AYI NEDJIMI Consultants

Memory Forensics : Strategies de Detection et de Remediation

Catégorie : Forensics Lecture : 3 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Guide expert d Memory Forensics : Volatility3/WinPMEM — Détection. Expert en cybersécurité et intelligence artificielle. Guide technique complet.

3.1 Techniques d'Injection de Code

Les backdoors modernes utilisent diverses techniques d'injection pour s'exécuter dans le contexte de processus légitimes. Volatility3 permet de détecter ces injections par plusieurs méthodes :

Process Hollowing Detection

Le process hollowing consiste à vider un processus légitime de son code et le remplacer par du code malveillant :

```
# Détection de process hollowing
def detect_process_hollowing(proc):
    # Vérification de l'alignement PEB/Image Base
    peb = proc.Peb
    image_base = peb.ImageBaseAddress

    # Lecture des headers PE en mémoire
    pe_header = read_pe_header(context, proc, image_base)

    # Comparaison avec le fichier sur disque
    disk_path = proc.SeAuditProcessCreationInfo.ImageFileName
    disk_pe = parse_pe_from_disk(disk_path)

    if pe_header.entry_point != disk_pe.entry_point:
        print(f"Process hollowing detected in PID {proc.UniqueProcessId}")
        print(f"Memory EP: 0x{pe_header.entry_point:08x} vs Disk EP:
0x{disk_pe.entry_point:08x}")

    # Vérification des sections VAD
    for vad in proc.VadRoot.traverse():
        if vad.is_executable() and not vad.is_image():
            print(f"Suspicious executable VAD at 0x{vad.Start:016x}")
```

Reflective DLL Injection

L'injection réflexive permet de charger une DLL directement en mémoire sans passer par les APIs Windows standard : Pour approfondir, consultez Anti-Forensics.

```
# Détection de DLL réflexives
def detect_reflective_dll(proc):
    # Scan des régions mémoire exécutables
    for vad in proc.VadRoot.traverse():
        if not vad.is_executable():
            continue

        # Recherche de patterns PE dans la mémoire
        memory_data = read_vad_content(context, proc, vad)

        # Signature MZ/PE sans mapping légitime
        if memory_data[:2] == b'MZ':
            pe_offset = struct.unpack('
```

3.2 Détection des Hooks Système

Les hooks permettent aux malwares d'intercepter et modifier le comportement du système. Volatility3 offre plusieurs plugins pour leur détection :

SSDT Hooking
La System Service Dispatch Table (SSDT) est une table critique contenant les adresses des services système :

```
# Analyse SSDT pour détecter les hooks
from volatility3.plugins.windows import ssdt

def analyze_ssdt_hooks(context):
    # Récupération de la SSDT
    ssdt_entries = ssdt.SSDT.get_ssdt(context, layer_name, symbol_table)

    for index, entry in enumerate(ssdt_entries):
        function_address = entry.Address
        module = get_module_for_address(context, function_address)

        # Vérification si l'adresse pointe vers ntoskrnl
        if not module or module.BaseDllName != "ntoskrnl.exe":
            print(f"SSDT Hook detected: Index {index} ->
0x{function_address:016x}")

            # Analyse du code au point de hook
            hook_code = read_memory(context, function_address, 32)
            disasm = disassemble(hook_code, function_address)
            print(f"Hook code: {disasm}")
```

IDT Hooking
L'Interrupt Descriptor Table peut être modifiée pour intercepter les interruptions :

```
# Détection de hooks IDT
def detect_idt_hooks(context):
    # Lecture de l'IDT via IDTR
    idtr = get_idtr(context)
    idt_base = idtr.base

    for vector in range(256):
        idt_entry = read_idt_entry(context, idt_base, vector)
        handler_address = idt_entry.offset

        # Vérification du module contenant le handler
        module = get_module_for_address(context, handler_address)

        if not is_legitimate_module(module):
            print(f"IDT Hook: Vector {vector:02x} -> 0x{handler_address:016x}")

            # Extraction du handler pour analyse
            handler_code = read_memory(context, handler_address, 256)
            analyze_handler(handler_code, vector)
```

