Defensive Rootkits: Engineering Kernel-Level Malware Analysis from Ring 0

By Alberto Marín

Published: 2026-03-24 · Archived: 2026-05-06 02:01:23 UTC

Introduction: The Sandbox Evasion Crisis

We are in the middle of the next transition. This time, the failures are silent. Modern malware is industrialized: sold as-a-service, rapidly iterated, and designed to scale. At the same time, defenders increasingly depend on automated pipelines. When those pipelines are evaded, failures are silent; which creates a dangerous gap between what is detected and what is actually happening.

Many of the malware families we see most often (e.g. infostealers, loaders, RATs and ransomware) reflect this shift. These categories dominate both prevalence and impact: infostealers account for a large share of infections, while loaders and RATs support intrusion chains, and ransomware remains a primary monetization stage. All have largely mastered sandbox evasion. Before delivering a payload, these samples interrogate their environment: they check hardware characteristics, measure execution timing and scan for telltale artifacts in the system before proceeding. If anything smells like an analysis environment, they shut down silently. The analyst sees nothing. The report says “no malicious behavior observed.” The threat passes through undetected.

This is not an edge-case problem. MITRE ATT&CK’s Virtualization/Sandbox Evasion technique (T1497) is consistently among the top-ten most observed techniques in real-world incidents. Commodity crimeware families like LummaC2, GuLoader and xLoader routinely apply evasion logic that was, just a few years ago, associated only with sophisticated threat actors. What was once a sign of sophisticated threat actors is now commodity technique.

Why Traditional Analysis Platforms Fall Short

The three dominant monitoring approaches to malware analysis each have structural blind spots.

User-mode monitoring instruments the analyzed process directly (typically by injecting a DLL that intercepts API calls). It is the simplest approach and works well against unsophisticated threats. Against anything that checks for foreign DLLs in its own memory, unusual executable regions, or simply enumerates the process list and looks for monitoring tools, it fails quickly. The hooks are visible to any code that knows where to look. And today’s commodity malware knows where to look.

For instance, malware can detect injected DLLs used for monitoring, read their own memory to look for inline hooks or even manually map needed .dlls in their memory to bypass already hooked functions. The most aggressive approach is to bypass user-mode hooks entirely by issuing `SYSCALL` instructions directly rather than calling through `ntdll.dll`, jumping over any user-mode intercept layer completely. All of these bypass techniques are well-documented and implemented in many off-the-shelf malware kits.

Hypervisor-based monitoring operates from outside the guest operating system, which eliminates many in-guest artifacts. This is a genuine architectural improvement. However, hypervisor-based analysis faces a deeper structural challenge: there is no easy access to the full context of every syscall being executed by monitored processes. Translating raw hardware-level observations into meaningful OS-level semantics (which process wrote to which registry key, what arguments a specific syscall received), requires complex memory-introspection and mapping that introduces both latency and analytical complexity. This semantic gap also makes it harder to bypass anti-analysis techniques and to assist malware detonation in the targeted way that a kernel-level component can. The performance overhead of managing VM exits for every relevant event compounds these difficulties.

Bare-metal analysis eliminates virtualization artifacts entirely by executing samples directly on physical hardware, rather than in a virtualized environment. However, this architectural choice alone does not capture all malicious activity. Without active process and behavior monitoring, malware that relies on persistence or staging can evade detection: for example, a loader that sets a Run key and exits cleanly will never reveal its payload unless the analysis system observes actions across reboots or extended execution. Rebuilding and re-imaging machines after each analysis is slow, expensive, and impractical at the scale required to process hundreds of thousands of samples daily for a Malware Intelligence solution. Bare-metal analysis is therefore most effective for targeted, in-depth investigations, but even then, it must be paired with monitoring mechanisms that track process behavior and system changes over time.

Kernel-level monitoring addresses the shortcomings of all three approaches from a different angle. By operating inside the guest OS at Ring 0, it has direct access to every system call, every kernel data structure, and every piece of runtime state the OS itself processes. There is no semantic gap: the kernel already knows which process made which call, what the arguments were, and what the result should be. Interception and modification are surgical and precise. Detection surface is minimal when implemented correctly.

Each approach fails to answer the same fundamental question: *If malware calls an operating system function to interrogate its environment, can we intercept that call and return something believable, without the interception itself being detectable?* User-mode cannot; it operates above the call path. Hypervisors can intercept certain instructions through VM-exit handling, but face inherent limits in manipulating OS-level semantics coherently from below the OS. Bare-metal cannot, without kernel modification. Only a kernel-level component can do this cleanly and at scale – but even it must contend with certain low-level hardware checks.

The Core Insight

The answer lies in operating at the same privilege level as both the operating system and the most sophisticated malware: Ring 0. A kernel-mode component running inside the analysis VM has access to every system call, every kernel data structure, and every piece of information that the OS itself processes. There is no semantic gap: the kernel already knows which process made which call, what the arguments were, and what the result should be. We simply intercept the answer before it reaches the caller and, where necessary, change it.

But our goal is not just evasion resistance. We designed this system to do something more: to **follow malware through its complete infection chain**. Observing not just the initial dropper's environment checks, but the persistence mechanisms it establishes, the next-stage payloads those mechanisms trigger, and the behavioral evidence that malware routinely destroys before analysts can examine it. We call the result a **defensive rootkit**.

The architectural answer to this crisis has three interlocking elements. First, a kernel-mode driver running at Ring 0 (below every user-mode monitoring technique), intercepts all system calls with full semantic context: which process made the call, what the arguments were, and what the correct answer would be. This is the foundation that makes anti-evasion surgical rather than heuristic. Second, a hypervisor layer operating outside the guest OS is the most robust place to handle hardware-level probes such as `CPUID` fingerprinting, `RDTSC` -based timing anomalies, CPU feature flag queries, and VM artifact masking (device identifiers, firmware strings, and other platform-identifying values that are cleanly interceptable from outside the guest). Third, a real-time persistence detection and execution engine follows malware through multi-stage infection chains (e.g. detecting registry Run key writes, scheduled task registrations, and service entries as they occur), then immediately triggering registered next-stage payloads without requiring a system reboot. Together, these three layers address the full attack surface of sandbox evasion: the OS-level checks that a kernel driver intercepts, the hardware-level checks that a hypervisor handles, and the staging gap that persistence-based payloads depend on. The following sections explain each layer in technical depth.

Defensive Rootkits: The Concept

Defensive Rootkits: What the Term Means, and Why We Use It Deliberately

A rootkit, in the traditional sense, is a kernel-mode component that uses system call interception, stealth techniques, and direct kernel structure manipulation to modify the system's behavior; typically to conceal malware from the operating system and from security tools. The term carries connotations of malice, concealment, and active deception at the kernel level.

We use the term deliberately. What we built is a **defensive rootkit**, a kernel-mode component that applies exactly the same techniques, with the opposite purpose: not to conceal malware from defenders, but to conceal the analysis infrastructure from malware, and to intercept the environment checks that evasive samples use to decide whether to detonate.

The symmetry is not coincidental. It is the point.

Malware rootkits hook `NtQuerySystemInformation` to hide their processes from process enumeration. Our defensive rootkit hooks the same function to hide analysis tools from malware. Malware rootkits intercept registry queries to conceal their configuration. Ours intercepts registry queries to spoof hardware identifiers. Malware rootkits use filesystem minifilters to hide their files. Ours uses the same minifilter mechanism to intercept file deletions and preserve forensic evidence. Same mechanism, opposite direction.

It is also worth noting why traditional offensive rootkits are nowadays relatively rare in commodity malware: Windows kernel protection mechanisms, such as PatchGuard (Kernel Patch Protection), actively detect and respond to unauthorized modifications of critical kernel structures, making it significantly more difficult and risky for malware to operate stably at the kernel level. Also operating at the kernel level requires a significant engineering investment. This same engineering investment is precisely what enables our defensive rootkit to operate below the detection horizon of any user-mode analysis technique.

Beyond Concealment: What Defensive Rootkits Actually Do

It would be easy to reduce defensive rootkits to their anti-evasion capabilities and to describe them as sophisticated sandbox-hardening tools. That framing undersells what they enable.

Anti-evasion is the first capability: ensuring that malware detonates and executes its first stage. But our kernel driver does not stop there. Once malware is running and actively executing, our driver:

- **Monitors every persistence mechanism** the malware establishes (e.g. registry Run keys, startup folder files, service registrations, scheduled tasks) in real time, from kernel space, before malware has any chance to obscure them.
- **Forces execution of next-stage payloads** without waiting for an actual system reboot or user login event, allowing analysts to observe the complete infection chain (dropper to loader to final payload) in a single analysis session.
- **Preserves forensic evidence** that malware routinely destroys: files that are dropped and deleted before analysts can examine them, memory contents of processes at the moment of termination, and the complete chain of spawned processes and injection targets.
- **Conceals the analysis infrastructure from malware**: monitoring agents, analysis tools, and other infrastructure processes running alongside the sample are hidden at the kernel level. Malware attempting to enumerate running processes, open handles to known tools, or detect analysis artifacts sees a clean environment, ensuring it does not abort before executing its payload.
- **Automatically tracks injections** across process boundaries, ensuring that even code injected into unrelated processes remains under observation without requiring manual configuration.
- **Detects and resists kernel-level tampering**: if a sample achieves Ring 0 control inside the analysis VM and attempts to remove or bypass the driver's hooks, a dedicated protection thread detects the modification and restores it within one second. The tampering attempt itself is logged as a high-confidence signal of a kernel-aware adversary. This is not a capability analysis platforms commonly provide.

The goal is not just to see stage one. It is to see everything, including the behavior of samples sophisticated enough to fight back.

Zynap's Mixed Approach: Better Together

Before describing the kernel driver in detail, it is worth explaining why Zynap uses both kernel-level and hypervisor-level monitoring and why this combination delivers meaningfully better results than either approach deployed alone.

What Each Layer Does Best, and Why Neither Is Sufficient Alone

Hypervisor monitoring excels at capabilities that in-guest software fundamentally cannot replicate. Consider the `CPUID` instruction: it is not a system call. Malware executes it directly in user space and receives the result directly from the CPU. A hypervisor handles the resulting VM exit and can synthesize any response it chooses (e.g. erasing VMware or VirtualBox vendor strings from CPUID results, returning physical hardware identifiers and masking CPU feature flags that are anomalous in virtual environments).

Similarly, hardware-level timing anomalies (measurable differences in instruction latency that reveal virtualized execution) are most cleanly handled at the hypervisor level. Careful VM-exit management and virtual

environment tuning can bring timing characteristics within the range of physical hardware.

And critically: hypervisor monitoring is isolated from the guest OS. A malware sample that achieves full kernel-level control inside the guest cannot tamper with or detect hypervisor-level monitoring, because the hypervisor is invisible from below. For the most capable samples, this provides an additional safety boundary.

Kernel-level monitoring inside the guest does things that a hypervisor cannot do without enormous complexity. System calls carry rich semantic context: the calling process, the exact parameters, the OS-level return value. Intercepting `NtQuerySystemInformation` at the kernel level gives us the caller's process ID, the specific information class being queried, and direct write access to the returned buffer (all without any semantic gap translation). We know *which monitored malware process* made the call, *what it was asking for*, and we can modify the answer with surgical precision.

Kernel-level monitoring also enables capabilities that are inherently about OS-level state: watching for registry writes to persistence-related keys, intercepting file deletion to preserve forensic evidence, detecting code injection into other processes, and tracking the full chain of child processes spawned by malware. A hypervisor monitoring below the OS must infer OS-level intent indirectly (working through raw memory introspection and complex OS structure reconstruction rather than accessing this information natively as the kernel already does). This involves managing exceptions, forcing VM exits to occur (for example, by changing page protections to cause access faults), and dealing with the resulting engineering complexity. The kernel, by contrast, has all this information natively and immediately available at the point of every syscall.

A hypervisor-only approach covers `CPUID`, low-level timing, and VM artifact masking, but struggles with the semantic richness needed for persistence detection, injection tracking, and surgical system call manipulation. A kernel-only approach covers all OS-level semantics beautifully, but cannot address hardware-level probes that execute below the OS entirely.

Evasive malware does not constrain itself to one probe type. A sophisticated sample checks `CPUID` *and* queries disk size via system call *and* reads registry keys *and* measures sleep precision *and* enumerates the process list. Any analysis approach that only covers some of these is allowing the rest to be used as reliable detection signals.

How They Complement Each Other in Practice

In Zynap's architecture, the two layers divide responsibilities according to where each has a genuine advantage:

- **Hypervisor layer**: handles `CPUID` spoofing, `RDTSC`-based timing anomalies, low-level CPU feature masking, and VM artifact masking at the hardware configuration level (device identifiers, firmware strings, and other platform-identifying values that are straightforward to intercept and replace from outside the guest). It provides the "hardware looks real" layer that is visible to any code executing on the CPU, regardless of privilege level.
- **Kernel driver layer**: handles everything that requires OS-level context. For example: syscall interception, registry and filesystem manipulation, process tracking, injection detection, persistence monitoring, and forensic evidence capture. It provides the "the OS looks real" layer, controlling what the operating system reports to processes and what the kernel does on their behalf.

Together, these layers address the full spectrum of environment checks that evasive malware performs. A sample using hardware-level `CPUID` fingerprinting encounters our hypervisor response. The same sample querying registry keys for hardware identifiers encounters our kernel-level registry filter. The same sample measuring disk size through a system call encounters our SSDT hook. For any individual evasion technique, at least one layer of defense covers it. For most techniques, both do.

This is not a configuration option, it is an architectural choice. And it delivers a consistent, coherent picture of a physical workstation at every level a malware sample probes.

One important clarification about the role of the kernel driver within the broader platform: **the defensive rootkit is not Zynap's primary monitoring solution**. Capturing a complete picture of what malware actually does at the API level (which functions it calls, with what arguments, in what order) is the responsibility of a dedicated kernel monitoring driver that operates alongside the defensive rootkit. That driver intercepts high-level API calls (without any user-mode hooks) and provides the behavioral record analysts rely on. SSDT hooks on syscalls alone are a poor fit for this role: syscalls operate at a lower abstraction level than the Win32 or NT APIs malware actually uses, so they produce a noisier, harder-to-interpret call record compared to monitoring at the API layer directly.

The defensive rootkit's direction is different: **improving analysis coverage** by ensuring malware detonates and executes fully (bypassing anti-analysis checks), detecting and triggering persistence mechanisms, and preserving forensic artifacts that malware destroys. The two drivers are complementary and communicate with each other. For example, when the monitoring driver detects a process injection, it can notify the defensive rootkit to add the target process to its tracked list, and vice versa, ensuring both drivers maintain consistent scope across the full infection chain.

