AYI NEDJIMI Consultants

Reverse Engineering et Analyse de Malware : Guide Pratique

Catégorie : Techniques de Hacking Lecture : 9 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Guide pratique de reverse engineering et analyse de malware : Ghidra, IDA Pro, x64dbg, analyse statique et dynamique, unpacking, anti-debug, YARA.

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.

En matiere de CTF (Capture The Flag), les challenges de reverse engineering sont parmi les plus formateurs. Ils obligent les participants a maitriser les architectures processeur, les formats executables, les conventions d'appel et les techniques d'obfuscation. Cette experience pratique se transfere directement aux contextes professionnels d'analyse de menaces reelles. Guide pratique de reverse engineering et analyse de malware : Ghidra, IDA Pro, x64dbg, analyse statique et dynamique, unpacking, anti-debug, YARA. Les techniques offensives évoluent rapidement : reverse engineering analyse malware guide fait partie des compétences essentielles que tout pentester et red teamer doit maîtriser pour mener des missions réalistes. 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.

Cadre legal en France

En France, le reverse engineering est encadre par le Code de la propriete intellectuelle. L'article L122-6-1 autorise explicitement la decompilation d'un logiciel lorsqu'elle est necessaire pour obtenir les informations indispensables a l'interoperabilite avec d'autres logiciels. Par ailleurs, la recherche en securite beneficie d'un cadre plus favorable depuis la loi pour une Republique numerique de 2016, qui protege les lanceurs d'alerte en matiere de securite informatique (article L2321-4 du Code de la defense). L'ANSSI (Agence Nationale de la Securite des Systemes d'Information) encourage la divulgation responsable des vulnerabilites et offre un canal de signalement dedie.

Il est neanmoins essentiel de distinguer clairement les contextes legitimes (analyse de malware sur des echantillons obtenus dans le cadre d'un incident, recherche de vulnerabilites avec autorisation, education et CTF) des usages illicites (contournement de protections DRM sans base legale, utilisation a des fins de piratage). Toute activite de reverse engineering doit s'inscrire dans un cadre legal et ethique clair, avec une documentation rigoureuse des autorisations obtenues et des methodes employees.

Avertissement legal
Les techniques presentees dans cet article sont destinees exclusivement a des fins educatives, de recherche en securite et de reponse a incident. L'analyse de malware doit toujours etre realisee dans un environnement isole et securise. Assurez-vous de disposer des autorisations necessaires avant toute activite de reverse engineering sur des logiciels tiers.

Vos équipes savent-elles réagir face à une intrusion en cours ?

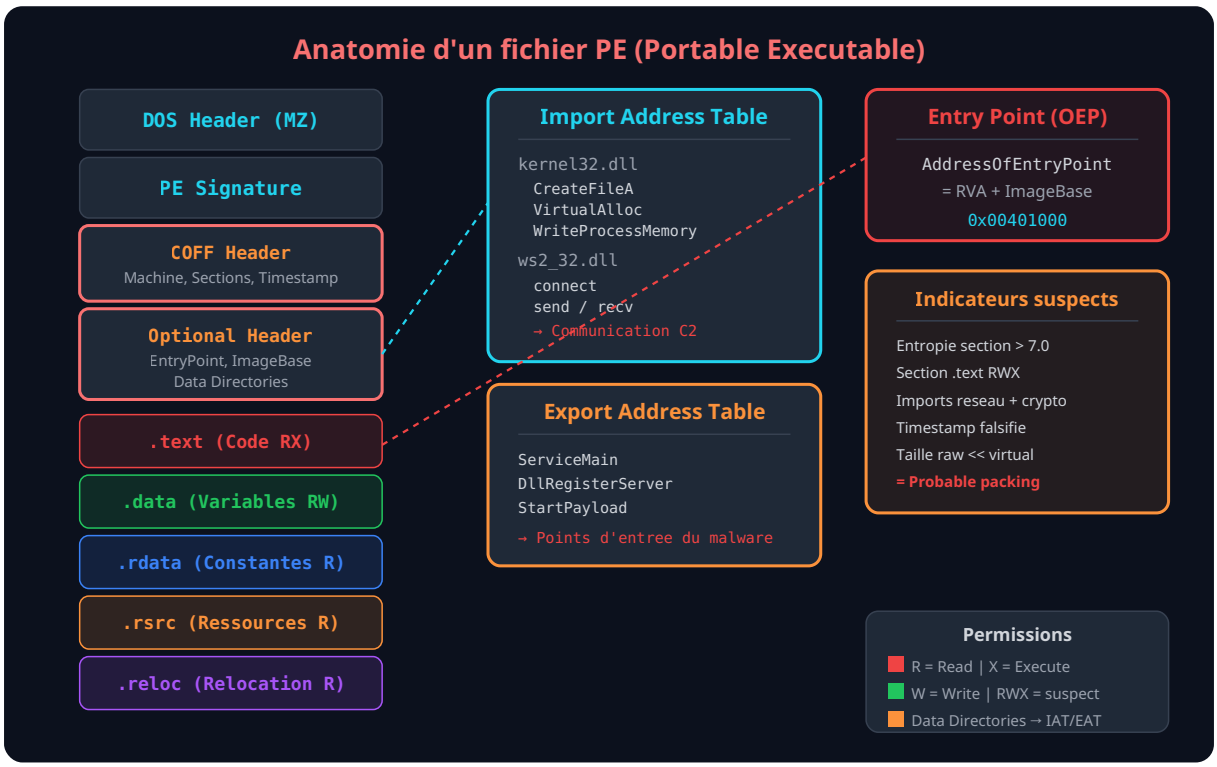
L'**Import Address Table (IAT)** est particulierement importante en analyse de malware : elle liste les fonctions importees depuis les DLL systeme (kernel32.dll, ntdll.dll, ws2_32.dll, etc.). L'analyse de l'IAT revele immediatement les capacites d'un binaire : des imports de `CreateRemoteThread`, `VirtualAllocEx` et `WriteProcessMemory` suggerent une injection de processus, tandis que `InternetOpenA`, `HttpSendRequest` indiquent des communications reseau.

