Asynchronous Procedure Calls (APC)
Injection Detection Plugin Suite

Kartik N. Iyer and Parag H. Rughani

Volatility Plugin Contest 2025
APCWatch & MalAPC Plugin Suite

December 25, 2025

Abstract

This documentation presents a comprehensive plugin suite for detecting Asynchronous Procedure Call (APC) injection attacks in Windows memory forensics. The suite consists of two complementary Volatility 3 plugins: **APCWatch** for APC enumeration and baseline analysis, and **MalAPC** for advanced threat detection and payload analysis. Together, these tools provide investigators with the capability to identify and analyze one of the most sophisticated code injection techniques employed by modern malware. This documentation covers the theoretical foundations, implementation details, practical usage, and forensic analysis workflows for both plugins.

Contents

1 Introduction

Memory forensics has become an essential part of modern digital investigations, allowing analysts to look beyond the files stored on disk and examine what was actually happening in a system's memory at a specific moment. By studying a memory dump, investigators can uncover evidence that usually disappears once a computer is turned off, such as running processes, loaded modules, in-memory code, network sessions, and encryption keys. This provides a real-time view of system activity, helping uncover hidden malware, injected code, and other suspicious behavior that traditional forensics might miss.

Over the years, tools and methodologies for memory analysis have evolved significantly, and one framework that continues to lead the field is the **Volatility Framework**, developed by the Volatility Foundation. Its modern version, Volatility 3, was redesigned to support newer operating systems and offer a cleaner, more modular architecture. It also enables researchers to build their own plugins and extend its capabilities, making it a powerful platform for exploring complex forensic artifacts and investigating advanced attack techniques.

Each year, the Volatility Foundation organizes a Plugin Contest that encourages researchers to develop creative solutions to real-world forensic challenges. These contests have become a driving force behind innovation in memory forensics, allowing the community to share tools, ideas, and discoveries that push the boundaries of what can be achieved through memory analysis.

This year, our contribution focuses on detecting **APC (Asynchronous Procedure Call) injection**, a stealthy code injection technique often used by attackers to execute malicious payloads inside legitimate processes. APC injection works by queuing a custom callback function to a thread within another process, causing that thread to unknowingly execute the injected code. To detect and analyze such activity, we developed two Volatility 3 plugins: **APCWatch** and **MalAPC**.

APCWatch inspects thread-related kernel structures and identifies processes with active APC queues, providing detailed information about queued callbacks and their memory locations. **MalAPC** builds upon this analysis to detect processes that exhibit signs of APC injection, such as callbacks pointing to suspicious memory regions or executable pages modified during runtime. Together, these plugins enhance an investigator's ability to identify and understand one of the more covert injection methods seen in modern malware.

2 Introducing the Asynchronous Procedure Calls Injection (APC) Plugin Suite

2.1 Abstract

APC injection is a subtle yet powerful code injection technique often exploited by malware to execute malicious routines within legitimate processes. To address this, this section presents two specialized Volatility 3 plugins — **APCWatch** and **MalAPC** — developed to detect and analyze APC-related activities within memory samples. APCWatch focuses on enumerating threads and extracting APC queue elements, while MalAPC correlates these findings with memory protection and mapping details to identify suspicious callbacks pointing to executable or non-module regions. Together, they provide a precise and reliable method for uncovering stealthy APC-based injections, enhancing the overall depth and accuracy of memory forensic investigations.

During my academic work in malware analysis and memory forensics, I became interested in Asynchronous Procedure Calls (APCs) after a conversation with my professor. We dug into how APCs let code be scheduled to run in the context of another thread, and it quickly became clear this mechanism—useful for legitimate synchronization and notification tasks—can also be repurposed by attackers to execute code stealthily. That discussion led us to focus our project on APCs and to develop practical tools that make their traces easier to find in memory.

APCs are a kernel-provided way to queue a callback to a specific thread so that the callback executes when that thread enters an alertable state. This behavior is legitimate and used by applications for callbacks and async notifications, but it also means an attacker can arrange for code to run inside a target process without creating new remote threads or leaving obvious artifacts on disk. Because APCs execute within the target process's address space and privileges, they can be an attractive technique for stealthy post-exploitation activity.

Recognizing this dual nature, we built two Volatility 3 plugins: **APCWatch** to enumerate and expose queued APCs across threads and processes, and **MalAPC** to apply practical checks—based on memory mappings and protection attributes—so investigators can prioritize callbacks that look out of place. Our goal was not to claim perfect detection, but to provide concise, evidence-driven outputs that shorten the path from suspicion to investigation. By focusing on APCs in a targeted, transparent way, we aim to make a modest but useful contribution to routine forensic workflows. The plugins are intended to help analysts spot APC-related anomalies quickly and to provide the contextual details needed for sensible follow-up, rather than to replace deeper manual analysis steps.

2.2 Overview of APC Callbacks

Asynchronous Procedure Calls, or APCs, are a kernel mechanism that lets code be scheduled to run in the context of a specific thread. They are normally used for legitimate tasks such as deferred work, notifications, and certain inter-thread callbacks. Because APCs execute inside the target thread's address space and with its privileges, they are a convenient tool for legitimate async operations that need to run as part of an existing thread.

That same behavior is what makes APCs attractive to attackers. By queuing a callback into another process's thread, an adversary can run code inside that process without creating a new remote thread or leaving obvious on-disk traces. This technique can be used for stealthy code execution and process manipulation, and it often leaves subtle indicators in thread APC queues and memory mappings rather than in obvious artifacts.

2.2.1 APC Types and Execution Context

Windows implements two primary types of APCs:

- **Kernel-Mode APCs:** Execute with kernel privileges and are typically used for system-level operations such as I/O completion and deferred procedure calls.
- **User-Mode APCs:** Execute in user context and require the target thread to be in an alertable wait state. These are the primary target for malicious exploitation.

APC injection is therefore an important area for memory forensics. Instead of relying on broad, noisy heuristics, focused inspection of thread lists, APC queue entries, callback addresses, and associated VAD protections can reveal where callbacks point and whether those targets look legitimate. Our tools are built to provide that targeted visibility so analysts can quickly see which callbacks deserve closer scrutiny and move from suspicion to evidence-based investigation.