This post focuses on the kernel driver layer: its architecture, implementation, and the engineering discipline required to make it reliable in production. The hypervisor layer (`CPUID` spoofing, timing normalization, VM artifact masking, and low-level hardware fingerprint masking) is a subject we'll cover in a dedicated follow-up post.

Full Platform Component Map

The platform is organized across two privilege layers: Kernel Driver (Ring 0) and Hypervisor (Ring -1 / VMM). Each with its own subsystems and clean interface boundaries. The kernel driver alone is organized into mulitple subsystems. At the highest level, the two layers and their major components interact as follows:

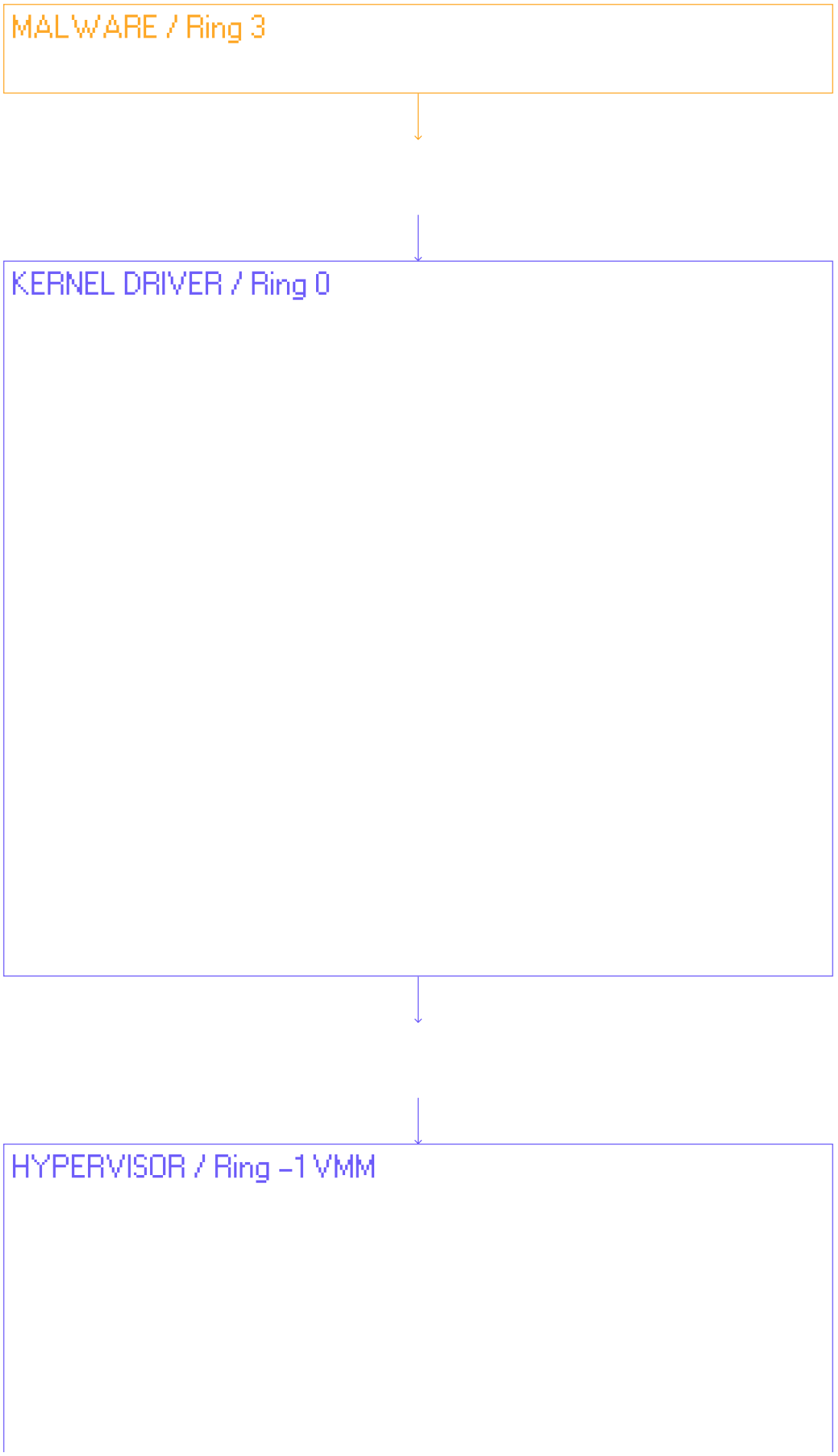
MALWARE / Ring 3
KERNEL DRIVER / Ring 0
HYPERVISOR / Ring -1 VMM

Figure 1. Full Platform Compenent Map

An important design principle: **the driver acts only on specific processes**. It maintains three distinct process lists, each with a well-defined behavioral role:

- **Monitored processes** are the malware samples under analysis. All SSDT hook logic, minifilter interception, persistence detection, and injection tracking apply exclusively to processes on this list. If the calling process is not monitored, hooks pass the call through to the original syscall without modification. Resulting in zero overhead and zero behavioral change for the rest of the system.
- **Blacklisted processes** are explicitly excluded from all driver logic. A blacklisted process is neither monitored nor subjected to SSDT hook behavior (its syscalls pass through unmodified and it is not tracked for persistence or injection events). This is used to whitelist known-good system components or support processes that must operate without interference from the analysis driver.
- **Hidden processes** are concealed from monitored processes. When a monitored malware process enumerates running processes (e.g. via `NtQuerySystemInformation`), entries for hidden processes are stripped from the returned list. This allows analysis tooling and any other infrastructure processes to remain invisible to the sample under observation.

The key invariant is that monitored is mutually exclusive with both hidden and blacklisted: a process under active analysis is neither concealed nor exempted. Hidden and blacklisted can coexist, which gives flexibility when a process needs to be both invisible to malware and excluded from hook logic. System-wide hooks exist, but their logic executes only when the calling process is on the relevant list, keeping overhead minimal for everything outside the monitored set.

The Engine: SSDT Hooking on x64

The System Service Descriptor Table (SSDT) is the kernel's dispatch table for system calls. When user-mode code executes a `SYSCALL` instruction, the processor transitions to Ring 0 and the kernel uses a numeric index to look up the corresponding handler in the SSDT. Intercepting that lookup is how we intercept system calls.

SSDT hooking is a technique that has been covered extensively in the community (there's a wealth of public literature on the mechanics, and the references at the end of this section point to some great resources). What we'll go through here is how we implement it reliably in a production analysis platform.

On 32-bit Windows, `KeServiceDescriptorTable` was an exported symbol and modifying it was straightforward. On 64-bit Windows, several things changed: the symbol is no longer exported, the table lives in write-protected memory, and multi-processor systems require careful synchronization to modify it safely. The hooking process requires a chain of non-trivial steps.

> **A note on diagnostic utilities in code excerpts**: `DBG_INFO`, `DBG_WARNING`, `DBG_ERROR` (and `W`-suffixed Unicode variants) wrap `DbgPrintEx` and compile to nothing in release builds (zero overhead in production). Base variants are callable at any IRQL; `W` variants add an IRQL check that suppresses output above `PASSIVE_LEVEL`, where `%wZ` format specifiers can page-fault. `LgPrintfW` writes

> structured entries to a persistent log file and survives release builds, callable up to `DISPATCH_LEVEL`: synchronous inline write at `PASSIVE_LEVEL`; blocking work item at `APC_LEVEL`; fire-and-forget async at `DISPATCH_LEVEL` (returns `STATUS_PENDING`, completes later at `PASSIVE_LEVEL`).

Locating the SSDT Dynamically

Since `KeServiceDescriptorTable` is not exported, we must locate it by navigating the kernel's own code. Our starting point is the LSTAR MSR (`0xC0000082`), which holds the address of the system call entry point on x64 Windows. From there, we apply a two-stage byte-pattern search through the kernel code. Stage 1 scans from KiSystemCall64Shadow for an `LFENCE` instruction followed by a characteristic pattern from the Meltdown mitigation code, then follows the relative jump to locate `KiSystemServiceUser` . The specific pattern:

```
fffff803`154b43cc  0faee8          lfence
fffff803`154b43cf  65c60425...00   mov byte ptr gs:[853h], 0
fffff803`154b43d8  e93d0a97ff      jmp nt!KiSystemServiceUser
```

Stage 2 scans from `KiSystemServiceUser` for the `LEA R10, [RIP + offset]` instruction (opcode `4C 8D 15`) that directly references `KeServiceDescriptorTable` :

```
 fffff807`05211c84  4c8d1535fc9e00  lea r10, [nt!KeServiceDescriptorTable]
```

The signed 32-bit relative offset in bytes 3-6 of this instruction, added to the instruction's address plus its 7-byte length, gives us the table address directly. Each stage searches within a 0x400-byte window, which has proven sufficient and stable across Windows 10 and 11 builds.

This would be the complete function for SSDT location:

```
 _Must_inspect_result_
static PSERVICE_DESCRIPTOR_TABLE_ENTRY SdtGetSSDTBaseAddr()
{
    PUCHAR KiSystemCall64Shadow = NULL;
    PUCHAR KiSystemServiceUser = NULL;
    PUCHAR StartAddr = NULL;
    PUCHAR i = NULL;
    UCHAR  b1 = 0, b2 = 0, b3 = 0;

    LONG KiSystemServiceUser_offset = 0;

    // Latest Win10 with Meltdown mitigation or Windows 11 returns KiSystemCall64Shadow in MSR instead of KiSyst
    // Get KiSystemCall64Shadow address from LSTAR MSR
    KiSystemCall64Shadow = (PVOID)__readmsr(0xC0000082);
    StartAddr = KiSystemCall64Shadow;

    // Validate the MSR-provided address as it comes from hardware (could be invalid if MSR is corrupted)
```

```
if (!StartAddr || !MmIsAddressValid(StartAddr))
{
    DBG_ERROR("[!] Invalid StartAddr from MSR: 0x%p", StartAddr);
    return NULL;
}

// Stage 1: Find KiSystemServiceUser
for (i = StartAddr; i <= StartAddr + 0x400; i++)
{
    // Post-Meltdown Windows 10 and Windows 11, msr[0xC0000082] = nt!KiSystemCall64Shadow
    // We need search back
    //  fffff803`154b43c8 4883c408        add     rsp,8
    //  fffff803`154b43cc 0faee8          lfence  <-- all of these bytes
    //  fffff803`154b43cf 65c604255308000000 mov   byte ptr gs:[853h],0 <-- last byte
    //  fffff803`154b43d8 e93d0a97ff      jmp     nt!KiSystemServiceUser (fffff803`14e24e1a)  Branch <-- fi

    // Check if we can safely read 3 bytes for LFENCE pattern
    if (!MmIsAddressValid(i) || !MmIsAddressValid(i + 2))
    {
        continue;
    }

    b1 = *(i);
    b2 = *(i + 1);
    b3 = *(i + 2);

    // Search for lfence instruction (opcodes 0F AE E8 -> LFENCE)
    if (b1 == 0x0f && b2 == 0xae && b3 == 0xe8)
    {
        // Once found lfence instruction, search for jmp nt!KiSystemServiceUser, it should be really close
        // To ensure we are in the correct jmp, we check previous byte (0x00) (from "mov byte ptr gs:[853h],
        // and check for byte 0xe9 (relative jmp from "jmp nt!KiSystemServiceUser")
        PUCHAR j = NULL;
        for (j = i; j < i + 0x64; j++)
        {
            if (!MmIsAddressValid(j - 1) || !MmIsAddressValid(j + 4))
            {
                continue;
            }

            b1 = *(j - 1);
            b2 = *(j);

            if (b1 == 0x00 && b2 == 0xe9)
            {
                // We found "jmp nt!KiSystemServiceUser"
                // We use j+1 as origin to skip first byte (opcode for relative jmp 0xe9)
```

```
                // 0xe9 xx xx xx xx (4 bytes for relative offset to nt!KiSystemServiceUser)
                RtlCopyMemory(&KiSystemServiceUser_offset, j + 1, 4);

                // fffff803`154b43d8 e93d0a97ff      jmp     nt!KiSystemServiceUser (fffff803`14e24e1a)
                // jmp nt!KiSystemServiceUser instruction is 5 bytes long

                // Calculate the target address properly maintaining pointer type
                // j was declared as PUCHAR, as we are using it to read individual bytes when checking for
                // the jump pattern and PUCHAR makes byte-by-byte reading clear and explicit.
                // The cast to ULONG_PTR for address arithmetic is proper for pointer calculations
                KiSystemServiceUser = (PUCHAR)(KiSystemServiceUser_offset + (ULONG_PTR)j + 5);

                break;
            }
        }
    }
    if (KiSystemServiceUser)
    {
        // Avoid doing unnecessary iterations if we already found nt!KiSystemServiceUser
        break;
    }
}

if (!KiSystemServiceUser)
{
    DBG_ERROR("[!] Unable to find nt!KiSystemServiceUser");
    return 0;
}

// At this point, we should have found nt!KiSystemServiceUser address, which is close to nt!KiSystemServiceR
// nt!KiSystemServiceRepeat is the actual function that references SSDT as it can be seen:

/*
    lkd> u nt!KiSystemServiceRepeat

    nt!KiSystemServiceRepeat:
    fffff807`05211c84 4c8d1535fc9e00  lea     r10,[nt!KeServiceDescriptorTable (fffff807`05c018c0)]
    fffff807`05211c8b 4c8d1daead8e00  lea     r11,[nt!KeServiceDescriptorTableShadow (fffff807`05afca40)]
    fffff807`05211c92 f7437880000000  test    dword ptr [rbx+78h],80h
    fffff807`05211c99 7413            je      nt!KiSystemServiceRepeat+0x2a (fffff807`05211cae)
    fffff807`05211c9b f7437800002000  test    dword ptr [rbx+78h],200000h
    fffff807`05211ca2 7407            je      nt!KiSystemServiceRepeat+0x27 (fffff807`05211cab)
    fffff807`05211ca4 4c8d1d55af8e00  lea     r11,[nt!KeServiceDescriptorTableFilter (fffff807`05afcc00)]
    fffff807`05211cab 4d8bd3          mov     r10,r11
*/

// Proceed to search for fffff807`05211c84 4c8d1535fc9e00  lea     r10,[nt!KeServiceDescriptorTable
```

```
    // like we usually do in earlier Windows10 versions

    // Stage 2: Find SSDT reference
    PSERVICE_DESCRIPTOR_TABLE_ENTRY KeServiceDescriptorTable = NULL;

    LONG KeServiceDescriptorTable_offset = 0;

    StartAddr = KiSystemServiceUser;
    for (i = StartAddr; i <= StartAddr + 0x400; i++)
    {
        // Check if we can safely read 7 bytes (full instruction length)
        if (!MmIsAddressValid(i) || !MmIsAddressValid(i + 6))
        {
            continue;
        }

        b1 = *(i);
        b2 = *(i + 1);
        b3 = *(i + 2);

        if (b1 == 0x4c && b2 == 0x8d && b3 == 0x15)
        {
            // We found "lea r10,[nt!KeServiceDescriptorTable]"
            // We use i+3 as origin to skip first three bytes (opcodes of lea instruction)
            // 0x4c 0x8d 0x15 xx xx xx xx (4 bytes for relative offset to nt!KeServiceDescriptorTable)
            RtlCopyMemory(&KeServiceDescriptorTable_offset, i + 3, 4);

            //fffff807`05211c84 4c8d1535fc9e00  lea     r10, [nt!KeServiceDescriptorTable(fffff807`05c018c0)]
            // lea r10, [nt!KeServiceDescriptorTable] instruction is 7 bytes long
            KeServiceDescriptorTable =
                (PSERVICE_DESCRIPTOR_TABLE_ENTRY)(KeServiceDescriptorTable_offset + (ULONG_PTR)i + 7);
            break;
        }
    }

    DBG_INFO("[+] SdtGetSSDTBaseAddr offset: 0x%lx", KeServiceDescriptorTable_offset);
    DBG_INFO("[+] SdtGetSSDTBaseAddr addr: 0x%p", KeServiceDescriptorTable);

    return KeServiceDescriptorTable;
}
```