Le format **ELF Linux** suit une structure similaire avec un ELF Header, des Program Headers (segments pour le chargement en memoire) et des Section Headers. Les sections `.plt` (Procedure Linkage Table) et `.got` (Global Offset Table) jouent un role equivalent a l'IAT pour la resolution dynamique des symboles. Le format **Mach-O** d'Apple utilise des load commands pour decrire la structure du binaire, avec des segments (similaires aux sections PE/ELF) et des tables de symboles pour les imports/exports.

Compilation et linking : du source au binaire

Le processus de compilation transforme le code source en binaire executable en quatre etapes principales : le **preprocessing** (expansion des macros, inclusion des headers), la **compilation** proprement dite (transformation en code assembleur), l'**assemblage** (conversion en code machine, creation de fichiers objets .o/.obj) et le **linking** (resolution des symboles, fusion des fichiers objets, creation de l'executable final). Comprendre ce processus aide le reverse engineer a identifier les patterns generes par les differents compilateurs (MSVC, GCC, Clang) et a reconnaitre les optimisations appliquees.

Le linking peut etre **statique** (les bibliotheques sont integrees directement dans l'executable, ce qui produit des binaires plus volumineux mais autonomes) ou **dynamique** (les bibliotheques sont chargees a l'execution via des DLL/SO). Le linking dynamique est plus courant et facilite l'analyse car les fonctions importees sont visibles dans l'IAT. Le linking statique, en revanche, complique l'analyse car les fonctions de la bibliotheque standard sont incorporees directement dans le code et doivent etre identifiees par signature (FLIRT signatures dans IDA Pro).



La navigation dans Ghidra s'organise autour de plusieurs vues complementaires : le **Listing** (desassemblage lineaire), le **Decompiler** (pseudo-code C), le **Function Graph** (representation visuelle du control flow), le **Symbol Tree** (arborescence des fonctions, imports, exports) et le **Data Type Manager** (gestion des structures et types). La vue decompilee est particulierement utile car elle transforme le code assembleur en un pseudo-code C comprehensible, meme pour les analystes qui ne maitrisent pas parfaitement l'assembleur.

```
# Script Ghidra (Python/Jython) pour lister les appels a des API suspectes
# A placer dans le Script Manager de Ghidra

from ghidra.program.model.symbol import SymbolType

suspicious_apis = [
    "VirtualAlloc", "VirtualAllocEx", "VirtualProtect",
    "CreateRemoteThread", "WriteProcessMemory", "NtWriteVirtualMemory",
    "CreateProcess", "ShellExecute", "WinExec",
    "InternetOpen", "HttpSendRequest", "URLDownloadToFile",
    "CryptEncrypt", "CryptDecrypt", "BCryptEncrypt",
    "RegSetValue", "RegCreateKey",
    "IsDebuggerPresent", "CheckRemoteDebuggerPresent"
]

symbol_table = currentProgram.getSymbolTable()
for symbol in symbol_table.getAllSymbols(True):
    if symbol.getSymbolType() == SymbolType.FUNCTION:
        for api in suspicious_apis:
            if api.lower() in symbol.getName().lower():
                refs = getReferencesTo(symbol.getAddress())
                print(f"[!] {symbol.getName()} referenced from {len(list(refs))}
locations")
                for ref in getReferencesTo(symbol.getAddress()):
                    func = getFunctionContaining(ref.getFromAddress())
                    if func:
                        print(f"    Called from: {func.getName()} @
{ref.getFromAddress()}")
```