2.3 Purpose and Objective of the Plugin

The APCWatch and MalAPC plugins were developed to address a gap in existing memory-forensic tooling. While Volatility's `malfind` plugin is effective at detecting indicators of code injection, it does not differentiate between injection techniques or reveal the specific mechanisms malware may have used. Since modern injection methods vary significantly in how they manipulate memory, analysts often need more targeted insight to correctly interpret unusual findings. APC-based injection, in particular, has become a subtle technique that is not highlighted by general-purpose detection mechanisms, creating the need for a more focused analytical approach.

To fill this niche, APCWatch and MalAPC provide dedicated visibility into Asynchronous Procedure Calls (APCs) within Windows memory snapshots. APCWatch systematically enumerates queued APCs across threads and processes, giving analysts a clear view of legitimate and potentially suspicious APC activity. MalAPC builds on this by examining the memory regions associated with APC routines, flagging those that reside in RWX or unbacked executable areas, and offering optional hexdumps and disassembly for deeper investigation. Together, these plugins help distinguish benign APC behavior from indicators of APC-based injection, enhancing the precision and context of memory-forensic analysis.

2.3.1 Plugin Arguments

Both plugins provide configurable command-line arguments to customize their behavior:

APCWatch Arguments:

- `-pid`: Filter analysis to specific process IDs

- `-show-kernel-apcs`: Include kernel-mode APCs in output
- `-verbose`: Display additional debugging information

MalAPC Arguments:

- `-pid`: Filter on specific process IDs
- `-dump-payloads`: Dump suspicious memory regions
- `-dump-size`: Size of memory to dump in bytes (default: 512)
- `-use-whitelist`: Skip analysis for known legitimate processes (default: True)
- `-min-threat-score`: Minimum threat score to report (1-15, default: 5)
- `-enable-jit-detection`: Enable JIT compiler detection (default: True)

2.4 Core Features and Working of the Plugin

Our APC analysis suite, composed of the complementary APCWatch and MalAPC plugins, is purpose-built for the Windows environment, providing deep insight into the execution mechanics of Asynchronous Procedure Calls. Both plugins are fully integrated and compatible with Volatility 3, serving different, but equally critical, stages of a memory investigation. This section walks through the primary functions and mechanisms of these tools.

2.4.1 APC Enumeration and Baseline Analysis (APCWatch)

We start with APCWatch, which is designed to provide the forensic investigator with a comprehensive, thread-level map of all queued APCs. Before you can hunt for malicious activity, you need to understand the normal baseline. The plugin achieves this by traversing all active processes and their threads. Specifically, it digs into the `Tcb` (Thread Control Block) of each `_ETHREAD` structure to locate the `ApcState` (`_KAPC_STATE`).

From there, we can follow the `ApcListHead` to harvest every queued `_KAPC` object. Once an object is found, we extract the critical components: the memory addresses of the `KernelRoutine` and the `NormalRoutine`, which are the key pointers that malware loves to hijack. We also pull important context flags, like the APC Mode (Kernel or User), and the various Status Flags (e.g., Inserted, UserAPCPending), which tell us about the APC's current execution status.

Safely retrieving these pointer offsets is crucial to avoid memory reading errors:

```python
# Snippet from apcwatch.py's _apc_generator method
# Safely resolving the routine addresses
if hasattr(apc, 'KernelRoutine') and apc.KernelRoutine:
    kernel_routine = apc.KernelRoutine.vol.offset
if hasattr(apc, 'NormalRoutine') and apc.NormalRoutine:
    normal_routine = apc.NormalRoutine.vol.offset
```

Listing 1: APC Routine Extraction from APCWatch

2.4.2 Suspicious Injection Detection and Payload Analysis (MalAPC)

MalAPC employs a multi-layered threat detection approach that combines individual routine analysis with behavioral pattern recognition. Rather than simply flagging anomalies, the plugin implements a comprehensive scoring system that evaluates multiple indicators of compromise to accurately assess threat severity. The plugin's detection capabilities are built around four fundamental analysis components, each implemented within the `_analyze_routine` method to systematically evaluate APC routine addresses.

Memory Protection Analysis The most reliable indicator of injected shellcode is the presence of memory regions with execute, read, and write permissions simultaneously. MalAPC examines the Virtual Address Descriptor (VAD) tree to identify regions marked with suspicious protection flags, particularly `PAGE_EXECUTE_READWRITE` and `PAGE_EXECUTE_WRITECOPY`. These protection combinations are extraordinarily rare in legitimate applications but ubiquitous in shellcode injection scenarios, where attackers need to write their payload into memory and subsequently execute it. The detection of such regions immediately triggers a high-confidence alert.

```python
# From the _analyze_routine method in malapc.py
if vad_info['protection'] in self.SUSPICIOUS_PROTECTIONS:
    analysis['reasons'].append(f"RWX memory ({vad_info['protection']})")
    analysis['score'] += 6
    analysis['suspicious'] = True
```

Listing 2: Memory Protection Analysis in MalAPC

Module Boundary Verification Legitimate code execution in Windows processes occurs within well-defined module boundaries—namely, the executable image and its loaded Dynamic Link Libraries (DLLs). MalAPC maintains a detailed map of all loaded modules by parsing the Process Environment Block (PEB) and its Loader Data structures.

```python
# From the _get_loaded_modules_detailed method
modules[base_addr] = {
    'name': name,
    'base': base_addr,
    'end': base_addr + size,
    'size': size
}
```

Listing 3: Module Mapping Construction

When an APC routine address falls outside these established boundaries, it indicates that code execution is being directed to an unmapped region—a clear sign of injection. The plugin reports this condition with explicit attribution:

```python
if not module_info:
    analysis['reasons'].append("Not in loaded module")
    analysis['score'] += 4
    analysis['module'] = "NO_MODULE"
```

Listing 4: Module Boundary Violation Detection

Private Executable Memory Detection Beyond the obvious RWX regions, attackers often employ more subtle techniques by allocating private memory with execute permissions. These allocations don't correspond to any file-backed mapping and exist solely in the process's private address space. MalAPC identifies these suspicious allocations by examining VAD type characteristics:

```python
# From the _analyze_routine method
if 'Private' in vad_info['type'] and not module_info:
    if 'EXECUTE' in vad_info['protection']:
        if vad_info['size'] < 8192:  # Less than 2 pages
            analysis['reasons'].append("Small private executable allocation (shellcode pattern)")
            analysis['score'] += 5
        else:
            analysis['reasons'].append("Private executable allocation")
            analysis['score'] += 3
        analysis['suspicious'] = True
```

