AYI NEDJIMI Consultants

Anti-Rétro-Ingénierie APT - Techniques d'Évasion Avancées

Catégorie : Retro-Ingenierie Lecture : 4 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Analyse technique des techniques anti-rétro-ingénierie utilisées par les malwares APT : anti-debug, anti-sandbox, obfuscation strings, Lazarus, Turla.

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.

L'anti-debugging est la première ligne de défense des malwares APT. Ces techniques détectent la présence d'un débogueur (OllyDbg, x64dbg, WinDbg, GDB) et altèrent le comportement du malware — souvent en terminant le processus, en corrompant les données, ou en empruntant un chemin d'exécution leurre. Pour approfondir, consultez notre article sur Reverse Engineering Dotnet Decompilation Analyse. Analyse technique des techniques anti-rétro-ingénierie utilisées par les malwares APT : anti-debug, anti-sandbox, obfuscation strings, Lazarus, Turla. La rétro-ingénierie est une discipline fondamentale en analyse de malware et en recherche de vulnérabilités. Anti-Rétro-Ingénierie APT - Techniques d'Évasion Avancées couvre les techniques avancées utilisées par les analystes. Nous abordons notamment : questions frequentes, 10. conclusion. 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.

2.1 Windows API : IsDebuggerPresent et PEB

La méthode la plus basique mais toujours utilisée consiste à interroger le **Process Environment Block (PEB)**. Le champ `BeingDebugged` à l'offset `0x002` du PEB est mis à 1 par le système lorsqu'un débogueur est attaché. Pour approfondir, consultez notre article sur Deobfuscation Malwares Polymorphes.

```
// Vérification directe du PEB (x64)
#include <windows.h>
#include <intrin.h>

BOOL check_peb_debugger() {
    // Méthode 1 : API standard
    if (IsDebuggerPresent())
        return TRUE;

    // Méthode 2 : Accès direct au PEB via GS segment (x64)
    PPEB peb = (PPEB)__readgsqword(0x60);
    if (peb->BeingDebugged)
        return TRUE;

    // Méthode 3 : NtGlobalFlag (offset 0xBC en x64)
    // Valeur 0x70 = FLG_HEAP_ENABLE_TAIL_CHECK |
    //               FLG_HEAP_ENABLE_FREE_CHECK |
    //               FLG_HEAP_VALIDATE_PARAMETERS
    DWORD ntGlobalFlag = *(DWORD*)((BYTE*)peb + 0xBC);
    if (ntGlobalFlag & 0x70)
        return TRUE;

    return FALSE;
}

// Méthode 4 : NtQueryInformationProcess
BOOL check_remote_debugger() {
    BOOL debuggerPresent = FALSE;

    typedef NTSTATUS (NTAPI *pNtQIP)(
        HANDLE, ULONG, PVOID, ULONG, PULONG);

    pNtQIP NtQueryInformationProcess = (pNtQIP)
        GetProcAddress(GetModuleHandleA("ntdll.dll"),
                       "NtQueryInformationProcess");

    // ProcessDebugPort = 7
    DWORD_PTR debugPort = 0;
    NtQueryInformationProcess(GetCurrentProcess(), 7,
                              &debugPort, sizeof(debugPort), NULL);
    if (debugPort != 0)
        return TRUE;

    // ProcessDebugObjectHandle = 0x1E
    HANDLE debugObject = NULL;
    NTSTATUS status = NtQueryInformationProcess(
        GetCurrentProcess(), 0x1E,
        &debugObject, sizeof(debugObject), NULL);
    if (status == 0 && debugObject != NULL)
        return TRUE;

    return FALSE;
}
```

2.2 Timing Checks avec RDTSC

Disposez-vous en interne des compétences de rétro-ingénierie nécessaires pour analyser un malware ciblant votre organisation ?