IDA Pro et IDA Free

IDA Pro reste l'outil commercial de reference en reverse engineering, utilise par la majorite des equipes professionnelles de threat intelligence et de malware analysis. Sa force reside dans la qualite de sa detection de fonctions, son systeme de **FLIRT signatures** (Fast Library Identification and Recognition Technology) qui identifie automatiquement les fonctions de bibliotheques standard, et son decompilateur **Hex-Rays** qui produit un pseudo-code C de haute qualite.

La version gratuite **IDA Free** supporte les binaires x86 et x64 avec un decompilateur cloud. Les fonctionnalites cles d'IDA incluent la **vue graphe** (affichage du control flow sous forme de graphe avec les blocs basiques), les **cross-references (xrefs)** qui permettent de tracer tous les appels et references a une fonction ou une donnee, le **type system** riche pour appliquer des structures C/C++ aux donnees, et le support des **plugins IDAPython** pour automatiser l'analyse.

Pour analyser un malware avec x64dbg, commencez par charger l'executable et placer un **breakpoint sur l'entry point**. Identifiez les appels a `VirtualAlloc` ou `VirtualProtect` qui indiquent souvent une decompression ou un dechiffrement de code en memoire. Utilisez les **breakpoints conditionnels** pour arreter l'execution uniquement quand un registre contient une valeur specifique (par exemple, quand EAX contient l'adresse d'un buffer dechiffre). Le plugin **ScyllaHide** masque la presence du debugger pour contourner les techniques anti-debug basiques.

```
// Script x64dbg - Tracer les appels VirtualAlloc
// Placer un BP conditionnel sur VirtualAlloc
bp VirtualAlloc
SetBreakpointCommand VirtualAlloc, "log 'VirtualAlloc({arg.get(0)}, size={arg.get(1)},
type={arg.get(2)}, protect={arg.get(3)})'; run"

// Breakpoint hardware en ecriture sur une adresse memoire
bphws 0x00401000, "w", 4

// Logger les appels CreateFile
bp CreateFileA
SetBreakpointCommand CreateFileA, "log 'CreateFileA: {s:arg.get(0)}'; run"

// Dumper une region memoire
savedata "C:\\dump\\payload.bin", 0x10000, 0x5000
```

WinDbg pour l'analyse kernel

WinDbg (Windows Debugger) est indispensable pour le debugging kernel-mode, necessaire pour analyser les rootkits et les drivers malveillants. Configure en mode kernel debugging via une connexion serie virtuelle, COM pipe ou reseau entre deux VMs, WinDbg permet d'inspecter les structures du noyau Windows (EPROCESS, ETHREAD, DRIVER_OBJECT), de tracer les appels systeme et de detecter les hooks SSDT/IDT. Pour une analyse approfondie des techniques d'exploitation kernel, consultez notre article sur l'exploitation kernel Windows.

Monitoring API et comportemental

API Monitor intercepte et enregistre tous les appels API Windows effectues par un processus. Il permet de filtrer par categorie (fichiers, registre, reseau, processus, threads) et de visualiser les parametres et valeurs de retour de chaque appel. **Process Monitor (ProcMon)** de Sysinternals capture les operations sur le systeme de fichiers, le registre et les processus en temps reel. **Process Hacker** offre une vue detaillee des processus en cours, de leurs threads, handles, connexions reseau et modules charges.

Pour la capture reseau, **Wireshark** enregistre tout le trafic genere par le malware. **FakeNet-NG** de Mandiant va plus loin en interceptant et en simulant les reponses des serveurs distants, ce qui permet au malware de poursuivre son execution meme sans connexion Internet. Il supporte les protocoles HTTP, HTTPS, DNS, TCP et UDP generiques.