For deeper reading on SSDT internals, location patterns, and evolution across Windows versions, the following are excellent references:

- Windows 11 SSDT and ShadowSSDT fetch problem
- SSDT Hook – MoukaNotes

- A Syscall Journey in the Windows Kernel – Alice Climent-Pommeret (archived version)

Compatibility: Fallback Path for Earlier Windows 10 Builds

The two-stage function above targets post-Meltdown Windows 10 and Windows 11, where the LSTAR MSR points to `KiSystemCall64Shadow`. Earlier Windows 10 builds (pre-Meltdown mitigation) do not have `KiSystemCall64Shadow`, the MSR points directly to `KiSystemCall64`, and `KiSystemServiceRepeat` (which contains the `LEA R10, [KeServiceDescriptorTable]` reference) is close enough to the entry point that a single-stage scan suffices. The driver includes a dedicated fallback function for this case:

```
 _Must_inspect_result_
static PSERVICE_DESCRIPTOR_TABLE_ENTRY SdtGetSSDTBaseAddrOld()
{
    // Get KiSystemCall64 address from LSTAR MSR
    PUCHAR  StartSearchAddress = (PUCHAR)__readmsr(0xC0000082);

    // Validate the MSR-provided address as it comes from hardware (could be invalid if MSR is corrupted)
    if (!StartSearchAddress || !MmIsAddressValid(StartSearchAddress))
    {
        DBG_ERROR("[!] Invalid StartSearchAddress from MSR: 0x%p", StartSearchAddress);
        return NULL;
    }

    PUCHAR  EndSearchAddress = StartSearchAddress + 0x500;
    PUCHAR  i = NULL;
    UCHAR  b1 = 0, b2 = 0, b3 = 0;
    LONG  offset = 0;
    PSERVICE_DESCRIPTOR_TABLE_ENTRY addr = 0;

    for (i = StartSearchAddress; i < EndSearchAddress; i++)
    {
        // Check if we can safely read 7 bytes (full instruction length)
        if (!MmIsAddressValid(i) || !MmIsAddressValid(i + 6))
        {
            continue;
        }

        b1 = *(i);
        b2 = *(i + 1);
        b3 = *(i + 2);

        if (b1 == 0x4c && b2 == 0x8d && b3 == 0x15)
        {
            RtlCopyMemory(&offset, i + 3, 4);

            //fffff800`03e8b772 4c 8d 15 c7 20 23 00 4c-8d 1d 00 21 23 00 f7 83  L... #.L...!#...
```

```
            //templong = 002320c7, i = 03e8b772, 7 is the instruction length
            addr = (PSERVICE_DESCRIPTOR_TABLE_ENTRY)(offset + (ULONG_PTR)i + 7);
            break;
        }
    }

    DBG_INFO("[+] SdtGetSSDTBaseAddr offset: 0x%lx", offset);
    DBG_INFO("[+] SdtGetSSDTBaseAddr addr: 0x%p", addr);

    return addr;
}
```

The difference is straightforward: on older builds there is no intermediate `KiSystemCall64Shadow` indirection, so Stage 1 (the LFENCE/ `MOV GS` pattern scan) is skipped entirely. The driver tries `SdtGetSSDTBaseAddr` first; if that returns `NULL`, it falls back to `SdtGetSSDTBaseAddrOld`. The search window is slightly wider (0x500 bytes vs. 0x400) to accommodate the larger code distance on older kernels. Neither function uses hardcoded offsets or build-specific tables: both rely exclusively on byte-pattern scanning from the LSTAR MSR, so the driver works correctly across the entire Windows 10/11 version matrix.

Syscall indices are resolved dynamically at driver load time. We read `ntdll.dll` from disk, parse its PE export table, and extract the `MOV EAX, imm32` value from each syscall stub (the immediate operand is the syscall number). This makes the driver compatible across Windows versions without hardcoded index tables that would require maintenance on every new build. The resolution logic walks `ntdll.dll` 's export directory, resolves each target function by name, converts its address to an RVA, then scans the first 32 bytes of the stub for the `0xB8` (`MOV EAX, imm32`) opcode.

```
 ntdll!NtCreateFile:
  4C 8B D1          mov r10, rcx
  B8 52 00 00 00    mov eax, 52h    ; <- syscall index
  0F 05             syscall
  C3                ret
```

```
 /*
 * Searches for syscall index in the assembly code of a syscall stub by analyzing
 * the function's prologue for the standard Windows x64 syscall pattern. This function
 * serves as a critical component in SSDT hook implementation by extracting the
 * system call index from the 'mov eax, XX' instruction present in ntdll.dll stubs.
*/
_Must_inspect_result_
static ULONG SdtSearchSyscallIndexFromBinaryPattern(
    CONST IN ULONGLONG Address,
    CONST IN ULONG AddressRVA,
    CONST IN ULONG Size)
{
    // Parameter validation
```

```
    if (!Address ||
        !Size ||
        AddressRVA >= Size)
    {
        return 0;
    }

    ULONG SyscallIndex = 0;

    // Search in the first 32 bytes, the binary pattern mov eax, XX - (0xB8 ?? ?? ?? ??)
    // to get the SSDT index for this syscall.
    for (ULONG i = 0; i < 32 && AddressRVA + i < Size; i++)
    {
        UCHAR CurrentByte = *(PUCHAR)((PUCHAR)Address + i);

        if (CurrentByte == 0xC2 || CurrentByte == 0xC3) // ret
        {
            DBG_INFO("[+] Found RET");
            break;
        }
        // 0xB8 is followed by a 4-byte immediate; guard against reading past the search window or the
        // mapped image. In practice unreachable (0xB8 always appears early in the stub) but retained for correc
        if (CurrentByte == 0xB8 && i + 4 < 32 && AddressRVA + i + 4 < Size) // mov eax, XX - (0xB8 ?? ?? ?? ??)
        {
            DBG_INFO("[+] Found 0xB8 opcode. At i: %lu. Retrieving syscall index...", i);
            SyscallIndex = *(PULONG)((PUCHAR)Address + i + 1);
            break;
        }
    }

    return SyscallIndex;
}
```

Once each syscall index is resolved, it is stored alongside the original and new SSDT offsets in the `g_SSDT_HooksInfo` array. This structure is the foundation for both self-protection (detecting tampered entries) and clean unhooking:

```
 /*
 * SSDT hook state tracking structure
 * Maintains information about active SSDT hooks for protection and restoration.
 * Used to verify hook integrity and restore original state when needed.
*/
typedef struct _SSDT_HOOKS_INFO {
    PCHAR SyscallName;          // Name of the hooked syscall
    ULONG SyscallId;            // SSDT index of the syscall
    ULONG OldOffsetToFunction;  // Original SSDT entry offset
```

```
    ULONG NewOffsetToFunction;  // Hooked syscall offset in SSDT
} SSDT_HOOKS_INFO, *PSSDT_HOOKS_INFO;

// Successful Hooks Information (For SSDT UnHook and Protection of our Hooks)
static SSDT_HOOKS_INFO g_SSDT_HooksInfo[ARRAYSIZE(g_Hooks)];
```

At hook installation time, after writing each new trampoline offset to the SSDT, the driver records the syscall name, index, original offset, and new offset in the corresponding `g_SSDT_HooksInfo` entry. The self-protection thread uses this stored state to detect and restore any tampered entries (described in detail in Section 8).

Code Caves and Trampoline Architecture

Writing our hook functions' addresses directly into SSDT entries is not viable: the SSDT format stores a **signed 32-bit relative offset** from `KiServiceTable`, not an absolute address. Our hook functions live in non-paged pool, which is typically far too distant from `KiServiceTable` for the offset to fit in 32 bits.

The solution is to place 12-byte trampolines in **code caves** (sequences of NOP `0x90` and INT3 `0xCC` padding bytes that the compiler leaves between functions in the kernel's `.text` section). Since both `KiServiceTable` and the `.text` section belong to `ntoskrnl.exe` 's address space, the relative offset from any cave to the table always fits in 32 bits. A typical Windows kernel build contains roughly 3,000 such caves which is more than enough for the hooks we need.

Each trampoline is a 12-byte absolute indirect jump: `MOV RAX, hook_address; JMP RAX`, patched at install time with the address of the actual hook function:

```
 // 12-byte trampoline shellcode for hook redirection
static UCHAR g_TrampolineOpcodes[] = {
    0x48, 0xB8,                                     // mov rax, imm64
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, // <- address placeholder - 8 bytes overwritten with actual
    0xFF, 0xE0 };                                   // jmp rax
```

Finding those caves requires scanning `ntoskrnl.exe` 's `.text` section at runtime. The driver locates the module base via `ZwQuerySystemInformation` with `SystemModuleInformation`, parses the PE section headers to find the `.text` section's virtual address and size, then scans for contiguous sequences of `0x90` (NOP) or `0xCC` (INT3) bytes of the required length:

```
/*
* Searches for a code cave within a specified memory range by identifying continuous
* sequences of NOP (0x90) or INT3 (0xCC) instructions. This function serves as a
* critical component in the SSDT hooking system by locating suitable spaces in
* executable memory for injecting trampoline code.
*
* Code cave identification:
*   * Searches for continuous sequences of:
*     - 0x90 (NOP instruction)
```

```
 *     - 0xCC (INT3/breakpoint instruction)
 *   * Must match exact size requirement
 *   * Average availability of ~3000 valid caves in typical kernel builds
*/
_IRQL_requires_(PASSIVE_LEVEL)
_Must_inspect_result_
static NTSTATUS SdtSearchCodeCave(
    CONST IN PUCHAR StartAddress,
    CONST IN PUCHAR EndAddress,
    CONST IN ULONG CodeCaveSize,
    OUT PVOID* CodeCave)
{
    // IRQL validation
    if (KeGetCurrentIrql() > PASSIVE_LEVEL)
    {
        return STATUS_INVALID_LEVEL;
    }

    // Parameter validation
    if (!StartAddress ||
        !EndAddress ||
        !CodeCave ||
        !CodeCaveSize ||
        EndAddress <= StartAddress ||
        !MmIsAddressValid(StartAddress) ||
        !MmIsAddressValid(EndAddress))
    {
        return STATUS_INVALID_PARAMETER;
    }

    for (ULONG i = 0, j = 0; StartAddress + CodeCaveSize + i <= EndAddress; i++)
    {
        UCHAR CurrentByte = *(PUCHAR)((PUCHAR)StartAddress + i);

        if (CurrentByte == 0x90 || CurrentByte == 0xCC)
        {
            // Check if we have found a CodeCave of the needed size CodeCaveSize
            if (++j == CodeCaveSize)
            {
                *CodeCave = (PVOID)((PUCHAR)StartAddress + i - CodeCaveSize + 1);
                return STATUS_SUCCESS;
            }
        }
        else
        {
            j = 0;
        }
```

```
    }

    return STATUS_UNSUCCESSFUL;
}
```

For each hook, a fresh cave is located by searching from the previous cave's address forward, ensuring no two trampolines share the same padding sequence. The sequential search approach works well given the density of NOP/INT3 padding in a typical kernel build. Roughly 3,000 caves are available in the `.text` section, far more than the number of hooks we install.

The SSDT entry encodes the offset to the trampoline in its upper 28 bits; the lower 4 bits encode the syscall's parameter count and must be preserved when modifying entries. When installing a hook we read the original entry, extract its lower nibble, and OR it into the new offset value pointing to our trampoline.

A key strength of the trampoline design is that each hook retains a pointer to the **original** syscall function (`OldNtXxx`). This means every hook can call through to the real implementation, passing original or modified parameters as needed, and intercept or modify the return value. Hooks are not dead-ends, they are interception points that can observe, modify inputs, call the real function, and then observe or modify outputs. This is what enables malware detonation: instead of blocking or failing a suspicious call, we allow it to succeed while curating the result.

A short representative subset of the hooks we install:

```
static HOOK g_Hooks[] = {
    "NtCreateFile", (PVOID)NewNtCreateFile, (PVOID)&OldNtCreateFile,
    "NtDelayExecution", (PVOID)NewNtDelayExecution, (PVOID)&OldNtDelayExecution,
    "NtDeviceIoControlFile", (PVOID)NewNtDeviceIoControlFile, (PVOID)&OldNtDeviceIoControlFile,
    "NtOpenProcess", (PVOID)NewNtOpenProcess, (PVOID)&OldNtOpenProcess,
    "NtProtectVirtualMemory", (PVOID)NewNtProtectVirtualMemory, (PVOID)&OldNtProtectVirtualMemory,
    "NtQuerySystemInformation", (PVOID)NewNtQuerySystemInformation, (PVOID)&OldNtQuerySystemInformation,
    "NtQueryVolumeInformationFile", (PVOID)NewNtQueryVolumeInformationFile, (PVOID)&OldNtQueryVolumeInformationF
    "NtRaiseException", (PVOID)NewNtRaiseException, (PVOID)&OldNtRaiseException,
    "NtTerminateProcess", (PVOID)NewNtTerminateProcess, (PVOID)&OldNtTerminateProcess,
    "NtTerminateThread", (PVOID)NewNtTerminateThread, (PVOID)&OldNtTerminateThread,
    // ... additional hooks for injection detection, APC monitoring, forensic capture, etc.
};
```

Each hook serves a specific purpose (e.g. anti-evasion, injection detection, forensic capture, or some combination). Having `OldNtXxx` function pointers means hooks can always call the original syscall and can pass through any parameters unchanged when the calling process is not monitored, adding zero overhead to unmonitored system activity. When a monitored process is involved, the hook intercepts and modifies inputs or outputs to serve the analysis goals.