Les débogueurs introduisent des délais mesurables lors du single-stepping. L'instruction `RDTSC` (Read Time-Stamp Counter) mesure les cycles CPU avec une précision nanoseconde, permettant de détecter ces ralentissements.

```
// Détection par timing RDTSC
#include <intrin.h>

BOOL check_timing_rdtsc() {
    unsigned __int64 t1, t2, t3;

    // Mesure 1 : instructions triviales
    t1 = __rdtsc();

    // Bloc de code anodin qui sera single-stepped
    volatile int x = 0;
    for (int i = 0; i < 100; i++) x += i;

    t2 = __rdtsc();

    // Seuil : ~500K cycles normaux, >10M avec debugger
    if ((t2 - t1) > 10000000)
        return TRUE;

    // Mesure 2 : QueryPerformanceCounter (alternative)
    LARGE_INTEGER freq, c1, c2;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&c1);

    Sleep(0);  // Yield minimal

    QueryPerformanceCounter(&c2);

    // >1ms = probable debugger
    double elapsed = (double)(c2.QuadPart - c1.QuadPart) / freq.QuadPart;
    if (elapsed > 0.001)
        return TRUE;

    return FALSE;
}

// Variante assembleur inline (x86)
// Utilisé par APT41 dans ShadowPad
BOOL __declspec(naked) timing_check_asm() {
    __asm {
        rdtsc
        mov ecx, eax    ; stocker low DWORD du TSC
        rdtsc
        sub eax, ecx    ; delta
        cmp eax, 0xFF   ; seuil
        ja  debugged
        xor eax, eax    ; return FALSE
        ret
    debugged:
        mov eax, 1      ; return TRUE
        ret
    }
}
```

2.3 Exception-Based Anti-Debug

Les débogueurs interceptent certaines exceptions avant le handler du programme. En générant des exceptions contrôlées (INT 2D, INT 3, division par zéro), le malware peut détecter si le flux d'exception a été modifié.

```
// Anti-debug par exception handler (SEH)
BOOL check_exception_handler() {
    __try {
        // INT 2D : Debug Break sous Windows
        // Si un debugger est attaché, il consomme l'exception
        // et le handler ne sera jamais appelé
        __asm {
            __emit 0xCD  // INT
            __emit 0x2D  // 0x2D
            nop
        }
        // Si on arrive ici SANS exception, debugger détecté
        return TRUE;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // Exception attrapée = pas de debugger
        return FALSE;
    }
}

// Variante avec hardware breakpoint detection
BOOL check_hardware_breakpoints() {
    CONTEXT ctx;
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;

    if (!GetThreadContext(GetCurrentThread(), &ctx))
        return FALSE;

    // DR0-DR3 contiennent les adresses des HW breakpoints
    if (ctx.Dr0 || ctx.Dr1 || ctx.Dr2 || ctx.Dr3)
        return TRUE;

    return FALSE;
}
```

Note analyste : Les malwares APT avancés combinent typiquement 5 à 8 vérifications anti-debug différentes, exécutées à intervalles réguliers tout au long de l'exécution, pas seulement au démarrage. Le groupe Lazarus est connu pour intégrer des timing checks dans ses boucles de communication C2.

Cas concret

L'analyse du malware Pegasus par le Citizen Lab et Amnesty International a révélé un arsenal d'exploitation zero-click ciblant iOS. La rétro-ingénierie des exploits FORCEDENTRY a montré une utilisation innovante de fichiers PDF malveillants traités par le moteur de rendu d'iMessage, sans aucune interaction de la victime.