Analyse automatisee en sandbox

Les sandbox automatisees executent les echantillons et produisent des rapports detailles. **ANY.RUN** offre une analyse interactive en temps reel dans un navigateur, avec la possibilite d'interagir avec le malware (cliquer sur des boutons, fermer des boites de dialogue). **Joe Sandbox** genere des rapports extremement detailles incluant le graphe de comportement, les IOC, les regles YARA matchees et la classification MITRE ATT&CK. **CAPE (Malware Configuration And Payload Extraction)**, basee sur Cuckoo Sandbox, excelle dans l'extraction automatique de configurations de malware et de payloads dechiffres. Ces plateformes sont particulierement utiles pour le triage a grande echelle lorsque le volume d'echantillons depasse la capacite d'analyse manuelle.

Les malwares detectent les environnements virtualises pour eviter l'analyse en sandbox. Les techniques de detection de VM incluent :

- **Instruction CPUID** : l'hypervisor bit (bit 31 de ECX pour CPUID leaf 1) indique la presence d'un hyperviseur. Le vendor string (CPUID leaf 0x40000000) revele le type : "VMwareVMware", "Microsoft Hv", "KVMKVMKVM".
- **Verification du registre** : cles HKLM\SOFTWARE\VMware, HKLM\SYSTEM\CurrentControlSet\Services\VBoxGuest.
- **Adresse MAC** : les 3 premiers octets identifient le fabricant (00:0C:29 = VMware, 08:00:27 = VirtualBox, 00:15:5D = Hyper-V).
- **WMI queries** : Win32_ComputerSystem.Model contient "VirtualBox" ou "VMware".
- **Fichiers et processus** : presence de vmtoolsd.exe, VBoxService.exe, fichiers vmware*.sys.
- **Resolution d'ecran / nombre de CPU / taille RAM** : les sandbox ont souvent une configuration minimale (1 CPU, 2 Go RAM, 1024x768).

Les techniques **anti-sandbox** specifiques ciblent l'environnement d'analyse automatisee : verification de l'**interaction utilisateur** (mouvements de souris, frappes clavier), **delais d'execution** (sleep de plusieurs minutes pour depasser le timeout de la sandbox), verification du **nombre de fichiers recents**, du **nombre de programmes installes** ou de l'**historique du navigateur** (une machine d'analyse fraichement installee sera suspectee). Certains malwares verifient meme la presence d'un nom d'utilisateur ou d'un hostname typique des sandbox (malware, sandbox, analysis, test).

Obfuscation de code

L'obfuscation rend le code difficile a comprendre sans empecher son execution. Le **control flow flattening** transforme la structure conditionnelle naturelle du code en un switch/case geant dans une boucle, eliminant la hierarchie logique visible dans le graphe de controle. Les **opaque predicates** sont des conditions qui semblent complexes mais dont le resultat est toujours le meme (toujours vrai ou toujours faux), ajoutant de faux chemins d'execution. Le **chiffrement de strings** remplace chaque chaine en clair par un appel a une fonction de dechiffrement, rendant l'analyse des strings completement inefficace. Les techniques de Living-off-the-Land combinent souvent ces obfuscations avec l'utilisation d'outils systeme legitimes.

Une fois l'OEP atteint, utilisez le plugin **Scylla** integre a x64dbg pour dumper le processus et reconstruire l'IAT. Scylla analyse la memoire du processus, identifie les imports resolus, et reconstruit une table d'importation valide dans le PE dumpe. L'outil **pe-sieve** de hasherezade peut egalement detecter et extraire automatiquement les modules depackes en memoire.

Desobfuscation de strings et emulation

La desobfuscation des chaines de caracteres est souvent la premiere etape apres l'unpacking. Quand le malware utilise un chiffrement XOR simple ou une fonction de dechiffrement custom, un **script Python** suffit pour decoder toutes les strings :

```
# Desobfuscation XOR avec cle rotative
def xor_decrypt(data, key):
    return bytes([b ^ key[i % len(key)] for i, b in enumerate(data)])

# Exemple : dechiffrer un buffer avec une cle de 4 octets
encrypted = bytes.fromhex("4a1b3c2d5e6f7081...")
key = bytes.fromhex("deadbeef")
print(xor_decrypt(encrypted, key))

# Script Ghidra pour trouver et decoder les appels de dechiffrement
# Identifier le pattern : push encrypted_addr; push key; call decrypt_func
from ghidra.program.util import DefinedDataIterator
for ref in getReferencesTo(toAddr(0x00401230)):  # adresse de decrypt_func
    caller = ref.getFromAddress()
    # Extraire les parametres pushes avant l'appel
    # ... logique d'extraction des arguments
```