`SdtWriteHooks` is the orchestration function that ties together all of the primitives described above into a complete SSDT hook installation pipeline. It accepts a pointer to `KiServiceTable` (the SSDT base), the mapped

`ntdll.dll` image and its size, and a start/end address range for code cave allocation within the kernel `.text` section. After IRQL and parameter validation, it iterates over the global `g_Hooks` array (the table of syscalls to intercept), and for each entry executes the following sequence:

1. **Resolve the syscall index**: `SdtGetSyscallIndexByName` parses the `ntdll.dll` stub to extract the `MOV EAX,imm32` syscall number for the target syscall name.
2. **Save the original address**: `SdtGetAddressFromSSDTById` decodes the current SSDT entry and stores the real syscall address in the hook descriptor's `OldAddress` pointer, making it available to hook handlers for optional call-through to the real implementation.
3. **Find a code cave**: `SdtSearchCodeCave` scans the kernel `.text` section for a region of contiguous NOP (`0x90`) or INT3 (`0xCC`) bytes of sufficient length to hold the trampoline. Each iteration advances the search base past the previously allocated cave to prevent collisions between hooks.
4. **Write the trampoline**: `SdtWriteTrampoline` patches the code cave with the absolute `MOV RAX, imm64 / JMP RAX` sequence redirecting execution to the hook handler.
5. **Calculate the new SSDT offset**: The SSDT stores relative offsets from `KiServiceTable` , not absolute addresses, and the lower 4 bits of each entry encode the syscall's parameter count. The new offset is computed as `(CodeCave - KiServiceTable) << 4` , with the original lower 4 bits OR'd back in to preserve the parameter count metadata intact.
6. **Patch the SSDT entry**: Temporarily clear the WP bit in CR0 to write the new offset into `KiServiceTable[SyscallId]` , redirecting that syscall through the trampoline and into the hook handler.
7. **Record hook state**: On success, the original and new SSDT offsets, syscall name, and index are committed to `g_SSDT_HooksInfo` . This state is used later by the integrity-monitoring component to detect and restore any tampering with the installed hooks.

At this point, the full SSDT hooking pipeline is complete: locate the table dynamically via pattern scanning from the LSTAR MSR, resolve each syscall's index by parsing the `MOV EAX, imm32` stub from `ntdll.dll` , find a code cave in the kernel's `.text` section, write a 12-byte absolute trampoline there, calculate the SSDT-relative offset to the cave (preserving the lower 4 bits that encode the parameter count), and finally patch that offset into the correct SSDT entry (temporarily disabling write protection in CR0 to do so).

One aspect that technically minded readers will naturally ask about: since we are directly patching `KiServiceTable` (a structure that PatchGuard actively monitors, responding to unauthorized modifications with a `CRITICAL_STRUCTURE_CORRUPTION` bugcheck), how does the system not crash? The answer is that our platform operates in a purpose-built analysis environment where kernel integrity constraints are appropriately managed for this use case. This is an explicit architectural prerequisite of the deployment model.

Active Detonation: Answering Every Probe.

The Paradigm Shift: Both Hiding and Cooperating

Most sandbox-hardening approaches we've seen take a purely defensive posture: patch VMware registry keys, rename analysis tool processes, remove telltale files. This is fundamentally reactive: a growing blocklist that malware authors can enumerate, verify, and adapt to. Every static patch becomes a new detection signal once it is known.

Our driver takes both approaches simultaneously. On one side, it hides the analysis infrastructure: monitoring processes are invisible to malware's enumeration attempts, analysis-related files and registry keys are concealed or spoofed through the filesystem minifilter and registry callback, and the operating environment presents as a real physical workstation. On the other side, it *cooperates* with malware's probes rather than merely blocking them. For every environment check that malware performs routes through a syscall, we intercept it and return whatever answer encourages detonation. Malware can probe as aggressively as it wants. Every probe goes through our hooks. We control the response.

The combination is more powerful than either technique alone: not only do we prevent malware from detecting the analysis environment, but we actively assist malware in reaching its payload by capturing syscalls, spoofing return values to look real, and allowing malware to proceed through its execution chain. The filesystem minifilter and registry callback hide and spoof values at the filesystem and registry level. SSDT hooks do the same at the syscall level. The practical consequence: malware detonates. It sees a hardware environment that looks like a real physical workstation, concludes it is not being analyzed, and executes its payload. We capture everything.

Hardware Spoofing, Time Manipulation, and Process Hiding

Malware applies many categories of environment checks simultaneously. Effective detonation assistance requires addressing all of them coherently as a mismatch between two sources is itself a detection signal for sophisticated samples.

Time manipulation remains one of the most reliable evasion techniques we encounter in the wild. A call to `NtDelayExecution` (backing `Sleep()`) with an interval of minutes or hours, betting that the analysis environment has a fixed time window, will cause unsophisticated sandboxes to time out before the payload runs. Our hook on `NtDelayExecution` caps sleep intervals at a configurable maximum, but advances system time and the tick count by the amount skipped. This way, malware waking up perceives the correct elapsed time.

Disk size is queried via `NtDeviceIoControlFile` with multiple control codes (e.g. `IOCTL_DISK_GET_LENGTH_INFO`, `IOCTL_DISK_GET_DRIVE_GEOMETRY`, and `IOCTL_DISK_GET_DRIVE_GEOMETRY_EX`). Our hook on `NtDeviceIoControlFile` calls the real function first, then modifies the output buffer for monitored processes (e.g. spoofing `Length.QuadPart`, `Cylinders.QuadPart`, and `DiskSize.QuadPart`) to reflect a realistic physical machine. Similarly, `NtQueryVolumeInformationFile` is hooked to spoof `FileFsFullSizeInformation` and `FileFsSizeInformation`, adjusting `TotalAllocationUnits` and free-space figures consistently.

Processor count requires several independent sources to be spoofed consistently, because sophisticated samples cross-check them against each other:

1. `KUSER_SHARED_DATA`: The kernel structure mapped at fixed address `0xFFFFF78000000000` in kernel space (and `0x7FFE0000` in user space as a read-only mapping), is modified directly from kernel mode.
2. **Registry** (`HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\`): intercepted by the registry callback, which replaces the value data inside each processor subkey (vendor identifier, CPU name, and related fields) to reflect a physical machine.
3. `NtQuerySystemInformation` with `SystemBasicInformation`, `SystemEmulationBasicInformation`, and `SystemNativeBasicInformation` (intercepted by the SSDT hook), which also spoofs physical memory

figures across `SystemMemoryUsageInformation`, `SystemPerformanceInformation`, and `SystemBasicPerformanceInformation`.

These are some of the checks that should be taken into consideration when building a comprehensive detonation environment and the list evolves as malware authors discover new fingerprinting vectors.

KUSER_SHARED_DATA modifications deserve a closer look because they are more nuanced than a simple write. `KUSER_SHARED_DATA` uses a seqlock-style update protocol for fields like `TickCount` to allow non-blocking reads: Writers first update a "high" version field `High1Time`, then write the main value `LowPart`, and finally copy the high field again (`High2Time` = `High1Time`). Readers check that `High1Time` == `High2Time` before and after reading `LowPart`; if the values differ, a write was in progress and the read is retried. This pattern ensures readers either see a fully consistent value or retry, without blocking writers, and relies on atomic 32-bit reads/writes to detect in-progress updates.

```
 NTSTATUS DsIncreaseTickCount(CONST IN LARGE_INTEGER TimeSkipped)
{
    NTSTATUS ret = STATUS_SUCCESS;

    LARGE_INTEGER Ticks = DsMilliseconds2Ticks(TimeSkipped);

    ULONG SpoofedTickCount_Low = g_UserSharedData->TickCount.LowPart + Ticks.LowPart;
    LONG SpoofedTickCount_High = g_UserSharedData->TickCount.High1Time + Ticks.HighPart;

    // Write in correct seqlock order: High1Time, LowPart, High2Time
    g_UserSharedData->TickCount.High1Time = SpoofedTickCount_High;
    g_UserSharedData->TickCount.LowPart = SpoofedTickCount_Low;
    g_UserSharedData->TickCount.High2Time = SpoofedTickCount_High;

    return ret;
}
```

The `DsMilliseconds2Ticks` helper converts a millisecond interval to tick units. The Windows system timer runs at 15.625 ms, but this cannot be represented directly with floating-point math in kernel mode. While CPUs can execute floating-point instructions in ring 0, the Windows kernel does not automatically save or restore the FPU/SSE registers on context switches. Any floating-point instruction in kernel code that runs without explicitly saving and restoring the state (via `KeSaveExtendedProcessorState` / `KeRestoreExtendedProcessorState`) can corrupt the FPU state of an interrupted user-mode thread (a subtle, hard-to-reproduce form of memory corruption).

To avoid this entirely, we use an integer multiplication/division trick to safely approximate tick conversion. Since the system timer runs at 15.625 ms, we represent this period as the integer ratio 125/8 and perform all tick-count arithmetic using integer multiplication and division. No floating-point instructions are needed, and truncation of fractions is acceptable given the timer's granularity.

```
 // 15.625ms = 125/8 - integer representation avoids floating-point in kernel mode
#define TIMER_TICKS_PER_PERIOD      125     // Numerator of system timer period (15.625ms)
```

```
#define PERIODS_PER_INTERVAL        8       // Denominator of system timer period

// Clear calculation without floating point:
// Instead of: value / 15.625
// We use:     (value * PERIODS_PER_INTERVAL) / TIMER_TICKS_PER_PERIOD
Ticks.QuadPart = (TimeSkipped.QuadPart * PERIODS_PER_INTERVAL) / TIMER_TICKS_PER_PERIOD;
```

KUSER_SHARED_DATA is mapped at two fixed virtual addresses: read-only at 0x7FFE0000 in user mode, and writable at 0xFFFFF78000000000 in kernel mode, both established at boot. Unlike write-protected structures such as the SSDT, the kernel-mode mapping of KUSER_SHARED_DATA is already writable (no need for CR0.WP manipulation here). Direct writes through SharedUserData are the correct approach.

```
 // Spoof USER_SHARED_DATA NumberOfPhysicalPages - direct write
g_UserSharedData->NumberOfPhysicalPages += SPOOFED_RAM_ADDITIONAL_PHYSICAL_PAGES;
```

System uptime spoofing is a wrapper around tick count advancement: `DsSpoofSystemUptime` calls `DsSpoofTickCount`, which invokes `DsIncreaseTickCount` with a preconfigured startup time offset. Together these modifications ensure that `GetTickCount` and direct `KUSER_SHARED_DATA` reads reflect a machine that has been running for a plausible amount of time.

Note that writing large jumps to TickCount will cause a brief black screen flash, as the Desktop Window Manager uses it for frame scheduling and loses sync when the value changes abruptly. In practice, a brief mouse movement is sufficient to restore normal rendering.

Process Hiding is also an critical capability as it allows to hide security or specific analysis tools that we want to run in the same machine that detonates malware while we want to keep them safe from detection.

Process enumeration via `NtQuerySystemInformation` with `SystemProcessInformation`, `SystemSessionProcessInformation`, or `SystemExtendedProcessInformation` is filtered by our hook, which walks the returned process list and removes entries for hidden processes before the result reaches the caller. Here is a representative excerpt showing the `SystemBasicInformation` spoofing path (the first check in a hook that handles multiple information classes):

```
 NTSTATUS NTAPI NewNtQuerySystemInformation(
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT OPTIONAL PULONG ReturnLength)
{
    GrdIncThreadsIntoHooks();

    ULONG ProcessId = PslGetCurrentThreadProcessId();

    // Call the original syscall first. Let it populate the buffer
    NTSTATUS NtStatus = OldNtQuerySystemInformation(
```

```
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength,
        ReturnLength);

    if (NT_SUCCESS(NtStatus) && PslIsProcessMonitored(ProcessId) && ExGetPreviousMode() != KernelMode)
    {
        if ((SystemInformationClass == SystemBasicInformation ||
            SystemInformationClass == SystemEmulationBasicInformation ||
            SystemInformationClass == SystemNativeBasicInformation) &&
            SystemInformation &&
            SystemInformationLength >= sizeof(SYSTEM_BASIC_INFORMATION))
        {
            SYSTEM_BASIC_INFORMATION* sbi = (SYSTEM_BASIC_INFORMATION*)SystemInformation;
            __try
            {
                ProbeForWrite(sbi, sizeof(SYSTEM_BASIC_INFORMATION), 1);
                sbi->NumberOfProcessors = (CCHAR)SPOOFED_NUMBER_OF_PROCESSORS;
                sbi->NumberOfPhysicalPages += SPOOFED_RAM_ADDITIONAL_PHYSICAL_PAGES;
            }
            __except (EXCEPTION_EXECUTE_HANDLER)
            {
                DBG_ERROR("[!] Unable to spoof NumberOfProcessors at NewNtQuerySystemInformation");
            }
        }
        // ... additional information class handlers follow (SystemProcessInformation,
        //     SystemMemoryUsageInformation, SystemPerformanceInformation, etc.)
    }

    GrdDecThreadsIntoHooks();
    return NtStatus;
}
```

As we can see, the pattern is consistent across all handlers: call the real function first, then selectively modify the output buffer for monitored processes only. The `__try/__except` block around each `ProbeForWrite` + modification sequence is essential (user-mode callers can provide buffers that become invalid between the original call and our modification), and an unhandled access violation in kernel mode is a system crash.

Process hiding can be implemented at two distinct levels, and both have trade-offs. The first approach is based on unlinking a process from the `EPROCESS.ActiveProcessLinks` doubly-linked list (this is a well-known rootkit technique that makes the process invisible to any code walking that list). The second approach consists on filtering the output of `NtQuerySystemInformation` via SSDT hook (this is more surgical as it provides per-caller control). The hook can hide processes from monitored (malware) processes while returning accurate information to everything else on the system.