Inline Hooking Detection

Les hooks inline modifient directement le code des fonctions :

```
# Détection de hooks inline dans les APIs critiques
def detect_inline_hooks(proc):
    critical_dlls = ['ntdll.dll', 'kernel32.dll', 'kernelbase.dll',
'user32.dll']

    for dll_name in critical_dlls:
        dll_base = get_dll_base(proc, dll_name)
        if not dll_base:
            continue

        # Parse exports
        exports = parse_exports(context, proc, dll_base)

        for export_name, export_rva in exports.items():
            function_address = dll_base + export_rva

            # Lecture des premiers bytes de la fonction
            function_bytes = read_process_memory(context, proc,
function_address, 16)

            # Détection de patterns de hook courants
            if function_bytes[0] == 0xE9:  # JMP relatif
                jmp_target = struct.unpack('
```

3.3 Analyse des Communications Backdoor

Les backdoors maintiennent souvent des canaux de communication avec leurs serveurs de commande et contrôle. L'analyse réseau en mémoire permet d'identifier ces connexions :

```
# Analyse des connexions réseau actives
from volatility3.plugins.windows import netscan

def analyze_network_connections(context):
    # Scan des structures réseau
    for net_obj in netscan.NetScan.scan(context, layer_name, symbol_table):
        if isinstance(net_obj, netscan.TcpConnection):
            local_addr = net_obj.LocalAddress
            remote_addr = net_obj.RemoteAddress
            state = net_obj.State
            pid = net_obj.Owner.UniqueProcessId if net_obj.Owner else 0

            # Détection de patterns suspects
            if is_suspicious_port(net_obj.RemotePort):
                print(f"Suspicious connection: {local_addr}:{net_obj.LocalPort}
-> "
                      f"{remote_addr}:{net_obj.RemotePort} (PID: {pid})")

            # Vérification de la légitimité du processus
            if pid and not is_legitimate_network_process(pid):
                proc = get_process_by_pid(context, pid)
                print(f"Unexpected network activity from {proc.ImageFileName}
(PID: {pid})")

            # Analyse du contenu des buffers réseau
            analyze_socket_buffers(context, net_obj)
```

```
# Détection de mécanismes de persistance
def detect_persistence_mechanisms(context):
    # 1. Analyse des services Windows
    services = get_services(context)
    for service in services:
        # Vérification du binaire du service
        if service.Binary:
            binary_path = service.Binary.dereference()
            if is_suspicious_path(binary_path):
                print(f"Suspicious service: {service.Name} -> {binary_path}")

            # Vérification de services avec DLL
            if "svchost.exe" in binary_path.lower():
                dll_path = get_service_dll(service)
                if dll_path and not is_signed_dll(dll_path):
                    print(f"Unsigned service DLL: {dll_path}")

    # 2. Analyse des tâches planifiées en mémoire
    scheduled_tasks = extract_scheduled_tasks(context)
    for task in scheduled_tasks:
        if task.Action and is_suspicious_command(task.Action):
            print(f"Suspicious scheduled task: {task.Name}")
            print(f"  Action: {task.Action}")
            print(f"  Trigger: {task.Trigger}")

    # 3. Détection de modifications WMI
    wmi_consumers = scan_wmi_persistence(context)
    for consumer in wmi_consumers:
        if consumer.Type == "CommandLineEventConsumer":
            print(f"WMI persistence detected: {consumer.Name}")
            print(f"  Command: {consumer.CommandLine}")
```

4.3 Analyse Comportementale et Heuristiques

L'analyse comportementale permet d'identifier des patterns d'activité malveillante même pour des malwares inconnus :

```
# Analyse comportementale avancée
class BehavioralAnalyzer:
    def __init__(self, context):
        self.context = context
        self.suspicious_behaviors = []

    def analyze_process_behavior(self, proc):
        score = 0
        indicators = []

        # 1. Analyse de l'arbre de processus
        parent = self.get_parent_process(proc)
        if parent and self.is_suspicious_parent_child(parent, proc):
            score += 30
            indicators.append(f"Suspicious parent-child: {parent.ImageFileName}
-> {proc.ImageFileName}")

        # 2. Analyse des allocations mémoire
        rwx_count = 0
        large_alloc_count = 0

        for vad in proc.VadRoot.traverse():
            if vad.is_readable() and vad.is_writable() and vad.is_executable():
                rwx_count += 1

            size = (vad.End - vad.Start) >> 12  # Pages
            if size > 1000:  # Plus de 4MB
                large_alloc_count += 1

        if rwx_count > 5:
            score += 20
            indicators.append(f"Multiple RWX regions: {rwx_count}")

        # 3. Analyse des handles
        handle_stats = self.analyze_handles(proc)
        if handle_stats['process_handles'] > 10:
            score += 15
            indicators.append(f"Excessive process handles:
{handle_stats['process_handles']}")

        # 4. Analyse temporelle
        if self.detect_time_anomalies(proc):
            score += 25
            indicators.append("Temporal anomalies detected")

        # 5. Analyse de l'entropie du code
        code_entropy = self.calculate_code_entropy(proc)
        if code_entropy > 6.5:
            score += 20
            indicators.append(f"High code entropy: {code_entropy:.2f}")

        if score >= 50:
            print(f"Suspicious behavior detected in PID {proc.UniqueProcessId}
(Score: {score})")
            for indicator in indicators:
                print(f"  - {indicator}")

        return score, indicators
```