Listing 5: Private Executable Memory Detection

Shellcode Pattern Recognition To increase detection confidence, MalAPC includes heuristic analysis capabilities that scan suspicious memory regions for common shellcode artifacts. The `_contains_shellcode_patterns` method identifies characteristic instruction sequences frequently employed by exploit developers:

```
# PATTERN 1: NOP sled (6+ consecutive NOPs)
if b"\x90" * 6 in data:
    nop_count = data.count(b"\x90" * 6)
    detections.append(f"NOP sled detected ({nop_count}
       sequences)")
    score += 3

# PATTERN 2: GetPC stub (CALL $+5 pattern)
if b"\xe8\x00\x00\x00\x00" in data:
    detections.append("GetPC stub (position-independent code)")
    score += 4

# PATTERN 3: PEB/TEB access patterns
peb_patterns = [
    b"\x64\xa1\x30\x00\x00\x00",  # MOV EAX, FS:[0x30]
    b"\x65\x48\x8b\x04\x25",      # MOV RAX, GS:[0x??]
]
for pattern in peb_patterns:
    if pattern in data:
        detections.append("PEB/TEB direct access (shellcode
           pattern)")
        score += 5
        break
```

Listing 6: Shellcode Pattern Detection

These patterns represent standard shellcode techniques such as NOP sleds for alignment, position-independent code initialization, and direct access to process information structures. Their presence provides additional corroborating evidence of malicious activity.

Behavioral Pattern Analysis Beyond analyzing individual routines, MalAPC implements higher-level behavioral analysis through the `_detect_malicious_patterns` method. This approach recognizes that certain APC configurations, while individually benign, indicate injection when combined:

- **User-Mode APC Injection Pattern:** A user-mode APC with only a `NormalRoutine` and no `KernelRoutine` represents the classic `QueueUserAPC` injection technique, where attackers queue execution directly to shellcode without kernel-mode participation.

- **Kernel APC Anomalies:** While kernel APCs legitimately require both routines in certain scenarios, MalAPC incorporates kernel address space awareness to avoid false positives from legitimate system operations:

```python
def _is_kernel_address(self, addr: int) -> bool:
    """Check if address is in kernel space (supports both 32-bit and 64-bit)"""
    # For 64-bit systems: kernel space starts at 0xFFFF800000000000
    if addr >= 0xF80000000000:
        return True

    # For 32-bit systems: kernel space typically starts at 0x80000000
    if 0x80000000 <= addr <= 0xFFFFFFFF:
        return True

    return False
```

Listing 7: Kernel Address Space Detection

Threat Scoring and Classification MalAPC aggregates evidence from multiple detection mechanisms into a unified threat score, enabling analysts to prioritize their investigation efforts effectively. The scoring system assigns weights based on indicator reliability:

Table 1: Threat Score Weight Assignment

Detection Indicator	**Score Weight**
Kernel APC to User Space	+9 points
RWX Memory Detection	+6 points
Small Private Executable	+5 points
PEB/TEB Access Pattern	+5 points
Not in Loaded Module	+4 points
GetPC Stub Detection	+4 points
Private Executable Memory	+3 points
NOP Sled Detection	+3 points
High Entropy Small Region	+3 points
Behavioral Pattern Match	+2 points
Baseline Deviation	+2 points
Thread Context Anomaly	+2 points

These scores combine to produce a threat classification:

- **CRITICAL (10+ points):** Multiple strong indicators suggest active code injection
- **HIGH (8-9 points):** Significant evidence of suspicious activity
- **MEDIUM (5-7 points):** Noteworthy anomalies requiring review
- **LOW (1-4 points):** Minor deviations from normal behavior

Enhanced Detection: JIT Compiler Recognition One of the key features of MalAPC is the ability to distinguish between legitimate Just-In-Time (JIT) compiled code and malicious RWX allocations. Modern applications such as web browsers (.NET, JavaScript engines) and development environments routinely create RWX memory for dynamic compilation, which could otherwise lead to numerous false positives.

The `_is_legitimate_jit_allocation` method implements a multi-factor analysis:

```python
def _is_legitimate_jit_allocation(
    self,
    proc,
    vad_info: dict,
    loaded_modules: dict,
    process_name: str,
    proc_layer
) -> bool:
    """Detect legitimate JIT compiler memory allocations"""

    # Check if process is JIT-enabled
    if process_name.lower() not in self.JIT_PROCESSES:
        return False

    # Check if JIT modules are loaded
    has_jit_module = False
    for module in loaded_modules.values():
        if module['name'].lower() in self.JIT_MODULES:
            has_jit_module = True
            break

    if not has_jit_module:
        return False

    # JIT allocations are typically larger (> 4KB)
    if vad_info.get('size', 0) < 4096:
        return False

    # Check entropy variance - JIT code has stable high entropy
    try:
        region_start = vad_info['start']
        sample_data = proc_layer.read(region_start, min(
            vad_info.get('size', 256), 2048), pad=True)

        if len(sample_data) >= 256:
            entropy_variance = self._calculate_entropy_variance
                (sample_data)

            # JIT code has stable high entropy (low variance)
            # Shellcode often has variable entropy (high
                variance)
            if entropy_variance < 0.3:  # Low variance = likely
                 JIT
                return True
    except:
        pass

    return False
```

Listing 8: JIT Compiler Detection Logic

This approach examines:

1. Process name against known JIT-capable applications
2. Loaded modules for JIT compiler DLLs (clr.dll, coreclr.dll, v8.dll, etc.)
3. Allocation size (JIT regions are typically larger than shellcode)
4. Entropy variance (JIT code has consistent high entropy, shellcode varies)

Entropy Analysis for Shellcode Detection MalAPC employs Shannon entropy calculation to distinguish between compressed/encrypted shellcode and normal code:

```
def _calculate_entropy(self, data: bytes) -> float:
    """Calculate Shannon entropy of data"""
    if not data or len(data) == 0:
        return 0.0

    entropy = 0.0
    for i in range(256):
        count = data.count(bytes([i]))
        if count > 0:
            p_x = count / len(data)
            entropy += -p_x * math.log2(p_x)

    return entropy
```

Listing 9: Shannon Entropy Calculation

High entropy (> 7.3) in small memory regions often indicates encrypted or packed shellcode, triggering additional scrutiny.