But filtering `NtQuerySystemInformation` alone is not sufficient. Malware that suspects its process-enumeration results are being filtered may fall back to PID bruteforcing: iterating over integer values and calling `NtOpenProcess` for each one, checking whether a handle is returned for later inspection. Our `NtOpenProcess` hook closes this gap: when a monitored process attempts to open a handle to a hidden PID, the hook returns `STATUS_INVALID_CID` without calling the real function, as if that PID does not exist:

```
 NTSTATUS NTAPI NewNtOpenProcess(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN OPTIONAL PCLIENT_ID ClientId)
{
    GrdIncThreadsIntoHooks();

    NTSTATUS NtStatus;
    ULONG ProcessId = PslGetCurrentThreadProcessId();

    if (PslIsProcessMonitored(ProcessId) && ExGetPreviousMode() != KernelMode)
    {
        if (ClientId)
        {
            __try
            {
                ProbeForRead(ClientId, sizeof(CLIENT_ID), sizeof(ULONG_PTR));

                if (ClientId->UniqueProcess)
                {
                    ULONG TargetPid = (ULONG)(ULONG_PTR)ClientId->UniqueProcess;

                    if (PslIsProcessHidden(TargetPid))
                    {
                        DBG_INFO("[+] Hiding process %lu from process %lu.", TargetPid, ProcessId);

                        // Pretend the process does not exist
                        NtStatus = STATUS_INVALID_CID;
                        goto ignore_original_and_exit;
                    }
                }
            }
            __except (EXCEPTION_EXECUTE_HANDLER)
            {
                NtStatus = STATUS_ACCESS_VIOLATION;
                goto ignore_original_and_exit;
            }
        }
    }
```

```
    // Call the original syscall
    NtStatus = OldNtOpenProcess(ProcessHandle, DesiredAccess, ObjectAttributes, ClientId);

ignore_original_and_exit:
    GrdDecThreadsIntoHooks();
    return NtStatus;
}
```

The combination of `NtQuerySystemInformation` filtering and `NtOpenProcess` interception means that monitoring processes are hidden from both enumeration-based discovery and PID-bruteforce discovery (the two primary methods malware uses to locate and interact with analysis tools). Covering all these vectors eliminates the inconsistencies that sophisticated samples exploit. If even one source returns the real value, a capable sample will notice and abort detonation.

Sidebar: The `ExGetPreviousMode() != KernelMode` Guard

Every SSDT hook applies its interception logic only when `ExGetPreviousMode()!= KernelMode`. This is a critical safety invariant, not an optimization. `ExGetPreviousMode()` returns `UserMode` when the thread entered via a `SYSCALL` from user space, and `KernelMode` when the call originated from another kernel component.

Without this guard, hooks would fire on kernel-mode callers too. Many kernel subsystems call `NtQuerySystemInformation`, NtCreateFile, and similar functions internally and spoofing those return values corrupts kernel state and causes hard-to-diagnose deferred crashes. With the guard, interception applies only to user-mode syscalls. As a side effect, since the vast majority of calls through hooked functions originate from the kernel itself, the guard also eliminates overhead for most invocations.

Following the Full Infection Chain.

Why Persistence Detection Changes the Game

This is one of the capabilities we consider most important in the driver, and requires dedicated kernel-level persistence awareness to address effectively.

Modern multi-stage malware follows a consistent pattern: stage one (a loader or dropper) arrives via phishing or exploitation, performs environment checks, and if it decides the environment is safe, establishes persistence. This persistence mechanism is designed to trigger stage two when the system “reboots” or when a user logs in. Stage two is the actual payload: the information stealer, the ransomware core, the banking trojan, the RAT.

Some sandboxes analyze stage one. They watch it drop a file and write a Run key, and then conclude the analysis. Stage two is never executed because there is no simulated reboot, no triggering of the persistence mechanism. The analyst knows persistence was established but never sees what it deploys.

If you cannot see the final payload, you cannot build detection for it. If you cannot build detection for it, it lands in your environment undetected.

Our driver closes this gap.

Filesystem Minifilter and Registry Callback: Monitoring for Persistence

Our filesystem minifilter and registry callback both maintain real-time awareness of persistence-related activity from monitored processes.

The **registry callback** (registered via `CmRegisterCallbackEx`) intercepts all registry operations system-wide. When a monitored process writes to persistence-related registry locations (e.g. `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`, RunOnce or service registration paths), the filter captures the written value and immediately records and dispatches it for tracking. The write operation completes normally; the malware's persistence mechanism is established as intended. We simply observe it, in real time, before any cleanup can occur.

The `CmRegisterCallbackEx` API delivers every registry operation to a single callback function with an operation type parameter (`REG_NOTIFY_CLASS`). Rather than a large `switch` statement, the implementation uses a dispatch table (an array of function pointers indexed by operation type). This keeps the main callback function minimal and makes adding or removing handlers for specific operation types straightforward:

```
 // Dispatch table: one entry per REG_NOTIFY_CLASS value
static PEX_CALLBACK_FUNCTION g_RegistryCallbackTable[MaxRegNtNotifyClass] = { 0 };

// Main callback - routes to the appropriate handler
static NTSTATUS RfRegistryCallback(
    IN PVOID CallbackContext,
    IN PVOID Argument1,
    IN PVOID Argument2)
{
    REG_NOTIFY_CLASS Operation = (REG_NOTIFY_CLASS)(ULONG_PTR)Argument1;

    // Defensive bounds check: future Windows versions may introduce REG_NOTIFY_CLASS values
    // beyond the SDK-defined MaxRegNtNotifyClass. Always guard before indexing kernel tables.
    if (Operation >= MaxRegNtNotifyClass || !g_RegistryCallbackTable[Operation])
        return STATUS_SUCCESS;

    return g_RegistryCallbackTable[Operation](CallbackContext, Argument1, Argument2);
}
```

The table is populated at initialization with handlers for each operation we care about:

```
 static VOID RfInitRegistryCallbackTable()
{
    // Access Control: Key Creation and Opening (hide VM-related keys)
    g_RegistryCallbackTable[RegNtPreOpenKeyEx]    = (PEX_CALLBACK_FUNCTION)RfPreOpenCreateKeyEx;
    g_RegistryCallbackTable[RegNtPreCreateKeyEx]  = (PEX_CALLBACK_FUNCTION)RfPreOpenCreateKeyEx;
```

```
    // Access Control: Key Enumeration
    g_RegistryCallbackTable[RegNtQueryKey]         = (PEX_CALLBACK_FUNCTION)RfQueryKey;
    g_RegistryCallbackTable[RegNtPreEnumerateKey]  = (PEX_CALLBACK_FUNCTION)RfPreEnumerateKey;
    g_RegistryCallbackTable[RegNtEnumerateKey]     = (PEX_CALLBACK_FUNCTION)RfEnumerateKey;

    // Access Control: Deletion Protection
    g_RegistryCallbackTable[RegNtPreDeleteKey]      = (PEX_CALLBACK_FUNCTION)RfPreDeleteKey;
    g_RegistryCallbackTable[RegNtPreDeleteValueKey] = (PEX_CALLBACK_FUNCTION)RfPreDeleteValueKey;

    // Monitoring: Persistence Detection (captures Run key writes, service registrations)
    g_RegistryCallbackTable[RegNtPreSetValueKey]    = (PEX_CALLBACK_FUNCTION)RfPreSetValueKey;

    // Monitoring: Value Spoofing (hardware identifiers, CPU info, etc.)
    g_RegistryCallbackTable[RegNtPostQueryValueKey] = (PEX_CALLBACK_FUNCTION)RfPostQueryValueKey;
}
```

Any `REG_NOTIFY_CLASS` value with no registered handler returns `STATUS_SUCCESS` immediately, resulting in zero overhead for operations we do not need to intercept.

Protecting Persistence Entries from Deletion

The `RegNtPreDeleteKey` and `RegNtPreDeleteValueKey` entries serve a different purpose than the hiding handlers: they protect persistence-related registry keys and values from being deleted by monitored processes.

This matters because sophisticated malware sometimes probes the registry not only by opening keys but by attempting to delete them. If a deletion of a VM-artifact key succeeds, that is itself confirmation that the environment is real and that no defensive driver is intercepting registry operations. By blocking deletion attempts on the same set of keys that `RfPreOpenCreateKeyEx` hides, the driver presents a consistent picture regardless of how the malware interrogates the registry and eliminates this side-channel.

Hiding VM-Revealing Registry Keys

The `RegNtPreOpenKeyEx` and `RegNtPreCreateKeyEx` entries both route to `RfPreOpenCreateKeyEx`. This handler is responsible for making virtualization-related registry keys appear nonexistent to monitored processes. Hypervisors and virtualization platforms leave characteristic footprints in the registry (VMware stores identifiers under `HKLM\SOFTWARE\VMware, Inc.\VMware Tools`, VirtualBox leaves traces under `HKLM\HARDWARE\ACPI\DSDT\VBOX__`, and the HARDWARE enumeration tree contains device strings that reveal the underlying platform). Any monitored process attempting to open or create these keys should receive `STATUS_OBJECT_NAME_NOT_FOUND` as if the key simply does not exist.

```
/*
* Pre-operation callback for registry key open/create operations.
* Blocks access to hidden registry keys.
*
```

```
 * Parameters:
 *  CallbackContext - Context pointer supplied during registration (unused)
 *  Argument1 - Registry operation type (unused)
 *  CallbackData - Information about the key being opened/created
 *
 * Returns:
 *  NTSTATUS:
 *   - STATUS_SUCCESS - Allow the operation
 *   - STATUS_OBJECT_NAME_NOT_FOUND - Block access to hidden key
 *   - Other error codes from RfBuildCompleteRegistryKeyString
 *
 * Notes:
 *   - Must be called at PASSIVE_LEVEL due to memory operations
 *     * This callback is always invoked at IRQL = PASSIVE_LEVEL (guaranteed by the registry filter manager).
 *   - Only processes requests from monitored processes
 *   - Attempts to build complete path, falls back to relative path if needed
*/
_IRQL_requires_(PASSIVE_LEVEL)
_IRQL_requires_same_
_Function_class_(EX_CALLBACK_FUNCTION)
static NTSTATUS RfPreOpenCreateKeyEx(
    IN OPTIONAL PVOID CallbackContext,
    IN OPTIONAL PVOID Argument1,
    IN PREG_OPEN_KEY_INFORMATION CallbackData)
{
    UNREFERENCED_PARAMETER(CallbackContext);
    UNREFERENCED_PARAMETER(Argument1);

    NTSTATUS NtStatus = STATUS_SUCCESS;
    PUNICODE_STRING KeyNameBeingOpened = NULL;
    PUNICODE_STRING LocalCompleteName = NULL;
    ULONG ProcessId = PslGetCurrentThreadProcessId();

    // Parameter validation
    if (!CallbackData || !CallbackData->CompleteName)
    {
        DBG_ERROR("[!] Invalid Callback data at RfPreOpenCreateKeyEx");
        return STATUS_INVALID_PARAMETER;
    }

    // We do not care if the request comes from a non-monitored process or from Kernel
    if (!PslIsProcessMonitored(ProcessId) || ExGetPreviousMode() == KernelMode)
    {
        return STATUS_SUCCESS;
    }

    // Try to build complete path
```

```
    if (!NT_SUCCESS(RfBuildCompleteRegistryKeyString(CallbackData, &LocalCompleteName)))
    {
        KeyNameBeingOpened = CallbackData->CompleteName;
    }
    else
    {
        KeyNameBeingOpened = LocalCompleteName;
    }

    // HIDE
    if (RfIsHide(KeyNameBeingOpened))
    {
        DBG_INFOW("[+] Detected HIDDEN registry at RfPreOpenCreateKeyEx: %wZ", KeyNameBeingOpened);
        NtStatus = STATUS_OBJECT_NAME_NOT_FOUND;
    }

    if (LocalCompleteName)
    {
        ExFreePool(LocalCompleteName);
        LocalCompleteName = NULL;
    }

    return NtStatus;
}
```

The handler extracts the requested key path from the `REG_OPEN_CREATE_KEY_EX_INFORMATION` structure provided in `Argument2`, then checks it against a curated list of known VM-revealing path prefixes. When a monitored process attempts to open a matching key, the handler replaces the result with `STATUS_OBJECT_NAME_NOT_FOUND` before the operation reaches the registry stack.

The **filesystem minifilter** (operating through the Windows Filter Manager `fltmgr.sys`) intercepts `IRP_MJ_CREATE` and `IRP_MJ_SET_INFORMATION` operations. When a monitored process creates a file in a persistence-relevant location (e.g. startup folders) the minifilter captures the file path and records it. When a file is renamed or moved, the minifilter captures both the old and new paths via `FileRenameInformation` and `FileRenameInformationEx` callbacks.

Beyond persistence detection, the same minifilter implements **file hiding** for analysis infrastructure. A configuration table lists filenames and path fragments that should be invisible to monitored processes. For each `IRP_MJ_CREATE` operation from a monitored process, the pre-operation callback checks the requested path against this table. On a match, it sets `Data->IoStatus.Status` to `STATUS_OBJECT_NAME_NOT_FOUND` and returns `FLT_PREOP_COMPLETE`, short-circuiting the rest of the filter stack and returning the synthesized failure directly to the caller (as if the file simply does not exist). This same path also suppresses `DeleteOnClose` attempts against hidden files. Note that hiding files from direct open attempts (`IRP_MJ_CREATE`) is distinct from hiding them from directory listings, which would require scrubbing entries in an `IRP_MJ_DIRECTORY_CONTROL` post-operation

callback, which is a complementary layer that keeps hidden items absent from both direct opens and directory enumeration.

Both filters operate entirely transparently from the malware's perspective: there is no blocking, no modification of the malware's behavior. The malware succeeds in establishing persistence. We know about it the moment it happens.

Triggering Forced Execution of Next-Stage Payloads

When a persistence event is captured, a trusted user-space component spawns the registered payload in the appropriate execution context.

This transforms the analysis outcome from "we observed persistence establishment" to "we observed the complete infection chain, including the final payload's behavior." For multi-stage loaders that only deploy the final payload after confirming successful persistence, this capability is what makes full behavioral capture possible: within a single analysis session, without manual analyst intervention and without waiting for a real reboot cycle.

Automatic Injection Chain Propagation

Modern malware rarely operates in a single process. Loaders inject into legitimate system processes. Injected code spawns further processes. Without automatic tracking across process boundaries, monitoring the initial sample captures only a fraction of the malicious activity.