6.1 Techniques d'Optimisation pour l'Analyse de Dumps Volumineux

L'analyse de dumps mémoire de plusieurs dizaines de gigaoctets nécessite des optimisations spécifiques :

```
# Optimisation avec traitement parallèle
import multiprocessing
from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor

class OptimizedAnalyzer:
    def __init__(self, memory_dump, workers=None):
        self.memory_dump = memory_dump
        self.workers = workers or multiprocessing.cpu_count()

    def parallel_vad_scan(self, context):
        """Scan parallèle des VAD pour recherche de patterns"""
        all_vads = []

        # Collecte de tous les VADs
        for proc in self.get_processes(context):
            for vad in proc.VadRoot.traverse():
                all_vads.append((proc.UniqueProcessId, vad))

        # Division en chunks pour traitement parallèle
        chunk_size = len(all_vads) // self.workers
        chunks = [all_vads[i:i+chunk_size] for i in range(0, len(all_vads),
chunk_size)]

        results = []
        with ProcessPoolExecutor(max_workers=self.workers) as executor:
            futures = []
            for chunk in chunks:
                future = executor.submit(self.scan_vad_chunk, context, chunk)
                futures.append(future)

            for future in futures:
                results.extend(future.result())

        return results

    def scan_vad_chunk(self, context, vad_chunk):
        """Scan d'un chunk de VADs"""
        findings = []

        for pid, vad in vad_chunk:
            try:
                # Lecture du contenu VAD
                data = self.read_vad_content(context, pid, vad)

                # Recherche de patterns
                if self.contains_shellcode_pattern(data):
                    findings.append({
                        'pid': pid,
                        'vad_start': vad.Start,
                        'type': 'shellcode',
                        'confidence': self.calculate_shellcode_confidence(data)
                    })

                # Détection d'autres artefacts
                if self.contains_pe_header(data):
                    findings.append({
                        'pid': pid,
                        'vad_start': vad.Start,
                        'type': 'unmapped_pe',
                        'pe_info': self.extract_pe_info(data)
                    })
```

```
            except Exception as e:
                continue

        return findings

    def optimized_string_search(self, context, patterns):
        """Recherche optimisée de chaînes avec index"""
        # Création d'un index Aho-Corasick pour recherche multi-patterns
        import pyahocorasick

        automaton = pyahocorasick.Automaton()
        for idx, pattern in enumerate(patterns):
            automaton.add_word(pattern, (idx, pattern))
        automaton.make_automaton()

        findings = []

        # Scan avec buffer rotatif pour économiser la mémoire
        BUFFER_SIZE = 100 * 1024 * 1024  # 100MB

        with open(self.memory_dump, 'rb') as f:
            offset = 0
            overlap = max(len(p) for p in patterns)  # Pour gérer les patterns
à cheval

            while True:
                buffer = f.read(BUFFER_SIZE)
                if not buffer:
                    break

                # Recherche dans le buffer
                for end_index, (pattern_id, pattern) in automaton.iter(buffer):
                    start_index = end_index - len(pattern) + 1
                    findings.append({
                        'offset': offset + start_index,
                        'pattern': pattern,
                        'context': buffer[max(0,
start_index-50):min(len(buffer), end_index+50)]
                    })

                # Gestion de l'overlap
                if len(buffer) == BUFFER_SIZE:
                    f.seek(-overlap, 1)
                    offset += BUFFER_SIZE - overlap
                else:
                    break

        return findings
```

6.2 Caching et Mémorisation des Résultats

Pour éviter les recalculs coûteux lors d'analyses itératives :