2.4.3 Payload Extraction and Analysis

When MalAPC identifies a suspicious APC routine, it doesn't merely flag the alert—it actively retrieves the injected payload for forensic analysis. The plugin extracts up to 64 bytes by default (configurable via the `-dump-size` parameter) from the suspicious memory region:

```
dump_size = self.config.get('dump-size', 512)
actual_dump_size = min(dump_size, 2048)  # Cap at 2KB
data = proc_layer.read(routine_addr, actual_dump_size, pad=True
    )
analysis['hexdump'] = data[:actual_dump_size]
```

Listing 10: Payload Extraction

The extracted payload is presented in the output as a hexdump, allowing analysts to:

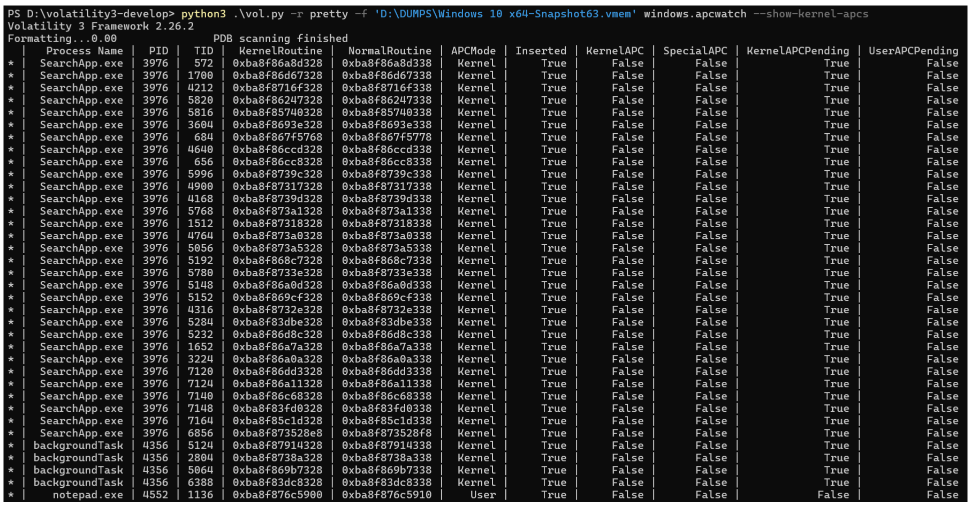
- Identify shellcode signatures
- Extract embedded strings or IP addresses
- Perform offline disassembly with tools like IDA Pro or Ghidra
- Submit samples to antivirus engines for identification

2.5 Output Format

The output of the APCWatch and MalAPC plugins are designed to be straightforward and user-friendly, providing critical information about processes and their associated APC callbacks in memory. This section explains each column and its purpose.

2.5.1 APCWatch Output Format

The output of the APCWatch plugin is designed to provide a granular, thread-level view of the system's asynchronous activity. By breaking down the specific state and destination of every queued job, analysts can establish a clear baseline of normal execution. The output columns are defined as follows:



```
PS D:\volatility3-develop> python3 .\vol.py -r pretty -f 'D:\DUMPS\Windows 10 x64-Snapshot63.vmem' windows.apcwatch --show-kernel-apcs
Volatility 3 Framework 2.26.2
Formatting...0.00               PDB scanning finished
  |   Process Name |  PID |  TID | KernelRoutine  |  NormalRoutine | APCMode | Inserted | KernelAPC | SpecialAPC | KernelAPCPending | UserAPCPending
* |  SearchApp.exe | 3976 |  572 | 0xba8f86a8d328 | 0xba8f86a8d338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 1700 | 0xba8f86d67328 | 0xba8f86d67338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4212 | 0xba8f8716f328 | 0xba8f8716f338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5820 | 0xba8f86247328 | 0xba8f86247338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5816 | 0xba8f85740328 | 0xba8f85740338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 3604 | 0xba8f8693e328 | 0xba8f8693e338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 |  684 | 0xba8f867f5768 | 0xba8f867f5778 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4640 | 0xba8f86ccd328 | 0xba8f86ccd338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 |  656 | 0xba8f86cc8328 | 0xba8f86cc8338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5996 | 0xba8f8739c328 | 0xba8f8739c338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4900 | 0xba8f87317328 | 0xba8f87317338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4168 | 0xba8f8739d328 | 0xba8f8739d338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5768 | 0xba8f873a1328 | 0xba8f873a1338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 1512 | 0xba8f87318328 | 0xba8f87318338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4764 | 0xba8f873a0328 | 0xba8f873a0338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5056 | 0xba8f873a5328 | 0xba8f873a5338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5192 | 0xba8f868c7328 | 0xba8f868c7338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5780 | 0xba8f8733e328 | 0xba8f8733e338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5148 | 0xba8f86a0d328 | 0xba8f86a0d338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5152 | 0xba8f869cf328 | 0xba8f869cf338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 4316 | 0xba8f8732e328 | 0xba8f8732e338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5284 | 0xba8f83dbe328 | 0xba8f83dbe338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 5232 | 0xba8f86d8c328 | 0xba8f86d8c338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 1652 | 0xba8f86a7a328 | 0xba8f86a7a338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 3224 | 0xba8f86a0a328 | 0xba8f86a0a338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 7120 | 0xba8f86dd3328 | 0xba8f86dd3338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 7124 | 0xba8f86a11328 | 0xba8f86a11338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 7140 | 0xba8f86c68328 | 0xba8f86c68338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 7148 | 0xba8f83fd0328 | 0xba8f83fd0338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 7164 | 0xba8f85c1d328 | 0xba8f85c1d338 |  Kernel |     True |     False |      False |             True |          False
* |  SearchApp.exe | 3976 | 6856 | 0xba8f873528e8 | 0xba8f873528f8 |  Kernel |     True |     False |      False |             True |          False
* | backgroundTask | 4356 | 5124 | 0xba8f87914328 | 0xba8f87914338 |  Kernel |     True |     False |      False |             True |          False
* | backgroundTask | 4356 | 2804 | 0xba8f8738a328 | 0xba8f8738a338 |  Kernel |     True |     False |      False |             True |          False
* | backgroundTask | 4356 | 5064 | 0xba8f869b7328 | 0xba8f869b7338 |  Kernel |     True |     False |      False |             True |          False
* | backgroundTask | 4356 | 6388 | 0xba8f83dc8328 | 0xba8f83dc8338 |  Kernel |     True |     False |      False |             True |          False
* |    notepad.exe | 4552 | 1136 | 0xba8f876c5900 | 0xba8f876c5910 |    User |     True |     False |      False |            False |          False
```