Pour les cas plus complexes, l'**emulation** permet d'executer selectivement des parties du code sans lancer le malware complet. **Unicorn Engine** est un framework d'emulation CPU leger qui supporte x86, ARM, MIPS et d'autres architectures. **Qiling** va plus loin en emulant non seulement le CPU mais aussi le systeme d'exploitation (appels systeme, structures du noyau, gestion des fichiers), permettant d'executer des fonctions individuelles du malware dans un environnement completement controle. Ces outils sont particulierement utiles pour dechiffrer des configurations, des URLs de C2 ou des payloads secondaires sans risque d'infection.

```
# Emulation avec Unicorn pour decoder des strings
from unicorn import *
from unicorn.x86_const import *

# Initialiser l'emulateur x86 32 bits
mu = Uc(UC_ARCH_X86, UC_MODE_32)

# Mapper la memoire et charger le code du malware
mu.mem_map(0x400000, 0x10000)  # section .text
mu.mem_map(0x410000, 0x10000)  # section .data

# Charger les sections depuis le PE
with open("malware.exe", "rb") as f:
    code = f.read()
mu.mem_write(0x400000, code[0x400:0x10400])  # .text
mu.mem_write(0x410000, code[0x10400:0x20400])  # .data

# Configurer les registres (stack)
mu.mem_map(0x7F0000, 0x10000)  # stack
mu.reg_write(UC_X86_REG_ESP, 0x7FFFF0)
mu.reg_write(UC_X86_REG_EBP, 0x7FFFF0)

# Emuler la fonction de dechiffrement (adresse 0x401230)
mu.emu_start(0x401230, 0x4012FF)  # start, end

# Lire le resultat dechiffre
result = mu.mem_read(0x410100, 256)
print(bytes(result).split(b'\x00')[0].decode())
```

Les malwares utilisant des techniques de persistance UEFI ou firmware, comme decrits dans notre article sur les UEFI bootkits, necessitent souvent des techniques d'unpacking et d'emulation specifiques pour analyser les implants boot-level.

Comment mettre en place Reverse Engineering et Analyse de Malware dans un environnement de production ?

La mise en place de Reverse Engineering et Analyse de Malware 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 Reverse Engineering et Analyse de Malware est-il essentiel pour la securite des systemes d'information ?

Reverse Engineering et Analyse de Malware 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.

Cette technique Reverse Engineering et Analyse de Malware : Guide Pratique est-elle utilisable dans un pentest autorisé ?

Oui, à condition d'avoir une lettre de mission signée définissant le périmètre, les horaires et les techniques autorisées. Documentez chaque action et restez dans le scope défini.

Pour approfondir ce sujet, consultez notre outil open-source exploit-framework-python qui facilite le développement et le test d'exploits.

Points cles a retenir

- Le triage initial (file, strings, FLOSS, entropie, VirusTotal) oriente l'ensemble de l'analyse en 15 minutes.
- Ghidra (gratuit) et IDA Pro (commercial) sont les piliers de l'analyse statique, avec leurs decompilateurs respectifs.
- L'analyse dynamique dans une sandbox isolee (FlareVM + REMnux) revele le comportement reel du malware.
- Les techniques anti-analyse (packing, anti-debug, anti-VM) se contournent avec methode et les bons outils.
- L'unpacking manuel (OEP finding + Scylla) et l'emulation (Unicorn/Qiling) permettent de retrouver le code original.
- Les regles YARA transforment l'analyse en detection operationnelle deployable dans le SIEM et l'EDR.
- La methodologie en 6 etapes garantit une analyse complete et un rapport exploitable.
- Le partage via MISP/STIX amplifie l'impact de chaque analyse.

References et ressources externes

- Ghidra - NSA Software Reverse Engineering — Plateforme d'analyse statique open source de reference
- x64dbg — Debugger open source pour Windows x86/x64
- YARA - Pattern matching tool — Standard de detection de malware par patterns

- MITRE ATT&CK Framework — Base de connaissances des tactiques et techniques adverses
- FlareVM - Mandiant — Distribution Windows pour l'analyse de malware
- REMnux — Distribution Linux pour la retro-ingenierie de malware
- MalwareBazaar - abuse.ch — Partage d'echantillons de malware
- MISP Project — Plateforme de partage d'indicateurs de menaces

Sources et références : MITRE ATT&CK · OWASP Testing Guide

Articles connexes

- Hacking WordPress : Fondamentaux, Vulnérabilités : Guide

FAQ

Qu'est-ce que Reverse Engineering et Analyse de Malware ?

Reverse Engineering et Analyse de Malware désigne l'ensemble des concepts, techniques et méthodologies abordés dans cet article. Les fondamentaux sont détaillés dans les premières sections du guide.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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