Au-delà de la détection de VM pure, les malwares APT profilent l'environnement pour identifier les caractéristiques d'une sandbox automatisée : peu de fichiers utilisateur, pas d'historique de navigation, résolution d'écran par défaut, etc.

```
"""
Techniques de fingerprinting anti-sandbox
Implémentées en Python pour l'analyse et la reproduction
"""
import os
import subprocess
import ctypes
import time

class SandboxDetector:
    def __init__(self):
        self.checks = []

    def check_disk_size(self, min_gb=60):
        """Sandboxes utilisent souvent des disques < 60 GB"""
        import shutil
        total, _, _ = shutil.disk_usage("C:\\")
        total_gb = total / (1024**3)
        return total_gb < min_gb

    def check_ram(self, min_gb=4):
        """Sandboxes avec RAM limitée"""
        import psutil
        ram_gb = psutil.virtual_memory().total / (1024**3)
        return ram_gb < min_gb

    def check_cpu_count(self, min_cores=2):
        """VMs de sandbox souvent mono-coeur"""
        return os.cpu_count() < min_cores

    def check_uptime(self, min_minutes=30):
        """Sandbox uptime est souvent très court"""
        uptime_ms = ctypes.windll.kernel32.GetTickCount64()
        uptime_min = uptime_ms / 60000
        return uptime_min < min_minutes

    def check_recent_files(self, min_files=20):
        """Vrais PC ont un historique de fichiers récents"""
        recent = os.path.expandvars(
            r"%APPDATA%\Microsoft\Windows\Recent")
        if not os.path.exists(recent):
            return True
        count = len(os.listdir(recent))
        return count < min_files

    def check_mouse_movement(self, duration_sec=10):
        """Sandboxes n'ont pas de mouvement de souris humain"""
        import ctypes

        class POINT(ctypes.Structure):
            _fields_ = [("x", ctypes.c_long), ("y", ctypes.c_long)]

        positions = set()
        for _ in range(duration_sec * 10):
            pt = POINT()
            ctypes.windll.user32.GetCursorPos(ctypes.byref(pt))
            positions.add((pt.x, pt.y))
            time.sleep(0.1)

        # Moins de 3 positions uniques = pas d'humain
        return len(positions) < 3
```

```
    def check_mac_vendors(self):
        """Détection des OUI VMware/VBox/QEMU"""
        vm_macs = [
            "00:0C:29",  # VMware
            "00:50:56",  # VMware
            "08:00:27",  # VirtualBox
            "52:54:00",  # QEMU/KVM
            "00:1C:42",  # Parallels
        ]
        result = subprocess.run(
            ["getmac", "/fo", "csv", "/nh"],
            capture_output=True, text=True)
        for mac_prefix in vm_macs:
            if mac_prefix.lower() in result.stdout.lower():
                return True
        return False

    def check_username(self):
        """Sandboxes utilisent des noms communs"""
        sandbox_names = [
            "sandbox", "malware", "virus", "sample",
            "test", "john", "user", "admin", "analyst",
            "cuckoo", "vmuser", "computername"
        ]
        username = os.environ.get("USERNAME", "").lower()
        hostname = os.environ.get("COMPUTERNAME", "").lower()
        for name in sandbox_names:
            if name in username or name in hostname:
                return True
        return False

    def run_all(self):
        """Exécute toutes les vérifications"""
        results = {
            "disk_small": self.check_disk_size(),
            "ram_low": self.check_ram(),
            "cpu_low": self.check_cpu_count(),
            "uptime_short": self.check_uptime(),
            "few_recent": self.check_recent_files(),
            "vm_mac": self.check_mac_vendors(),
            "sandbox_name": self.check_username(),
        }
        # Seuil : 3+ indicateurs = sandbox probable
        score = sum(results.values())
        return score >= 3, results, score
```

Technique APT41 : Le groupe APT41 utilise un système de « scoring » similaire dans son implant ShadowPad. Plutôt qu'un seul check binaire, il cumule un score de 0 à 20 basé sur la pondération de chaque indicateur. Le malware ne s'exécute que si le score est inférieur à un seuil configuré par l'opérateur C2.

Savez-vous identifier les techniques d'anti-analyse utilisées par les malwares modernes ?

```
// API hashing CRC32 - technique Equation Group
#define HASH_KERNEL32_LOADLIBRARY   0xEC0E4E8E
#define HASH_KERNEL32_GETPROCADDR   0x7C0DFCAA
#define HASH_NTDLL_NTWRITEFILE      0x95A28A3B

DWORD crc32_hash(const char* str) {
    DWORD hash = 0xFFFFFFFF;
    while (*str) {
        hash ^= (unsigned char)(*str++);
        for (int i = 0; i < 8; i++) {
            if (hash & 1)
                hash = (hash >> 1) ^ 0xEDB88320;
            else
                hash >>= 1;
        }
    }
    return hash ^ 0xFFFFFFFF;
}

// Résolution d'API par hash via PEB walking
FARPROC resolve_api_by_hash(DWORD target_hash) {
    // Accès au PEB
    PPEB peb = (PPEB)__readgsqword(0x60);
    PPEB_LDR_DATA ldr = peb->Ldr;

    // Parcours de la liste des modules chargés
    PLIST_ENTRY head = &ldr->InMemoryOrderModuleList;
    PLIST_ENTRY curr = head->Flink;

    while (curr != head) {
        PLDR_DATA_TABLE_ENTRY entry = CONTAINING_RECORD(
            curr, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);

        // Parse la table d'exports du module
        PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)entry->DllBase;
        PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)(
            (BYTE*)dos + dos->e_lfanew);

        DWORD exportRVA = nt->OptionalHeader.DataDirectory[0]
                          .VirtualAddress;
        if (exportRVA == 0) { curr = curr->Flink; continue; }

        PIMAGE_EXPORT_DIRECTORY exports = (PIMAGE_EXPORT_DIRECTORY)(
            (BYTE*)dos + exportRVA);

        DWORD* names = (DWORD*)((BYTE*)dos + exports->AddressOfNames);
        WORD*  ords  = (WORD*)((BYTE*)dos + exports->AddressOfNameOrdinals);
        DWORD* funcs = (DWORD*)((BYTE*)dos + exports->AddressOfFunctions);

        for (DWORD i = 0; i < exports->NumberOfNames; i++) {
            char* name = (char*)((BYTE*)dos + names[i]);
            if (crc32_hash(name) == target_hash) {
                return (FARPROC)((BYTE*)dos + funcs[ords[i]]);
            }
        }
        curr = curr->Flink;
    }
    return NULL;
}
```

