2026 incluent l'adoption d'une approche Zero Trust, l'automatisation des controles de securite, la mise en place d'une veille continue sur les vulnerabilites et l'integration des recommandations des organismes de reference comme l'ANSSI et le NIST.

Pour approfondir ce sujet, consultez notre outil open-source malware-analysis-toolkit qui facilite l'analyse automatisée de malwares.

> Points clés à retenir
>
> - Analyse avec Ghidra
> - Questions frequentes
> - 10. Conclusion : tendances émergentes et perspectives
> - Analyse avec Ghidra (2)

10. Conclusion : tendances émergentes et perspectives

10.1 L'obfuscation assistée par intelligence artificielle

L'intersection entre l'intelligence artificielle et l'obfuscation de malwares représente la prochaine frontière majeure. Les travaux de recherche récents démontrent l'utilisation de **réseaux adversariaux génératifs (GAN)** et de **modèles de langage (LLM)** pour générer des variantes de malwares capables d'échapper aux classifieurs basés sur le machine learning. Des preuves de concept comme **DeepLocker** (IBM Research) et **MalGAN** illustrent le potentiel de ces approches.

Les tendances observées incluent :

- **Génération de code mort sémantiquement plausible** : au lieu d'insérer des NOP ou des opérations trivialement détectables, les modèles d'IA génèrent du code mort qui ressemble à du code fonctionnel légitime, rendant la détection par analyse de patterns beaucoup plus difficile
- **Optimisation adversariale des mutations** : utilisation de techniques de reinforcement learning pour trouver les transformations d'obfuscation qui maximisent l'évasion des moteurs de détection tout en minimisant l'impact sur les performances
- **Obfuscation contextuelle** : adaptation dynamique des techniques d'obfuscation en fonction de l'environnement cible détecté (type d'EDR, version de l'OS, présence de sandbox)

En réponse, la communauté défensive développe des approches de **déobfuscation assistée par IA**, notamment l'utilisation de modèles de type transformer pour la prédiction de la sémantique de code obfusqué, et l'application de techniques de **program synthesis** pour la reconstruction automatique du code original à partir de traces d'exécution symbolique.

10.2 WebAssembly : le nouveau vecteur d'obfuscation

WebAssembly (Wasm) émerge comme un vecteur d'obfuscation particulièrement préoccupant. Initialement conçu pour l'exécution de code performant dans les navigateurs web, Wasm est de plus en plus utilisé comme couche d'obfuscation hors navigateur :

- **Cryptominers Wasm** : des malwares comme **CoinHive** (aujourd'hui disparu) et ses successeurs utilisent Wasm pour du minage de cryptomonnaies directement dans le navigateur, rendant l'analyse statique des pages web compromise plus complexe
- **Wasm comme packer universel** : la compilation de code C/C++ malveillant vers Wasm puis son exécution via des runtimes comme **Wasmer** ou **Wasmtime** crée une couche d'abstraction supplémentaire que les outils d'analyse PE/ELF traditionnels ne gèrent pas nativement
- **Obfuscation du bytecode Wasm** : les mêmes techniques d'obfuscation (CFF, prédicats opaques, MBA) sont applicables au bytecode Wasm, mais l'outillage d'analyse est beaucoup moins mature que pour x86/ARM

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

Articles connexes

- IA Frameworks pour l'Analyse de Malwares - Deep Learning

Analyse avec Ghidra (2)

Les outils de déobfuscation Wasm comme **wasm-decompile** (de l'outil wabt), **JEB** (PNF Software), et les modules Wasm de Ghidra progressent rapidement mais restent en retard par rapport aux outils x86 matures.

10.3 Recommandations pour les analystes

Face à l'évolution constante des techniques d'obfuscation, les analystes de malwares doivent adopter une approche combinant :

- **Maîtrise des fondamentaux** : la compréhension profonde des architectures CPU (x86, ARM), des formats binaires (PE, ELF, Mach-O) et des systèmes d'exploitation reste indispensable. Aucun outil automatisé ne remplace cette expertise
- **Automatisation intelligente** : développer et maintenir des pipelines de triage combinant YARA, analyse PE, exécution symbolique et décompilation pour traiter le volume croissant d'échantillons
- **Veille technique continue** : suivre les publications de conférences comme **REcon**, **Black Hat**, **DEF CON**, **SSTIC**, et les travaux de recherche sur l'obfuscation (IEEE S&P, USENIX Security, CCS)

- **Collaboration communautaire** : partager les signatures YARA, les scripts de déobfuscation et les IOCs via des plateformes comme **MISP**, **VirusTotal**, **MalwareBazaar** et les ISACs sectoriels
- **Formation continue** : la déobfuscation est un domaine qui évolue aussi vite que les techniques offensives. L'investissement dans la formation (certifications GREM, FOR610, cours OpenSecurityTraining) est un impératif opérationnel

La course entre obfuscation et déobfuscation est fondamentalement asymétrique : l'attaquant n'a besoin que d'une seule technique fonctionnelle pour échapper à la détection, tandis que le défenseur doit couvrir l'ensemble de l'espace des transformations possibles. C'est pourquoi l'approche la plus résiliente combine analyse statique, analyse dynamique, exécution symbolique et intelligence sur les menaces en une stratégie de défense en profondeur.

Ressources recommandées

- **Practical Malware Analysis** - Sikorski & Honig (No Starch Press)
- **The IDA Pro Book** - Chris Eagle (No Starch Press)
- **Ghidra Software Reverse Engineering for Beginners** - A. Kabir
- **angr documentation** - docs.angr.io
- **Triton documentation** - triton-library.github.io
- **YARA documentation** - yara.readthedocs.io
- **OALabs** (YouTube) - Tutoriels pratiques de déobfuscation
- **MalwareUnicorn RE101/RE102** - Cours gratuits de reverse engineering

---

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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