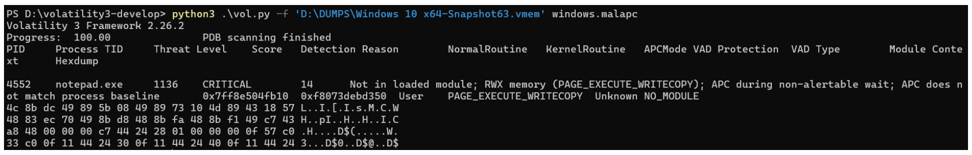
Figure 1: Output format for APCWatch

Table 2: APCWatch Output Columns

Column	Description
Process Name	Displays the executable name (e.g., `svchost.exe`) associated with the process where the APC is queued.
PID	The unique identifier for the process, allowing for cross-referencing with other plugins like `pslist`.
TID	The unique identifier for the specific thread that the APC is targeting. This is crucial because APCs are thread-specific, not just process-specific.
KernelRoutine	Shows the hexadecimal memory address of the kernel routine. This function executes in kernel mode and is responsible for freeing the APC object or performing kernel-level tasks.
NormalRoutine	Displays the hexadecimal memory address of the normal routine. This is the primary function intended to run in the target thread's context and is the most common target for manipulation by malware.
APCMode	Indicates whether the APC is scheduled to execute in Kernel mode or User mode. User-mode APCs are frequently abused for injection because they execute code within the user's application space.
Inserted	A boolean flag indicating whether the APC has been successfully inserted into the thread's APC queue.
KernelAPC	Boolean flag derived from the `InProgressFlags` of the `_KAPC_STATE`. Indicates if a kernel-mode APC is currently in progress.
SpecialAPC	Boolean flag indicating if a special kernel-mode APC is in progress.
KernelAPCPending	Reveals if there are pending kernel-mode APCs waiting to be delivered, helping analysts spot backed-up queues.
UserAPCPending	Indicates if there are pending user-mode APCs waiting to be delivered for the thread.

2.5.2 MalAPC Output Format

For MalAPC, the focus is on speed and threat detection. The columns are selected to highlight "red flags" immediately, so analysts don't have to dig through data to find proof of an attack.



```
PS D:\volatility3-develop> python3 .\vol.py -f 'D:\DUMPS\Windows 10 x64-Snapshot63.vmem' windows.malapc
Volatility 3 Framework 2.26.2
Progress:  100.00               PDB scanning finished
PID     Process TID     Threat Level    Score   Detection Reason        NormalRoutine   KernelRoutine   APCMode VAD Protection  VAD Type        Module Conte
xt      Hexdump

4552    notepad.exe     1136    CRITICAL        14      Not in loaded module; RWX memory (PAGE_EXECUTE_WRITECOPY); APC during non-alertable wait; APC does n
ot match process baseline       0x7ff8e504fb10  0xf8073debd350  User    PAGE_EXECUTE_WRITECOPY  Unknown NO_MODULE
4c 8b dc 49 89 5b 08 49 89 73 10 4d 89 43 18 57 L..I.[.I.s.M.C.W
48 83 ec 70 49 8b d8 48 8b fa 48 8b f1 49 c7 43 H..pI..H..H..I.C
a8 48 00 00 00 c7 44 24 28 01 00 00 00 0f 57 c0 .H....D$(.....W.
33 c0 0f 11 44 24 30 0f 11 44 24 40 0f 11 44 24 3...D$0..D$@..D$
```

Figure 2: Output format of MalAPC

Table 3: MalAPC Output Columns

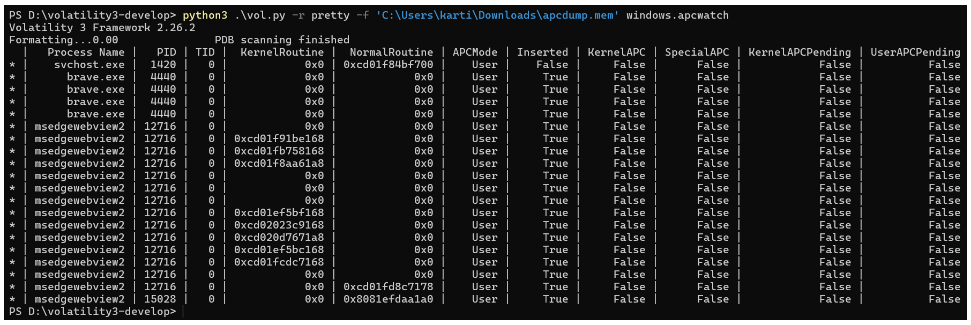
Column	Description
PID	The unique identifier of the potentially infected process.
Process	The executable name of the process under investigation.
TID	Thread identifier where the suspicious APC was detected.
Threat Level	Classification of the threat (CRITICAL, HIGH, MEDIUM, LOW) based on aggregated indicators.
Score	Numerical threat score representing the confidence level of detection (1-15+).
Detection Reason	Semicolon-separated list of specific indicators that triggered the alert (e.g., "Not in loaded module; RWX memory").
NormalRoutine	Hexadecimal address where the APC execution starts. In an attack, this usually points to the shellcode entry point.
KernelRoutine	Hexadecimal address of the associated kernel function.
APCMode	Indicates whether the attack is happening in User or Kernel space.
VAD Protection	Memory protection flags of the region (e.g., `PAGE_EXECUTE_READWRITE`). If you see "Execute" and "Write" together, it is almost always malicious.
VAD Type	Type of Virtual Address Descriptor (Private, Mapped, Image) indicating how the memory was allocated.
Module Context	Indicates which DLL or module contains the routine, or displays "NO_MODULE" if the address doesn't belong to any loaded module.
Hexdump	Raw bytes of the memory at the suspicious address, displayed in hexadecimal format. Useful for manual verification and pattern recognition.

3 Practical Tests

To demonstrate the real-world utility of our work, we tested the plugins against Windows 10 memory samples. These trials confirm the suite's ability to distinguish between legitimate background activity and malicious APC usage, specifically showcasing the automated detection and analysis of injected shellcode.