Our driver handles this automatically through two mechanisms: Kernel process creation callbacks and SSDT hooks on injection-related syscalls like `NtQueueApcThread` (and its Windows 10 variants `NtQueueApcThreadEx` and `NtQueueApcThreadEx2`) or `NtSetContextThread`. For instance, when a monitored process successfully queues an APC to a thread in another process, the hook calls `PslFollowThreadInjection` to automatically add the target process to the monitored list. `NtSetContextThread` and `NtProtectVirtualMemory` (when the latter adds executable permissions to a region in a different process) similarly trigger injection tracking. The result: analysts see the complete behavioral graph of the infection, from the initial sample through every spawned child and injection target, without any manual configuration of additional PIDs to monitor.

`PslFollowThreadInjection` is the function all injection-detecting hooks call once they identify a cross-process operation. It resolves the owning process of the target thread, confirms the target differs from the injector (rejecting same-process cases) and calls `PslThreadInjectionFollowProcess` to add the target to the monitored list and notify the monitoring driver. It is only called after a successful cross-process syscall, which is sufficient evidence of injection intent. `NtCreateThreadEx` needs no special handling here since remote thread creation already triggers the kernel thread-creation callback; it is only indirect techniques (e.g. APC queuing, thread hijacking, context redirection) that require this path.

The injection detection side, happens in the individual syscall hooks before they invoke `PslFollowThreadInjection`. APC injection can reach a target thread through three syscall variants: `NtQueueApcThread`, and the extended `NtQueueApcThreadEx` and `NtQueueApcThreadEx2` (introduced in Windows 10 with additional parameters that enable more sophisticated APC queuing control). All three variants receive their own SSDT hooks with identical detection logic. The base implementation for `NtQueueApcThread`:

```
 /*
 * SSDT hook for NtQueueApcThread that monitors APC-based thread injection techniques
 * for tracked processes. Detects when monitored processes queue APCs to other threads,
 * which is commonly used for code injection and process hollowing attacks.
 *
 * Since the original function is called before hook logic, parameter validation, buffer
 * alignment, and structure integrity are assumed to be handled by the original NtQueueApcThread
 * implementation.
*/
NTSTATUS NTAPI NewNtQueueApcThread(
    IN HANDLE ThreadHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN OPTIONAL PVOID ApcArgument1,
    IN OPTIONAL PVOID ApcArgument2,
    IN OPTIONAL PVOID ApcArgument3)
{
    GrdIncThreadsIntoHooks();

    // Call the original syscall
    NTSTATUS NtStatus = OldNtQueueApcThread(
        ThreadHandle,
        ApcRoutine,
        ApcArgument1,
        ApcArgument2,
        ApcArgument3);

    if (NT_SUCCESS(NtStatus))
    {
        ULONG ProcessId = PslGetCurrentThreadProcessId();

        if (PslIsProcessMonitored(ProcessId) && ExGetPreviousMode() != KernelMode)
        {
            // We can bypass PslAdd validity check as we know the process exists (we are doing this
            // after a successful call to the original syscall).
            NTSTATUS ret = PslFollowThreadInjection(ThreadHandle, ProcessId, 0, TRUE);
            if (!NT_SUCCESS(ret))
            {
                // Ignore error if TargetPid and InjectorPid is the same (not an actual thread injection)
                if (ret != STATUS_NOT_SUPPORTED)
                {
                    DBG_ERROR("[!] Unable to PslFollowThreadInjection for PID: %lu at NewNtQueueApcThread. NTSTA
                        ProcessId,
                        ret);
                    LgPrintfW(ERROR, L"[!] Unable to PslFollowThreadInjection for PID: %lu at NewNtQueueApcThrea
                        ProcessId,
                        ret);
```

```
                }
            }
            else
            {
                DBG_INFO("[+] Successfully followed thread injection at NewNtQueueApcThread. Injector PID: %lu",
                LgPrintfW(INFO, L"[+] Successfully followed thread injection at NewNtQueueApcThread. Injector PI
            }
        }
    }

    GrdDecThreadsIntoHooks();

    return NtStatus;
}
```

`NtSetContextThread` injection (commonly used to redirect execution in a thread belonging to another process by overwriting its register context) follows the same detection pattern. Our hook intercepts the context-modification step at the syscall boundary: if a monitored process calls `NtSetContextThread` on a thread belonging to a different process, `PslFollowThreadInjection` is called to add the target to the monitored list.

```
 /*
 * SSDT hook for NtSetContextThread that detects thread context manipulation in monitored
 * processes, a common technique used in code injection attacks (thread hijacking).
 *
 * The hook calls the original syscall first to ensure parameter validation and successful
 * execution. Only after success does it track the potential thread injection via
 * PslFollowThreadInjection, which monitors cross-process thread manipulation patterns.
*/
NTSTATUS NTAPI NewNtSetContextThread(
    IN HANDLE ThreadHandle,
    IN PCONTEXT ThreadContext)
{
    GrdIncThreadsIntoHooks();

    // Call the original syscall
    NTSTATUS NtStatus = OldNtSetContextThread(ThreadHandle, ThreadContext);

    if (NT_SUCCESS(NtStatus))
    {
        ULONG ProcessId = PslGetCurrentThreadProcessId();

        if (PslIsProcessMonitored(ProcessId) && ExGetPreviousMode() != KernelMode)
        {
            /*
             * No need to validate ThreadHandle or target process here:
             * - NtSetContextThread already validated parameters and thread existence.
```

```
             * - NtStatus success guarantees ThreadHandle refers to a valid thread.
             *
             * We can also bypass PslAdd validity check as we know the process exists
             * (we are doing this after a successfull call to the original syscall).
            */
            NTSTATUS ret = PslFollowThreadInjection(ThreadHandle, ProcessId, 0, TRUE);
            if (!NT_SUCCESS(ret))
            {
                // Ignore error if TargetPid and InjectorPid is the same (not an actual thread injection)
                if (ret != STATUS_NOT_SUPPORTED)
                {
                    DBG_ERROR("[!] Unable to PslFollowThreadInjection for PID: %lu at NewNtSetContextThread. NTS
                        ProcessId,
                        ret);
                    LgPrintfW(ERROR, L"[!] Unable to PslFollowThreadInjection for PID: %lu at NewNtSetContextThr
                        ProcessId,
                        ret);
                }
            }
            else
            {
                DBG_INFO("[+] Successfully followed thread injection at NewNtSetContextThread. Injector PID: %lu
                LgPrintfW(INFO, L"[+] Successfully followed thread injection at NewNtSetContextThread. Injector
            }
        }
    }

    GrdDecThreadsIntoHooks();

    return NtStatus;
}
```

The structure is identical to the APC injection hooks: call the original first, check success, confirm the caller is a monitored user-mode process, then call `PslFollowThreadInjection`. The `STATUS_NOT_SUPPORTED` check suppresses the expected non-error case where the thread being modified belongs to the calling process itself, which would mean this is a self-context modification, not an injection.

The `NtProtectVirtualMemory` case covers the write-then-make-executable pattern. When a monitored process calls `NtProtectVirtualMemory` to add executable permissions to a region in a *different* process, that is a strong signal that code has been written there and is about to be invoked. The hook resolves the process handle in the call to determine the target, then calls `PslFollowThreadInjection` via the owning thread.

Together, the detection paths described in this section demonstrate how kernel-level monitoring enables injection detection and dynamic scope expansion of the monitored process set. When a monitored process performs operations associated with code injection (e.g. queuing APCs to foreign threads, redirecting thread contexts, or marking remote memory regions as executable), the target process is automatically added to the monitored list.

This covers the injection patterns described above, and in each case the injection target begins generating behavioral records from its first instruction in the new context, without any manual analyst configuration.

Forensic Preservation: Capturing What Malware Destroys

Beyond detonation assistance and infection chain tracking, the driver provides forensic capture capabilities that preserve evidence malware routinely destroys: evidence that is gone by the time analysts examine the system post-analysis.

Pre-Deletion File Dumping

Malware frequently drops payloads to disk as temporary files like decrypted shellcode, second-stage executables, configuration dat, etc. Uses them briefly, and then deletes them. Without intervention, these files are gone from the sandbox as surely as from a physical machine.

Files can be deleted through two main mechanisms in Windows: opening with `FILE_DELETE_ON_CLOSE` or `DELETE` access then closing the handle (signaled through `IRP_MJ_CLEANUP`), or calling `NtSetInformationFile` with `FileDispositionInformation` or `FileDispositionInformationEx` (signaled through `IRP_MJ_SET_INFORMATION`). Our filesystem minifilter handles both paths.

For the `FILE_DELETE_ON_CLOSE` path, the minifilter uses file contexts to track state across the two-phase create/cleanup lifecycle. The workflow is split between the `IRP_MJ_CREATE` pre-operation and post-operation callbacks. In the pre-operation callback, when `FILE_DELETE_ON_CLOSE` is detected in the create options, a `DELETE_ON_CLOSE_CONTEXT` structure is allocated and populated with the file path, then passed as a completion context to the post-operation callback. In `FsFltCreatePostOperation` (once the kernel has completed the `IRP_MJ_CREATE` request and the file is successfully open), the context is attached to the file object via `FltSetFileContext`. This binding persists until the cleanup callback releases it: when the file handle is closed and `IRP_MJ_CLEANUP` fires, the cleanup callback retrieves the attached context and dumps the file before deletion completes. The two-phase design is necessary because `FltSetFileContext` requires a successfully opened file object, which only exists after the create operation completes:

```
/*
* Post-operation callback for IRP_MJ_CREATE in the minifilter, used to finalize context setup for
* files opened with the FILE_DELETE_ON_CLOSE option.
*
* Parameters:
*   Data              - Pointer to the FLT_CALLBACK_DATA structure for the completed create/open operation.
*   FltObjects        - Pointer to the FLT_RELATED_OBJECTS structure for the current operation.
*   CompletionContext - Pointer to a DELETE_ON_CLOSE_CONTEXT structure, if allocated in the pre-operation
*                       callback.
*   Flags             - Post-operation flags (unused).
*
* Returns:
*   FLT_POSTOP_CALLBACK_STATUS:
*     - FLT_POSTOP_FINISHED_PROCESSING: Indicates that post-operation processing is complete.
```

```
 *
 * Notes:
 *   - Must be called at IRQL ≤ APC_LEVEL.
 *     * This is guaranteed by the Filter Manager for minifilter post-operation callbacks.
 *     * No explicit IRQL validation is required in this function.
 *   - The function only processes requests from monitored user-mode processes; requests from non-monitored
 *     processes or from kernel mode are ignored.
 *   - If a DELETE_ON_CLOSE_CONTEXT was provided (i.e., FILE_DELETE_ON_CLOSE was requested and context was
 *     allocated in the pre-operation callback), the function attempts to set this context on the file object
 *     using FltSetFileContext.
 *   - If FltSetFileContext fails, the function frees the context and any associated memory.
 *   - If FltSetFileContext succeeds, ownership of the context is transferred to the filter manager, which
 *     will free it when the file context is deleted (e.g. in the cleanup pre-operation callback).
 *   - If the create/open operation failed, the function frees the context and any associated memory.
 *   - The function assumes that the minifilter infrastructure is correctly managing IRQL and context lifetimes.
 */
_IRQL_requires_max_(APC_LEVEL)
_Function_class_(PFLT_POST_OPERATION_CALLBACK)
static FLT_POSTOP_CALLBACK_STATUS FsFltCreatePostOperation(
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN OPTIONAL PVOID CompletionContext,
    IN FLT_POST_OPERATION_FLAGS Flags
)
{
    UNREFERENCED_PARAMETER(Flags);

    ULONG ProcessId = PslGetCurrentThreadProcessId();

    // We do not care if the request comes from a non-monitored process or from Kernel
    if (!PslIsProcessMonitored(ProcessId) || ExGetPreviousMode() == KernelMode)
    {
        return FLT_POSTOP_FINISHED_PROCESSING;
    }

    PDELETE_ON_CLOSE_CONTEXT DeleteOnCloseContext = (PDELETE_ON_CLOSE_CONTEXT)CompletionContext;

    // Only proceed if we have a context (i.e. DELETE_ON_CLOSE was requested)
    if (DeleteOnCloseContext)
    {
        if (NT_SUCCESS(Data->IoStatus.Status))
        {
            // File was successfully opened, set the file context
            NTSTATUS NtStatus = FltSetFileContext(
                FltObjects->Instance,
                FltObjects->FileObject,
                FLT_SET_CONTEXT_KEEP_IF_EXISTS, // Don't overwrite if another context is already set
```

```
                DeleteOnCloseContext,
                NULL // Don't need the old context back
            );

            if (!NT_SUCCESS(NtStatus))
            {
                // Failed to set context (maybe another filter set one first), free our context
                DBG_ERROR("[!] Unable to FltSetFileContext at FsFltCreatePostOperation. NTSTATUS: 0x%X", NtStatu

                if (DeleteOnCloseContext->FileName.Buffer)
                {
                    ExFreePool(DeleteOnCloseContext->FileName.Buffer);
                }
                FltReleaseContext(DeleteOnCloseContext);
            }
            /* If successful, ownership of the context is now with the filter manager. In other words, if
             * FltSetFileContext succeeds, the filter manager owns the context and will free it when we
             * call FltDeleteFileContext in cleanup (at FsFltCleanupPreOperation)
            */
            else
            {
                DBG_INFO("[+] Successfully executed FltSetFileContext at FsFltCreatePostOperation");
            }
        }
        else
        {
            // Create/open failed, free our context
            if (DeleteOnCloseContext->FileName.Buffer)
            {
                ExFreePool(DeleteOnCloseContext->FileName.Buffer);
            }
            FltReleaseContext(DeleteOnCloseContext);
        }
    }

    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

For the `NtSetInformationFile` (`FileDispositionInformation`) path, the minifilter intercepts the `IRP_MJ_SET_INFORMATION` pre-callback. Because this callback may fire at `APC_LEVEL` in some scenarios (asynchronous I/O completions, thread exit cleanup), the code path is IRQL-aware: if we are already at `PASSIVE_LEVEL` , it opens a handle to the file directly in the callback to prevent deletion while dumping. If not at `PASSIVE_LEVEL` , a `PFLT_GENERIC_WORKITEM` is queued via `FltQueueGenericWorkItem` to perform the dump at `PASSIVE_LEVEL` in a deferred context (accepting the small risk that in rare cases the file may already be deleted by the time the work item runs).

This IRQL discipline is not optional. File I/O operations such as opening a handle or reading file contents require `PASSIVE_LEVEL`. Attempting them at `APC_LEVEL` does not generate a clean error; it causes a bugcheck (`IRQL_NOT_LESS_OR_EQUAL`) or silent memory corruption depending on which code path is entered. Without the `FltQueueGenericWorkItem` deferral path, a driver would crash the system on any deletion attempt that arrives above `PASSIVE_LEVEL` (exactly the scenario that occurs when malware opens a file for deletion in an APC context).

The dumped files are available for post-analysis: static analysis, signature matching, further behavioral analysis in a controlled environment. Files that the malware tried to erase become permanent evidence.

Memory Capture: Intercepting Payloads at Critical Stages

When a monitored process terminates, its address space (containing heap allocations, decrypted payloads, runtime data structures, and whatever the malware had loaded and unpacked in memory), is freed by the kernel. Without intervention, that memory state is permanently gone.

Beyond termination, the driver also captures memory at a critically important earlier moment: when a monitored process marks a memory region as executable. This is the write-then-execute pattern that is common across many process injection techniques (e.g. shellcode or a PE image is written into a target process's memory, and then `NtProtectVirtualMemory` is called to add executable permissions before triggering execution). Intercepting this permission change captures the target process at precisely the moment the injected payload is finalized and about to run in its fully decrypted, pre-execution state.