Analyse complementaire

```
# Système de cache pour analyses répétées
import pickle
import sqlite3
from functools import lru_cache

class CachedAnalyzer:
    def __init__(self, cache_db="analysis_cache.db"):
        self.cache_db = cache_db
        self.init_cache_db()

    def init_cache_db(self):
        """Initialise la base de données de cache"""
        conn = sqlite3.connect(self.cache_db)
        cursor = conn.cursor()

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS analysis_cache (
                key TEXT PRIMARY KEY,
                result BLOB,
                timestamp DATETIME,
                dump_hash TEXT
            )
        ''')
        conn.commit()
        conn.close()

    @lru_cache(maxsize=1000)
    def cached_process_analysis(self, proc_key):
        """Analyse de processus avec cache LRU"""
        # Vérification du cache persistant
        cached_result = self.get_from_cache(proc_key)
        if cached_result:
            return cached_result

        # Analyse réelle si pas en cache
        result = self.analyze_process_internal(proc_key)

        # Sauvegarde en cache
        self.save_to_cache(proc_key, result)

        return result

    def get_from_cache(self, key):
        """Récupération depuis le cache persistant"""
        conn = sqlite3.connect(self.cache_db)
        cursor = conn.cursor()

        cursor.execute('''
            SELECT result FROM analysis_cache
            WHERE key = ? AND dump_hash = ?
        ''', (key, self.current_dump_hash))

        row = cursor.fetchone()
        conn.close()

        if row:
            return pickle.loads(row[0])
        return None
```

Questions frequentes

Comment mener une investigation forensique sur un systeme compromis ?

Une investigation forensique debute par la preservation des preuves via une image disque et un dump memoire, suivie de l'analyse des artefacts systeme (registres, journaux d'evenements, fichiers prefetch), la reconstruction de la timeline d'activite et la correlation des indicateurs de compromission pour identifier la source et l'etendue de l'attaque.

Quels sont les outils essentiels pour l'analyse forensique ?

Les outils essentiels pour l'analyse forensique incluent Volatility pour l'analyse memoire, Autopsy et FTK pour l'analyse disque, KAPE et Velociraptor pour la collecte automatisee, Plaso pour la creation de timelines, ainsi que des outils de triage comme Eric Zimmerman's tools pour l'analyse des artefacts Windows.

Pourquoi la chaine de custody est-elle importante en forensique ?

La chaine de custody garantit l'integrite et l'admissibilite des preuves numeriques en documentant chaque etape de manipulation, de la collecte a la presentation. Sans une chaine de custody rigoureuse, les preuves peuvent etre contestees juridiquement et perdre leur valeur probante.

Pour approfondir, consultez les ressources officielles : SANS White Papers, NVD - NIST et ANSSI.

Sources et références : SANS SIFT · MITRE ATT&CK

Articles connexes

- Email Forensics : Tracer les Campagnes Phishing en 2026
- Forensique Mémoire : Guide Pratique Volatility 3 en 2026
- MacOS Forensics : Artifacts et Persistence : Guide Complet

Conclusion et Perspectives

L'analyse forensique de la mémoire Windows avec Volatility3 et WinPMEM constitue un domaine en constante évolution, où la sophistication croissante des menaces nécessite une adaptation permanente des techniques d'investigation. Les méthodologies présentées dans cet article offrent une base solide pour la détection des backdoors et implants les plus aboutis, mais l'investigateur doit rester vigilant face aux nouvelles techniques d'évasion.

Les développements futurs dans ce domaine incluront probablement l'intégration de l'intelligence artificielle pour la détection comportementale, l'amélioration des techniques d'acquisition sur les systèmes avec protections matérielles avancées (Intel CET, ARM Pointer Authentication), et le développement de méthodes d'analyse pour les environnements cloud et conteneurisés.

La maîtrise de ces outils et techniques est devenue indispensable pour tout professionnel de la sécurité informatique confronté aux menaces modernes. L'investissement dans la formation continue et la pratique régulière sur des cas réels permettra de maintenir un niveau d'expertise adapté aux défis actuels et futurs de la cybersécurité.

Le registre Windows, dans sa complexité et sa richesse, continue d'offrir une fenêtre incomparable sur les activités système et utilisateur. Sa maîtrise représente non seulement une compétence technique essentielle, mais aussi un art nécessitant expérience, intuition, et rigueur méthodologique. Pour l'investigateur déterminé, il reste une source inépuisable de vérité numérique, révélant les secrets les plus profonds des systèmes Windows et les actions de ceux qui les utilisent.

Ressources open source associées :

- awesome-cybersecurity-tools — Liste de 100+ outils de cybersécurité

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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