3.1 Analysis of Known Clean Memory Sample

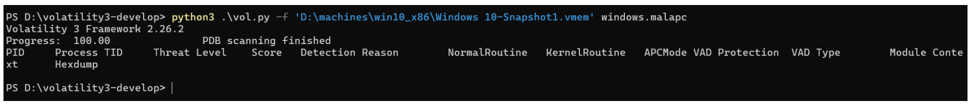
To establish a baseline for our analysis, we first ran the plugin against a clean Windows 10 memory snapshot. The resulting output, as shown in the provided terminal screenshot, specifically filters for processes with active or queued Asynchronous Procedure Calls (APCs). In this controlled, "clean" state, the results reflect the standard background rhythm of a healthy operating system.



```
PS D:\volatility3-develop> python3 .\vol.py -r pretty -f 'C:\Users\karti\Downloads\apcdump.mem' windows.apcwatch
Volatility 3 Framework 2.26.2
Formatting...0.00               PDB scanning finished
  |  Process Name |   PID | TID |  KernelRoutine |  NormalRoutine | APCMode | Inserted | KernelAPC | SpecialAPC | KernelAPCPending | UserAPCPending
* |   svchost.exe |  1420 |   0 |            0x0 | 0xcd01f84bf700 |    User |    False |     False |      False |            False |          False
* |     brave.exe |  4440 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* |     brave.exe |  4440 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* |     brave.exe |  4440 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* |     brave.exe |  4440 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01f91be168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01fb758168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01f8aa61a8 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01ef5bf168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd02023c9168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd020d7671a8 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01ef5bc168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 | 0xcd01fcdc7168 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 |            0x0 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 12716 |   0 |            0x0 | 0xcd01fd8c7178 |    User |     True |     False |      False |            False |          False
* | msedgewebview2 | 15028 |   0 |            0x0 | 0x8081efdaa1a0 |    User |     True |     False |      False |            False |          False
PS D:\volatility3-develop> |
```

Figure 3: APCWatch executed on a clean memory dump

To validate our threat detection logic, we subsequently executed MalAPC against the clean memory sample. As anticipated, the plugin returned no results. This empty output is significant; it indicates that despite the high volume of legitimate APC traffic observed earlier with APCWatch, the heuristic engine correctly recognized these operations as benign. This low false-positive rate is essential for maintaining analyst focus, ensuring that reported anomalies represent genuine deviations rather than standard system noise.



```
PS D:\volatility3-develop> python3 .\vol.py -f 'D:\machines\win10_x86\Windows 10-Snapshot1.vmem' windows.malapc
Volatility 3 Framework 2.26.2
Progress:  100.00               PDB scanning finished
PID     Process TID     Threat Level    Score   Detection Reason        NormalRoutine   KernelRoutine   APCMode VAD Protection  VAD Type        Module Conte
xt      Hexdump

PS D:\volatility3-develop> |
```

Figure 4: MalAPC executed on a clean memory dump

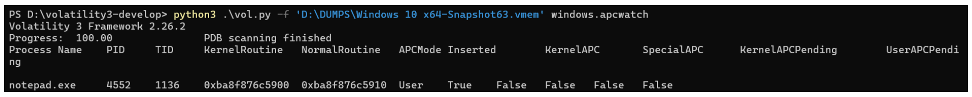
```
$ python3 vol.py -f "D:\machines\win10_x86\Windows 10-Snapshot1.vmem" windows.malapc

Volatility 3 Framework 2.26.2
Progress:  100.00               PDB scanning finished

(No threats detected)
```

3.2 Analysis of an Infected Memory Sample

To evaluate the detection mechanism against active injection techniques, we analyzed a test scenario targeting the standard Windows binary `notepad.exe`. The APCWatch plugin successfully identified a deviation in the process's execution flow, flagging a queued User mode APC attached to thread 1136. While `notepad.exe` is a trusted system application, it is frequently co-opted by adversaries as a host for malicious code to evade detection. Consequently, the distinct observation of an externally inserted APC within this context serves as a high-fidelity indicator of compromised process integrity.



```
PS D:\volatility3-develop> python3 .\vol.py -f 'D:\DUMPS\Windows 10 x64-Snapshot63.vmem' windows.apcwatch
Volatility 3 Framework 2.26.2
Progress:  100.00               PDB scanning finished
Process Name    PID     TID     KernelRoutine   NormalRoutine   APCMode Inserted        KernelAPC       SpecialAPC      KernelAPCPending        UserAPCPendi
ng

notepad.exe     4552    1136    0xba8f876c5900  0xba8f876c5910  User    True    False   False   False   False
```

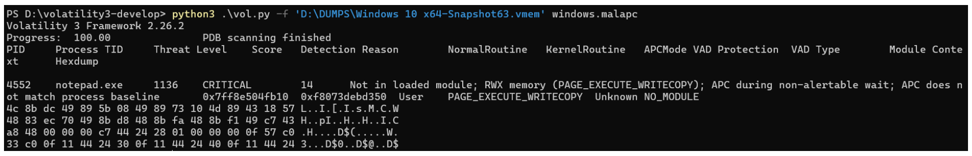
Figure 5: APCWatch executed on an infected memory dump

```
$ python3 vol.py -f infected.vmem windows.apcwatch

PID    Process        TID    KernelRoutine       NormalRoutine       APCMode   Inserted
----   -----------    ----   ----------------    ----------------    -------   ---------
4552   notepad.exe    1136   0xf8073debd350      0x7ff8e504fb10      User      Yes
```

The output corroborates the presence of the injection technique by revealing the exact `NormalRoutine` and `KernelRoutine` addresses, effectively isolating the entry point of the foreign code for subsequent forensic analysis.