Outil de l'analyste : HashDB (plugin IDA Pro par OALabs) et Shellcode Hashes (base de données communautaire) permettent de résoudre automatiquement les hash d'API les plus courants. Pour les hash custom, il faut recréer la fonction de hashing et la bruteforcer contre la liste complète des exports Windows.

```
// Script Frida pour bypass des anti-RE en temps réel
// Usage : frida -l bypass_anti_re.js -f malware.exe

console.log("[*] Anti-RE Bypass Script loaded");

// 1. Hook IsDebuggerPresent
Interceptor.attach(Module.findExportByName("kernel32.dll",
    "IsDebuggerPresent"), {
    onLeave: function(retval) {
        retval.replace(0);  // Toujours retourner FALSE
        console.log("[+] IsDebuggerPresent -> FALSE");
    }
});

// 2. Hook NtQueryInformationProcess
var ntdll = Module.findBaseAddress("ntdll.dll");
var pNtQIP = Module.findExportByName("ntdll.dll",
    "NtQueryInformationProcess");

Interceptor.attach(pNtQIP, {
    onEnter: function(args) {
        this.infoClass = args[1].toInt32();
        this.outBuffer = args[2];
    },
    onLeave: function(retval) {
        // ProcessDebugPort (7) ou ProcessDebugObjectHandle (0x1E)
        if (this.infoClass === 7 || this.infoClass === 0x1E) {
            this.outBuffer.writeU64(0);
            console.log("[+] NtQueryInformationProcess(" +
                        this.infoClass + ") -> 0");
        }
    }
});

// 3. Hook GetTickCount64 (anti-timing)
var originalTick = null;
Interceptor.attach(Module.findExportByName("kernel32.dll",
    "GetTickCount64"), {
    onEnter: function() {
        if (!originalTick) {
            originalTick = Date.now();
        }
    },
    onLeave: function(retval) {
        // Simuler un uptime de 3 heures (éviter détection sandbox)
        var fakeUptime = 10800000 + (Date.now() - originalTick);
        retval.replace(ptr(fakeUptime));
    }
});

// 4. Hook CPUID (anti-VM)
// Note : nécessite Stalker pour intercepter CPUID
var cm = new CModule(`
#include
extern void on_cpuid(GumCpuContext *ctx) {
    // Si leaf 0x40000000 (hypervisor vendor), retourner vide
    if (ctx->rax == 0x40000000) {
        ctx->rbx = 0;
        ctx->rcx = 0;
        ctx->rdx = 0;
    }
    // Si leaf 1, masquer le bit hypervisor (ECX bit 31)
```

```
    if (ctx->rax == 1) {
        ctx->rcx &= ~(1 << 31);
    }
}
`);

// 5. Patcher les checks de nom d'utilisateur/hostname
Interceptor.attach(Module.findExportByName("kernel32.dll",
    "GetComputerNameA"), {
    onLeave: function(retval) {
        var buf = this.context.rcx || this.context.r8;
        // Remplacer par un nom réaliste
        buf.writeAnsiString("DESKTOP-A7B3C9D");
        console.log("[+] ComputerName -> DESKTOP-A7B3C9D");
    }
});

console.log("[*] All hooks installed. Anti-RE bypassed.");
```