Following the initial lead provided by the enumeration phase, we executed MalAPC to definitively characterize the suspicious thread activity within `notepad.exe` (PID 4552). The results provide the forensic confirmation required to classify this as a positive injection event.



```
PS D:\volatility3-develop> python3 .\vol.py -f 'D:\DUMPS\Windows 10 x64-Snapshot63.vmem' windows.malapc
Volatility 3 Framework 2.26.2
Progress:  100.00               PDB scanning finished
PID     Process TID     Threat Level    Score   Detection Reason        NormalRoutine   KernelRoutine   APCMode VAD Protection  VAD Type        Module Conte
xt      Hexdump

4552    notepad.exe     1136    CRITICAL        14      Not in loaded module; RWX memory (PAGE_EXECUTE_WRITECOPY); APC during non-alertable wait; APC does n
ot match process baseline       0x7ff8e504fb10  0xf8073debd350  User    PAGE_EXECUTE_WRITECOPY  Unknown NO_MODULE
4c 8b dc 49 89 5b 08 49 89 73 10 4d 89 43 18 57 L..I.[.I.s.M.C.W
48 83 ec 70 49 8b d8 48 8b fa 48 8b f1 49 c7 43 H..pI..H..H..I.C
a8 48 00 00 00 c7 44 24 28 01 00 00 00 0f 57 c0 .H....D$(.....W.
33 c0 0f 11 44 24 30 0f 11 44 24 40 0f 11 44 24 3...D$0..D$@..D$
```

Figure 6: MalAPC executed on an infected memory dump

```
$ python3 vol.py -f infected.vmem windows.malapc --dump-payloads

PID    Process        TID    Threat     Score   Detection Reason
----   -----------    ----   --------   -----   ----------------------------------------
4552   notepad.exe    1136   CRITICAL   14      Not in loaded module; RWX memory
                                                (PAGE_EXECUTE_WRITECOPY); APC during
                                                non-alertable wait; APC does not match
                                                process baseline

NormalRoutine: 0x7ff8e504fb10
KernelRoutine: 0xf8073debd350
APCMode: User
VAD Protection: PAGE_EXECUTE_WRITECOPY
VAD Type: Unknown
Module Context: NO_MODULE

Hexdump:
```

```
4c 8b dc 49 89 5b 08 49 89 73 10 4d 89 43 18 57  L..I.[.I.s.M.C.W
48 83 ec 70 49 8b d8 48 8b fa 48 8b f1 49 c7 43  H..pI..H..H..I.C
a8 48 00 00 00 c7 44 24 28 01 00 00 00 0f 57 c0  .H....D$(.....W.
33 c0 0f 11 44 24 30 0f 11 44 24 40 0f 11 44 24  3...D$0..D$@..D$
```

The plugin automatically isolated the specific APC routine and mapped it to a memory region explicitly marked with `PAGE_EXECUTE_READWRITE` permissions. This specific permission set is a critical forensic artifact, as legitimate user-mode applications rarely allocate executable and writable memory for standard operations.

Furthermore, the tool successfully extracted the raw payload residing at the `NormalRoutine` address (`0x7ff8e504fb10`). The visible byte sequence in the hexdump—beginning with `0x4c 0x8b 0xdc`—corresponds to legitimate x64 function prologue instructions:

```
4c 8b dc          mov r11, rsp       ; Save stack pointer
49 89 5b 08       mov [r11+8], rbx   ; Save RBX register
49 89 73 10       mov [r11+0x10], rsi ; Save RSI register
4d 89 43 18       mov [r11+0x18], r8  ; Save R8 register
57                push rdi            ; Save RDI
48 83 ec 70       sub rsp, 0x70       ; Allocate 112 bytes stack
```

While this appears to be proper function prologue code, its location in memory that is:

1. Not backed by any loaded module
2. Marked with RWX permissions
3. Targeted by a user-mode APC injection

This combination effectively validates that the queued APC was intended to trigger the execution of arbitrary, injected code rather than a benign system function.

3.3 Comparative Analysis: MalAPC vs Malfind

To demonstrate MalAPC's specialized capabilities, we compared its output against Volatility's general-purpose `malfind` plugin on the same infected sample:

Table 4: Detection Comparison: MalAPC vs Malfind

Feature	**MalAPC**	**Malfind**
Detection Method	APC queue analysis + memory inspection	Memory region scanning only
Injection Context	Identifies specific APC injection mechanism	Generic suspicious memory detection
False Positives	Very low (JIT detection, whitelist, baselines)	Higher (flags legitimate JIT, .NET)
Threat Scoring	Multi-factor scoring (1-15+)	Binary (suspicious/not suspicious)
Analysis Speed	Faster (targeted APC inspection)	Slower (scans all VADs)
Output Detail	Thread-level precision, execution context	Process-level, memory regions only
Stealth Detection	Detects pre-execution injection	Detects post-injection memory

While `malfind` would detect the RWX memory region in notepad.exe, it would not reveal:

- That the code is queued via APC injection
- Which specific thread is targeted (TID 1136)
- The exact APC routine addresses
- Thread state context (non-alertable wait)
- Whether it matches expected process behavior

MalAPC provides this additional context, allowing analysts to understand not just *where* the malicious code is, but *how* it will execute.

4 Technical Implementation Details

4.1 Kernel Structure Traversal

Both plugins rely on accurately parsing Windows kernel structures. The core data structures involved in APC analysis include:

4.1.1 _ETHREAD Structure

The `_ETHREAD` structure represents a thread object in the Windows kernel. Key fields accessed by our plugins:

```
for thread in proc.ThreadListHead.to_list(
    f"{symbol_table_name}{constants.BANG}_ETHREAD",
    "ThreadListEntry"
):
    thread_id = int(thread.Cid.UniqueThread)
    apc_state = thread.Tcb.ApcState
```

Listing 11: ETHREAD Structure Traversal

4.1.2 _KAPC_STATE Structure

Contains the APC queue lists for a thread:

- `ApcListHead[0]`: Kernel-mode APC queue
- `ApcListHead[1]`: User-mode APC queue
- `KernelApcPending`: Boolean flag
- `UserApcPending`: Boolean flag

4.1.3 _KAPC Structure

Represents an individual APC object:

```
# Critical fields
apc.Type                # APC object type identifier
apc.ApcMode             # 0 = Kernel, 1 = User
apc.Inserted            # Boolean: is APC in queue?
apc.NormalRoutine       # Pointer to user callback
apc.KernelRoutine       # Pointer to kernel callback
apc.NormalContext       # Context parameter
apc.SystemArgument1     # Additional parameter
apc.SystemArgument2     # Additional parameter
```

Listing 12: KAPC Structure Fields

4.2 VAD Tree Analysis

The Virtual Address Descriptor (VAD) tree provides crucial memory mapping information:

```python
def _find_vad_for_address(self, addr: int, proc,
    protection_values: dict):
    """Find VAD information for an address"""
    try:
        for vad in proc.get_vad_root().traverse():
            vad_start = vad.get_start()
            vad_end = vad_start + vad.get_size()

            if vad_start <= addr < vad_end:
                protection = vad.get_protection(
                    protection_values,
                    vadinfo.winnt_protections
                )

                return {
                    'start': vad_start,
                    'end': vad_end,
                    'size': vad.get_size(),
                    'protection': protection,
                    'type': str(vad.VadType) if hasattr(vad, '
                       VadType') else "Unknown"
                }
    except Exception as e:
        vollog.debug(f"Error finding VAD: {e}")

    return None
```

Listing 13: VAD Tree Traversal for Address Resolution

4.3 Process Whitelisting Mechanism

To reduce false positives, MalAPC implements a whitelisting system:

```python
LEGITIMATE_PROCESSES = {
    "svchost.exe",          # System services
    "lsass.exe",            # Local security authority
    "csrss.exe",            # Client/server runtime
    "explorer.exe",         # Windows explorer
    "searchapp.exe",        # Windows search app
    "dwm.exe",              # Desktop window manager
    # ... additional system processes
}

def whitelist_filter(proc):
    if not original_filter(proc):
        return False
    proc_name = self._get_process_name(proc).lower()

    if not proc_name or proc_name == "unknown":
        return True  # Analyze unknown processes

    return proc_name not in self.LEGITIMATE_PROCESSES
```

Listing 14: Whitelist Implementation

Important: All entries in the whitelist must be lowercase to match the case-insensitive comparison performed by the filter.

4.4 Cross-Platform Kernel Address Detection

One critical feature is pr oper detection of kernel addresses on both 32-bit and 64-bit systems, ensuring accurate address classification across architectures.

```
def _is_kernel_address(self, addr: int) -> bool:
    """Check if address is in kernel space (32-bit and 64-bit)"
       ""
    # 64-bit Windows: kernel >= 0xFFFF800000000000
    # Practical threshold: >= 0xF80000000000
    if addr >= 0xF80000000000:
        return True

    # 32-bit Windows: kernel >= 0x80000000
    if 0x80000000 <= addr <= 0xFFFFFFFF:
        return True

    return False
```

Listing 15: Enhanced Kernel Address Detection

This prevents false positives like the SearchApp.exe case where legitimate 32-bit kernel APCs were misclassified as user-space injection.

4.5 Error Handling and Robustness

MalAPC implements defensive programming practice to handle corrupted memory:

```
def _extract_apc_details(self, apc, apc_state, thread) -> dict:
    details = {
        'kernel_routine': 0,
        'normal_routine': 0,
        'apc_mode': "Unknown"
    }

    try:
        if hasattr(apc, 'KernelRoutine') and apc.KernelRoutine:
            details['kernel_routine'] = int(apc.KernelRoutine)

        if hasattr(apc, 'NormalRoutine') and apc.NormalRoutine:
            details['normal_routine'] = int(apc.NormalRoutine)

        if hasattr(apc, 'ApcMode'):
            details['apc_mode'] = "Kernel" if int(apc.ApcMode)
               == 0 else "User"

    except Exception as e:
        vollog.debug(f"Error extracting APC details: {e}")

    return details
```

Listing 16: Safe Attribute Access Pattern

This pattern is used throughout both plugins to gracefully handle:

- Missing structure fields
- Null pointer dereferences
- Invalid memory reads
- Type conversion errors

5 Future Scope

Looking ahead, we plan to evolve MalAPC beyond its current static heuristic model into a more dynamic analysis tool. While the current threat scoring system is effective, we aim to integrate a lightweight emulation engine (such as Unicorn) to safely execute detected shellcode stubs in isolation; this would allow the plugin to reveal resolved API imports and actual behavior rather than just flagging memory anomalies. Additionally, we want to refine the JIT detection logic by replacing the current entropy-variance checks with a machine-learning-based classifier trained on legitimate runtime frameworks (like .NET and V8), which would significantly reduce false positives in complex browser environments and make the tool even more reliable for enterprise-grade memory forensics.

6 Why We Should Win the 2025 Volatility Plugin Contest

As security practitioners, we constantly watch the cat-and-mouse game between attackers and defenders evolve. Malware authors have moved beyond simply hiding files on disk; they are now abusing the fundamental mechanisms of the operating system to blend in. For us, the motivation behind this project was born from a specific frustration we encountered during investigations: the blind spot surrounding Asynchronous Procedure Calls (APCs). We realized that while many tools exist to find where malware is hiding in memory, very few effectively show how it is achieving execution. APCs have become a favorite vehicle for advanced threats—used in techniques like "Early Bird" injection and AtomBombing—precisely because they look like legitimate system noise. We built the APCWatch and MalAPC suite to turn that noise into a clear signal.

Our approach differs from standard tooling by offering a complete workflow rather than just a single scanner. We recognized that analysts need two things: broad visibility and pinpoint precision. By designing APCWatch to map the entire landscape of thread activity, and MalAPC to surgically hunt for anomalies like "floating" code and RWX violations, we have created a system that is both comprehensive and highly specific to modern threats. Winning this contest would be a tremendous honor, but our true goal is to arm the Volatility community with better sensors. As evasion techniques become more complex, our forensic capabilities must adapt to match them. We believe these plugins fill a crucial void in the current ecosystem, providing analysts with the ability to detect not just the presence of malware, but the specific mechanism of its persistence. We are proud to contribute this work to the foundation and look forward to seeing how it helps fellow researchers uncover what was previously invisible.

7 References

1. The Volatility Foundation. *Volatility Plugin Contest | The Volatility Foundation | Memory Forensics*. The Volatility Foundation - Promoting Accessible Memory Analysis Tools Within the Memory Forensics Community, 8 December 2025. Available: https://volatilityfoundation.org/volatility-plugin-contest/

2. volatilityfoundation. *volatility/volatility/plugins/malware/malfind.py at master · volatilityfoundation/volatility*. GitHub. Available: https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py

3. Stevewhims. *Asynchronous Procedure Calls - Win32 apps*. Microsoft Learn. Available: https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-c

4. EliotSeattle. *Windows Kernel Opaque Structures - Windows drivers*. Microsoft Learn. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/eprocess

5. MITRE ATT&CK. *Process injection: Asynchronous procedure call, sub-technique T1055.004 - Enterprise*. Available: https://attack.mitre.org/techniques/T1055/004/

6. Alvinashcraft. *Memory Protection Constants (WinNT.h) - Win32 apps*. Microsoft Learn. Available: https://learn.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

7. Nguyen, A. Q., & Deng, D. (2015). *Unicorn: Next generation CPU emulator framework*. Black Hat USA. Available: https://www.blackhat.com/us-15/briefings.html#unicorn-next-generation-cpu-emulator-framework