9.3 Fuzzing avec AFL++ pour découvrir les chemins cachés

```
#!/bin/bash
# Utilisation d'AFL++ pour fuzzer un malware et découvrir
# les chemins d'exécution cachés derrière les anti-RE

# 1. Compiler le harness avec instrumentation AFL
export AFL_CC_COMPILER=LLVM
afl-clang-lto -o harness harness.c \
    -fsanitize=address \
    -DFUZZ_MODE=1

# 2. Créer le corpus initial (inputs connus)
mkdir -p corpus/
echo -n "NORMAL_INPUT" > corpus/seed1.bin
echo -n "\x00\x00\x00\x00" > corpus/seed2.bin

# 3. Créer le dictionnaire de tokens du malware
cat > dict.txt << 'EOF'
# Tokens extraits du malware
"VMware"
"VBOX"
"Sandbox"
"cuckoo"
"\x0F\x31"  # RDTSC
"\xCD\x2D"  # INT 2D
"\x64\xA1\x30\x00\x00\x00"  # PEB access x86
EOF

# 4. Lancer le fuzzing multi-coeur
afl-fuzz -i corpus/ -o findings/ \
    -x dict.txt \
    -m 512 \
    -t 5000 \
    -M master \
    -- ./harness @@

# En parallèle sur d'autres coeurs :
# afl-fuzz -i corpus/ -o findings/ -S slave01 -- ./harness @@
# afl-fuzz -i corpus/ -o findings/ -S slave02 -- ./harness @@

# 5. Analyser les crashes et les chemins découverts
afl-tmin -i findings/master/crashes/ -o minimized/ -- ./harness @@

echo "[*] Chemins uniques découverts :"
ls findings/master/queue/ | wc -l

echo "[*] Crashes trouvés :"
ls findings/master/crashes/ | wc -l
```

1. Introduction	2. Techniques Anti-Debugging	2. Techniques Anti-Debugging : analyse approfondie
Implementation	Renforcement de la securite globale	Complexite de mise en oeuvre
Monitoring	Detection proactive des menaces	Ressources necessaires
Conformite	Alignement aux referentiels	Cout de certification

Pour approfondir ce sujet, consultez notre outil open-source reverse-engineering-scripts qui facilite l'assistance à la rétro-ingénierie de binaires.

Questions frequentes

Comment mettre en place Anti dans un environnement de production ?

La mise en place de Anti 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 Anti est-il essentiel pour la securite des systemes d'information ?

Anti 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 Anti en 2026 ?

Les bonnes pratiques pour Anti 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.

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

Articles connexes

- Malwares Mobiles & IA - Rétro-Ingénierie Cross-Platform
- Chasse aux Fantômes : Rétro-Ingénierie

10. Conclusion

L'analyse des techniques anti-RE déployées par les groupes APT révèle une **industrialisation de l'évasion**. Les implants modernes ne reposent plus sur une ou deux astuces, mais sur une architecture défensive multicouche où chaque protection renforce les autres.

Les tendances émergentes incluent :

- **IA offensive** : utilisation de modèles de langage pour générer du code polymorphe contextuel, rendant chaque sample unique et indétectable par signatures statiques
- **Anti-RE basée sur l'environnement** : le payload ne se déchiffre que sur la machine cible (clé dérivée du hostname, GUID machine, certificats installés), rendant l'analyse impossible sur une machine tierce
- **Firmware-level evasion** : implants dans le BIOS/UEFI, l'Intel ME, ou les contrôleurs BMC, invisibles à l'OS et survivant aux réinstallations

- **Communication covert channel** : DNS-over-HTTPS, steganographie dans les images de profil de réseaux sociaux, communications via des services cloud légitimes (OneDrive, Notion, Telegram)

Face à cette sophistication croissante, l'analyste doit maintenir une approche systématique : identifier les couches de protection, les neutraliser séquentiellement, et documenter chaque technique pour alimenter les signatures de détection. Les outils comme **Frida**, **IDAPython** et **AFL++** sont les alliés essentiels de cette contre-analyse.

Recommandation : Maintenez un catalogue interne des techniques anti-RE rencontrées, avec les scripts de contournement associés. La réutilisation de code entre campagnes APT est fréquente — une technique identifiée dans un sample Lazarus pourra être retrouvée dans une future campagne.

> Points clés à retenir
>
> • 2.1 Windows API : IsDebuggerPresent et PEB
> • 2.2 Timing Checks avec RDTSC
> • 2.3 Exception-Based Anti-Debug
> • Questions frequentes
> • 10. Conclusion

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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