```
 /*
 * SSDT hook for NtProtectVirtualMemory that monitors memory protection changes with
 * execute permissions for tracked processes. Triggers process memory dumps when
 * monitored processes modify memory protections to include any execute flag
 * (PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY).
 *
 * Since the original function is called before hook logic, parameter validation, buffer
 * alignment, and structure integrity are assumed to be handled by the original
 * NtProtectVirtualMemory implementation.
*/
NTSTATUS NTAPI NewNtProtectVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID* BaseAddress,
    IN OUT PSIZE_T RegionSize,
    IN ULONG NewProtection,
    OUT PULONG OldProtection)
{
    GrdIncThreadsIntoHooks();

    // Call the original syscall
    NTSTATUS NtStatus = OldNtProtectVirtualMemory(
        ProcessHandle,
        BaseAddress,
```

```
    RegionSize,
    NewProtection,
    OldProtection);

if (NT_SUCCESS(NtStatus) && (NewProtection & (PAGE_EXECUTE | PAGE_EXECUTE_READ | PAGE_EXECUTE_READWRITE | PA
{
    ULONG CurrentProcessId = PslGetCurrentThreadProcessId();

    if (PslIsProcessMonitored(CurrentProcessId) && ExGetPreviousMode() != KernelMode)
    {
        ULONG TargetProcessId;
        PEPROCESS TargetProcess;

        // Handle current-process pseudo-handle (NtCurrentProcess() == -1)
        if (ProcessHandle == NtCurrentProcess())
        {
            TargetProcess = PsGetCurrentProcess();
            if (TargetProcess == NULL)
            {
                goto return_from_original_syscall;
            }

            /*
             * CRITICAL: PsGetCurrentProcess does NOT increment reference count.
             * We must manually reference it to safely use TargetProcess
             * during memory dumping. Don't forget to ObDereferenceObject afterwards.
            */
            ObReferenceObject(TargetProcess);

            TargetProcessId = (ULONG)(ULONG_PTR)PsGetProcessId(TargetProcess);
        }
        else
        {
            NTSTATUS InfoFromHandleStatus;

            /*
             * GetProcessInfoFromProcessHandle returns a referenced PEPROCESS.
             * Caller is responsible for calling ObDereferenceObject(TargetProcess)
             * once we're done with it.
             *
             * NOTE: This is consistent with the NtCurrentProcess() case, where we
             * manually reference PsGetCurrentProcess().
            */

            // Uses UserMode AccesMode as these handles come from user-mode.
            InfoFromHandleStatus = GetProcessInfoFromProcessHandle(ProcessHandle, UserMode, &TargetProcessId
```

```
            if (!NT_SUCCESS(InfoFromHandleStatus))
            {
                DBG_ERROR("[!] Unable to GetProcessInfoFromHandle. NTSTATUS: 0x%X", InfoFromHandleStatus);
                goto return_from_original_syscall;
            }
        }

        if (!TargetProcessId)
        {
            // This should not happen if we got to this point.
            // Checking this for consistency.
            // Release reference we took before finishing
            ObDereferenceObject(TargetProcess);
            goto return_from_original_syscall;

        }

        DBG_INFO("[+] NtProtectVirtualMemory Hook: Detected memory protection change (+EXECUTE) for monitore
            TargetProcessId,
            CurrentProcessId);
        LgPrintfW(INFO, L"[+] NtProtectVirtualMemory Hook: Detected memory protection change (+EXECUTE) for
            TargetProcessId,
            CurrentProcessId);

        // Prepare the context for the dump function.
        PROCESS_DUMP_CONTEXT DumpContext;
        DumpContext.TargetProcess = TargetProcess;
        DumpContext.TargetProcessId = TargetProcessId;

        NTSTATUS DumpStatus = DmpTryDumpProcessMemoryToFile(&DumpContext, FALSE);

        if (!NT_SUCCESS(DumpStatus))
        {
            DBG_ERROR("[!] Unable to Dump process memory for PID: %lu at NtProtectVirtualMemory. NTSTATUS: 0
                TargetProcessId,
                DumpStatus);
            LgPrintfW(ERROR, L"[!] Unable to Dump process memory for PID: %lu at NtProtectVirtualMemory. NTS
                TargetProcessId,
                DumpStatus);
        }
        else
        {
            DBG_INFO("[+] Successfully dumped process memory for PID: %lu at NtProtectVirtualMemory", Target
            LgPrintfW(INFO, L"[+] Successfully dumped process memory for PID: %lu at NtProtectVirtualMemory\
        }

        // CRITICAL: Before we proceed, we must release the reference we took.
```

```
            ObDereferenceObject(TargetProcess);

        }
    }

return_from_original_syscall:

    GrdDecThreadsIntoHooks();

    return NtStatus;
}
```

This is a post-success hook: the original `OldNtProtectVirtualMemory` is called first, and the dump logic only executes if the call succeeded and the new protection includes any executable flag (`PAGE_EXECUTE` , `PAGE_EXECUTE_READ` , `PAGE_EXECUTE_READWRITE` , or `PAGE_EXECUTE_WRITECOPY`). The target can be the calling process itself (common during unpacking, when a loader marks its own shellcode executable), or a different process, which is the classic cross-process injection scenario. In the cross-process case, this hook serves dual duty: the same protection change that signals injection also triggers the memory snapshot, capturing the injected payload at the instant it becomes executable before a single instruction of it has run.

To perform the actual dump, we read the target process's address space page by page using `MmCopyVirtualMemory` . This contrasts with the user-mode `ReadProcessMemory` , which requires a valid handle with `PROCESS_VM_READ` access and routes through `ntdll.dll` where user-mode hooks could interfere.

Our hook on `NtTerminateProcess` provides complementary coverage at the other end of the process lifecycle: intercepting the explicit termination path (`ExitProcess()` , `TerminateProcess()`) to dump the process before the kernel frees its address space. Because `NtTerminateProcess` is called from user-mode thread context and always executes at `PASSIVE_LEVEL` , the dump can be performed synchronously inside the hook, with the address space fully intact and accessible.

The dump subsystem also handles WoW64 processes correctly. 32-bit malware running under the WoW64 compatibility layer on a 64-bit OS is common in commodity crimeware, and the subsystem automatically detects this case and uses the appropriate memory information structures and address space for the dump (a detail that matters in practice given how prevalent 32-bit crimeware remains).

The resulting memory dumps contain decrypted payloads that were never written to disk, unpacked shellcode, C2 configuration data extracted from memory, and the runtime state of whatever the malware was doing at the moment of capture.

The Last-Thread Exit Blind Spot: Why `NtTerminateThread` Is Also Hooked

`NtTerminateProcess` covers the explicit termination path (`ExitProcess()` , `TerminateProcess()`) but there is a second path that bypasses it entirely. When a process terminates by calling `ExitThread()` or `TerminateThread()` on its last remaining thread, the call routes through `NtTerminateThread` , not `NtTerminateProcess` . Inside the kernel, `NtTerminateThread` ends up driving the thread exit through

`PspExitThread`, which (upon detecting that no other threads remain alive) calls `PspTerminateProcess` directly. This is an internal kernel-to-kernel call; it does not go through the SSDT, and the `NtTerminateProcess` SSDT hook is never invoked. This is not an edge case in the Windows implementation, it is a direct consequence of how the SSDT boundary works: it intercepts user-mode-to-kernel transitions for public system calls, and `PspTerminateProcess` is an internal function with no SSDT entry.

Our hook on `NtTerminateThread` closes this gap: when the hook fires for a thread in a monitored process, it checks the remaining thread count via `GetProcessThreadCount`. If `ThreadCount == 1` (the thread being terminated is the last), the hook performs the memory dump before calling the original function, capturing the process state before `PspExitThread` hands control to the internal cleanup path.

A Note on NtRaiseException as a Complementary Capture Point

`NtTerminateProcess` covers the explicit, clean-exit termination path. A complementary hook point worth considering alongside it is `NtRaiseException`. When a process encounters an unhandled exception. Whether from a genuine crash, a deliberate self-destruct triggered by anti-analysis logic, or a corrupted state induced by a packer; the exception dispatch path passes through `NtRaiseException` before the process address space is freed.

This makes it analytically valuable in several specific scenarios:

Packer exception cleanup routines: Certain packers register an unhandled exception filter via `SetUnhandledExceptionFilter` that erases or unmaps the unpacked payload before re-raising the exception as a deliberate forensic countermeasure. An `NtRaiseException` hook fires *before* this filter executes, ensuring the payload is captured in its unpacked state, not after the cleanup has run.

Anti-analysis exception flooding: Some malware families deliberately generate large volumes of exceptions in rapid succession as an anti-analysis technique, intending to overwhelm monitoring tools or trigger anomalous behavior in systems that process every exception individually. To address this, the dump subsystem applies two layers of rate limiting for exception-triggered dumps. The first is a time-based deduplication window per process. The second is a per-process hard cap on total exception dumps. Together, these limits prevent unbounded artifact generation from exception-heavy samples while still capturing the first meaningful occurrences.

Together, these hooks provide layered termination coverage: `NtRaiseException` fires before any exception filter or cleanup routine runs, preserving the process state at the moment the exception was raised; `NtTerminateProcess` catches the eventual explicit termination.

Dropped File Tracking and the Complete Artifact Chain

The filesystem minifilter monitors `IRP_MJ_CREATE` operations across the system. When a monitored process creates a new file, the minifilter captures the path. The driver maintains a complete artifact chain: dropper created `payload1.exe`, `payload1.exe` created `payload2.dll`, `payload2.dll` wrote configuration to `config.dat`, and so on. This way, analysts can understand the complete sequence of files involved in an infection, even when some of those files are temporary or were deleted. Combined with the pre-deletion dumping, every file that touched the system during the analysis (regardless of whether the malware attempted to clean it up) is preserved for examination.

To avoid duplicates (the same file creation event can trigger multiple callbacks), the minifilter maintains two circular buffers: one for file creation events and one for file move/rename events; using dual-hash deduplication (djb2 and sdbm). Atomic index allocation ensures lock-free operation from multiple concurrent callbacks. The result is a clean, non-redundant event stream even during aggressive file creation activity.

The deduplication mechanism is concise enough to show in full. Each entry stores two independent hash values; requiring both to match before treating an event as a duplicate keeps the false-positive rate negligible:

```
 typedef struct _RECENT_FILE_NEW_NOTIFICATION_ENTRY {
    ULONG FilePathHash1;     // Primary hash (djb2)
    ULONG FilePathHash2;     // Secondary hash (sdbm)
} RECENT_FILE_NEW_NOTIFICATION_ENTRY, *PRECENT_FILE_NEW_NOTIFICATION_ENTRY;

#define MAX_FILE_NEW_RECENT_NOTIFICATIONS 128

static RECENT_FILE_NEW_NOTIFICATION_ENTRY g_RecentFileNewNotifications[MAX_FILE_NEW_RECENT_NOTIFICATIONS] = { 0
static volatile LONG g_RecentFileNewNotificationIndex = 0;
```

When recording a new notification, a slot is allocated atomically and the two hash values are written:

```
 static VOID FsFltMarkFileNewAsNotified(CONST IN PCUNICODE_STRING FilePath)
{
    ULONG hash1 = StrHashUnicodeString_djb2(FilePath);
    ULONG hash2 = StrHashUnicodeString_sdbm(FilePath);

    if (hash1 == 0 || hash2 == 0)
        return; // Can't mark this file_new

    // Get unique index atomically - circular buffer semantics
    ULONG Index = InterlockedIncrement(&g_RecentFileNewNotificationIndex) - 1;
    Index = Index % MAX_FILE_NEW_RECENT_NOTIFICATIONS;

    PRECENT_FILE_NEW_NOTIFICATION_ENTRY Entry = &g_RecentFileNewNotifications[Index];
    Entry->FilePathHash1 = hash1;
    Entry->FilePathHash2 = hash2;
}
```

The `InterlockedIncrement` ensures each concurrent callback gets a unique buffer slot without any lock. When checking for duplicates, the code scans all entries comparing both hash values; a match on both means the event has already been dispatched and should be suppressed. The same pattern is used for the move/rename buffer, extended with four hash fields (old path and new path, each hashed twice).

Self-Protection: The Arms Race Malware Cannot Win

Traditional kernel rootkits are uncommon in contemporary commodity malware. This is partly due to the engineering cost, but also because Windows has made it significantly harder to operate stably at the kernel level. Kernel Patch Protection (PatchGuard) actively monitors critical kernel structures and triggers a system crash (`BSOD`) if unauthorized modifications are detected, making it a substantial barrier for malware that wants to tamper with the SSDT or other protected structures. But kernel-capable malware still exists, particularly in targeted attacks and sophisticated crimeware families.

The Meta-Rootkit Problem

A kernel-mode malware sample can walk the SSDT, detect entries that have been modified from their expected values, and restore the originals; eliminating our monitoring silently. Without a self-protection mechanism, a sufficiently sophisticated sample could blind our analysis driver entirely before executing its payload.

This is the meta-rootkit problem: two kernel-mode components competing for control of the same system structures. The attacker's rootkit unhooks our defensive rootkit. Our defensive rootkit never sees the payload.

Continuous Hook Integrity Monitoring

We address this with a dedicated protection thread that runs continuously, validating SSDT integrity and restoring any hooks that have been tampered with:

```
 /*
 * System thread routine that periodically checks and repairs SSDT hooks.
 *
 * Parameters:
 *   StartContext - Context passed from CreateProtectionThread (always NULL)
 *
 * Notes:
 *   - Must run at PASSIVE_LEVEL because:
 *     * SdtCheckAndPatchSSDTHooks requires PASSIVE_LEVEL (uses SuperCopyMemory)
 *     * KeDelayExecutionThread requires <= APC_LEVEL
 *   - Thread runs continuously until g_StopProtectionThread is set
 *   - Performs SSDT hook validation every 1 second
 *   - Signals g_ProtectionThreadExitEvent before terminating
 *   - Thread termination:
 *     * Normal: When g_StopProtectionThread is set
 *     * Error: On invalid parameters or IRQL
 *   - Never returns directly - always terminates via PsTerminateSystemThread
 *
 * Global Dependencies:
 *   g_StopProtectionThread - Controls thread execution loop
 *   g_ProtectionThreadExitEvent - Signaled before thread exit
*/
_IRQL_requires_(PASSIVE_LEVEL)
static VOID GrdProtectionThread(IN PVOID StartContext)
{
```

```
    UNREFERENCED_PARAMETER(StartContext);  // Always NULL

    DBG_INFO("[+] Protection thread started");

    // IRQL validation
    if (KeGetCurrentIrql() > PASSIVE_LEVEL)
    {
        DBG_ERROR("[!] Protection thread: Invalid IRQL, terminating");
        // Signal exit event even on error
        KeSetEvent(&g_ProtectionThreadExitEvent, IO_NO_INCREMENT, FALSE);
        PsTerminateSystemThread(STATUS_INVALID_LEVEL);
        return;
    }

    // Initialize stop flag
    g_StopProtectionThread = FALSE;

    // Set delay interval for periodic checks
    LARGE_INTEGER Delay;
    Delay.QuadPart = -1 * 1000 * 1000 * 10; /* 1 second */

    DBG_INFO("[+] Protection thread entering main loop");

    // Main protection loop
    while (!g_StopProtectionThread)
    {
        NTSTATUS NtStatus = SdtCheckAndPatchSSDTHooks();
        if (!NT_SUCCESS(NtStatus))
        {
            DBG_ERROR("[!] Unable to SdtCheckAndPatchSSDTHooks! NTSTATUS: 0x%X", NtStatus);
        }

        // When KeDelayExecutionThread returns, we are guaranteed to be in PASSIVE_LEVEL
        KeDelayExecutionThread(KernelMode, FALSE, &Delay);
    }

    DBG_INFO("[+] Protection thread exiting main loop");

    // Signal that thread is about to exit BEFORE calling PsTerminateSystemThread
    KeSetEvent(&g_ProtectionThreadExitEvent, IO_NO_INCREMENT, FALSE);

    DBG_INFO("[+] Protection thread terminating");

    // This is the normal exit path when g_StopProtectionThread is signaled
    PsTerminateSystemThread(STATUS_SUCCESS);
}
```

`SdtCheckAndPatchSSDTHooks` iterates over the `g_SSDT_HooksInfo` array populated at install time and compares each current SSDT entry against the stored `NewOffsetToFunction`. Any entry that differs is immediately restored:

```
 _IRQL_requires_(PASSIVE_LEVEL)
_Must_inspect_result_
NTSTATUS SdtCheckAndPatchSSDTHooks()
{
    if (KeGetCurrentIrql() > PASSIVE_LEVEL)
        return STATUS_INVALID_LEVEL;

    if (!g_KiServiceTable)
        return STATUS_INVALID_PARAMETER;

    NTSTATUS ret = STATUS_SUCCESS;

    for (ULONG i = 0; i < ARRAYSIZE(g_SSDT_HooksInfo); i++)
    {
        // Only check entries where hook installation succeeded
        if (g_SSDT_HooksInfo[i].OldOffsetToFunction && g_SSDT_HooksInfo[i].NewOffsetToFunction)
        {
            ULONG CurrentValue = SdtGetEntryFromSSDTById(g_KiServiceTable, g_SSDT_HooksInfo[i].SyscallId);

            ULONG ExpectedValue = g_SSDT_HooksInfo[i].NewOffsetToFunction;
            if (CurrentValue != ExpectedValue)
            {
                // Tampering detected - log and restore
                LgPrintfW(INFO, L"[ROOTKIT] Detected rootkit patch for SyscallId: %lu and SyscallName: %s",
                    g_SSDT_HooksInfo[i].SyscallId,
                    g_SSDT_HooksInfo[i].SyscallName);

                NTSTATUS NtStatus = SysSuperCopyMemory(
                    &g_KiServiceTable[g_SSDT_HooksInfo[i].SyscallId],
                    &ExpectedValue,
                    sizeof(ExpectedValue));

                if (!NT_SUCCESS(NtStatus))
                {
                    DBG_ERROR("[!] Unable to patch hook for SyscallId: %lu, SyscallName: %s. NTSTATUS: 0x%X",
                        g_SSDT_HooksInfo[i].SyscallId,
                        g_SSDT_HooksInfo[i].SyscallName,
                        NtStatus);
                    ret = NtStatus;
                    // Continue - try to restore as many hooks as possible
                }
            }
        }
    }
```

```
    return ret;
}
```

SysSuperCopyMemory is a utility function for writing to write-protected kernel memory. Beyond the CR0 manipulation, it performs upfront parameter validation (IRQL check, null pointer checks, size validation) before touching any protected memory. The core mechanism is Bit 16 of CR0, the hardware enforcement for page-level write protection: when set, the CPU honors read-only page attributes even in Ring 0; when cleared, kernel code can write to any mapped page regardless of its protection flags. The WP bit is cleared, RtlCopyMemory writes the new value, and the WP bit is restored. This same function is used both in hook installation and restoration.

The function attempts to restore every tampered entry before returning, maximizing the number of active hooks even if one restoration fails. Within one second of any tampering, the hook is restored.

This continuous monitoring also enables detection of malicious kernel activity: if an SSDT entry is found to have changed when no legitimate modification was expected, that itself is a signal that a kernel-mode actor has interfered with the analysis environment. Such events can be logged for attribution and flagged as indicators of sophisticated, kernel-aware malware. This is actionable intelligence about the sophistication of the sample being analyzed.

If malware continuously removes our hooks, we continuously restore them. This creates a feedback loop that malware cannot escape without either burning measurable CPU (which itself becomes an anomaly) or crashing the system, which defeats its own objectives.

It is worth being explicit about what this means in practice: a sample that achieves kernel-level control inside the analysis VM (a capability that in itself represents a high level of sophistication) will still be detected and logged, because the defensive rootkit detects the tampering regardless of how it was performed. This is not a capability that analysis platforms commonly provide. The self-protection mechanism turns kernel-level interference from a blind spot into a detection signal.

Atomic Thread Counting for Safe Unhooking

There is a subtle race condition when restoring SSDT entries (e.g. during tampering response). A thread may be executing inside our hook function at the exact moment we overwrite the SSDT entry. We track in-flight hook executions with an atomic counter:

```
 // Atomic counter of threads currently executing hooked functions
static volatile LONG g_NumThreadsInsideHook = 0;

/*
 * Atomically increments the count of threads executing inside SSDT hooks.
 *
 * Notes:
 *   - No IRQL restrictions as InterlockedIncrement is safe at any IRQL
 *   - Thread-safe through use of InterlockedIncrement
```

```
 *   - Must be paired with a subsequent decrement
 *   - Used when a thread enters a hooked function
 *   - Updates g_NumThreadsInsideHook with new incremented value
 *
 * Global Dependencies:
 *   g_NumThreadsInsideHook - Atomic counter tracking threads in hooks
 *
 * Warning:
 *   - Must only be called when actually entering a hook
*/
VOID GrdIncThreadsIntoHooks()
{
    InterlockedIncrement(&g_NumThreadsInsideHook);
}

/*
 * Atomically decrements the count of threads executing inside SSDT hooks.
 *
 * Notes:
 *   - No IRQL restrictions as InterlockedDecrement is safe at any IRQL
 *   - Thread-safe through use of InterlockedDecrement
 *   - Must be paired with a previous increment
 *   - Used when a thread exits a hooked function
 *   - Updates g_NumThreadsInsideHook with new decremented value
 *
 * Global Dependencies:
 *   g_NumThreadsInsideHook - Atomic counter tracking threads in hooks
 *
 * Warning:
 *   - Must only be called when actually exiting a hook
*/
VOID GrdDecThreadsIntoHooks()
{
    InterlockedDecrement(&g_NumThreadsInsideHook);
}
```

Every hook function increments this counter on entry (before any other logic) and decrements it on exit (after all logic, in every return path). This can be seen in the hook code snippets shared. When unhooking, we wait for the counter to reach zero before freeing hook memory:

```
 /*
 * Safely unhooks the SSDT and waits for any pending hook operations to complete.
 *
 * Returns:
 * NTSTATUS:
```

```
 * - STATUS_SUCCESS - SSDT successfully unhooked and all operations completed
 * - STATUS_INVALID_LEVEL - Called at wrong IRQL
 * - Other NTSTATUS values from SdtRestoreSSDT
 *
 * Notes:
 * - Must run at PASSIVE_LEVEL because:
 * * SdtRestoreSSDT requires PASSIVE_LEVEL (SuperCopyMemory)
 * * Uses KeDelayExecutionThread which requires <= APC_LEVEL
 * - Function will block until all threads exit hooked functions
 * - Allows one thread (current IOCTL handler) to remain in hooks
 * - Implements graceful shutdown by:
 * 1. Restoring original SSDT entries
 * 2. Waiting for pending operations to complete
 * - Called through IOCTL interface
*/
_IRQL_requires_(PASSIVE_LEVEL)
_Must_inspect_result_
NTSTATUS GrdSafeManageUnhookSsdtIoctl()
{
 // IRQL validation
 if (KeGetCurrentIrql() > PASSIVE_LEVEL)
 {
 return STATUS_INVALID_LEVEL;
 }

 // Restore original SSDT entries (Unhook SSDT)
 NTSTATUS NtStatus = SdtRestoreSSDT();
 if (!NT_SUCCESS(NtStatus))
 {
 DBG_ERROR("[!] Unable to Restore SSDT. NTSTATUS: 0x%X", NtStatus);
 return NtStatus;
 }
 else
 {
 DBG_INFO("[+] Successfully Restored SSDT");
 }

 // Wait for threads to exit hooks
 // We allow one thread (current) to remain as it's the IOCTL handler
 // (We accept one thread because this thread has came from NtDeviceIoControlFile SSDT hook)
 while (GrdNumThreadsIntoHooks() > 1)
 {
 LARGE_INTEGER Delay;
 Delay.QuadPart = -1 * 1000 * 1000 * 10; /* 1 second */
 KeDelayExecutionThread(KernelMode, FALSE, &Delay);
 }
```

```
    return STATUS_SUCCESS;
}
```

InterlockedIncrement and InterlockedDecrement are safe at any IRQL, making this counter correct regardless of the IRQL at which a hook fires. The “tolerate 1” comment is important: the IOCTL handler that initiates unhooking may itself be executing inside a hook function, so the counter will be at least 1 during the wait. The entire safe-unhooking mechanism (including the thread counter, the SSDT restoration, and the drain wait), is exposed through the IOCTL interface, making SSDT hooks protection a dynamically controllable capability.

Real-World Validation: Families That Confirm the Architectural Choice

The techniques described in this post did not emerge from theoretical analysis. They emerged from encountering real samples that defeated conventional analysis platforms and understanding exactly how they did it. This section describes representative malware families well-documented in public research, whose evasion techniques map directly to the architectural decisions we made, and explains how each driver capability addresses each family’s anti-analysis approach.

GuLoader: The RDTSC Problem

GuLoader is a highly obfuscated malware family first seen in December 2019. Its anti-analysis approach is representative of the state of the art: a multi-layered gauntlet that checks several independent sources before proceeding. We will now go through each of these checks and how our architecture addresses them.

The `RDTSC` timing check is GuLoader’s most architecturally significant evasion. The loader executes a tight loop of instructions bracketed by two `RDTSC` reads, computes the delta, and compares it against a threshold. On physical hardware, this delta is small and consistent. In a virtualized environment the delta is measurably larger, because instructions inside that loop that cause VM exits (such as `CPUID`, which unconditionally triggers a VM exit in VMX non-root mode regardless of any hypervisor configuration) introduce round-trip latency through the hypervisor on every execution. GuLoader uses this discrepancy as a reliable VM indicator.

A traditional kernel-level approach to bypass RDTSC involves setting the CR4.TSD bit so that any user-mode execution of the instruction triggers a general protection fault (#GP). By hooking the corresponding IDT entry (interrupt 0x0D), the kernel can intercept the fault, verify that the instruction is indeed RDTSC, and emulate it by returning crafted timestamp values while skipping the original instruction. However, this method introduces noticeable side effects (like abnormal exception patterns) that can be detected by security mechanisms. In contrast, handling this at the hypervisor level avoids tampering with the guest OS entirely and provides stronger isolation, since the control logic operates outside the operating system’s visibility, making it both more robust against OS-level defenses and harder for software running inside the system to detect or interfere with.

This is precisely the capability that motivates the hypervisor layer in our architecture. One subtlety worth noting: unlike `CPUID`, `RDTSC` does not cause a VM exit by default. A hypervisor must explicitly enable the **RDTSC exiting** control bit (bit 12 of the Primary Processor-Based VM-Execution Controls) to intercept it; otherwise `RDTSC` executes at native hardware speed without any hypervisor involvement. An alternative is the **TSC offset**

feature, which lets the hypervisor add a constant to the TSC value returned to the guest without any VM exit at all, transparent to the guest and with zero per-instruction overhead. The check passes. GuLoader proceeds.

CPUID vendor fingerprinting is GuLoader's second major check. It executes `CPUID` with leaf 0 and inspects the returned vendor string against a list of known virtualization platform identifiers. The hypervisor present bit in CPUID leaf 1 ECX (bit 31) is also checked. Our hypervisor synthesizes a physical CPU vendor string and clears the hypervisor bit, making the environment appear as physical hardware to any code that interrogates `CPUID`.

Process and registry artifact checks complete GuLoader's gauntlet. It queries process names and registry keys for known sandbox indicators. Our `NtQuerySystemInformation` hook filters the process list; our registry callback hides VM-revealing keys. The complete GuLoader evasion sequence encounters a different kind of obstacle at every check: hardware-level responses for hardware-level checks, kernel-level responses for OS-level checks; with consistent results across all sources. In conventional sandbox testing, GuLoader historically achieves near-zero detonation rates against platforms lacking hypervisor-level capabilities. In our mixed-layer environment, GuLoader's full payload chain detonates reliably.

LummaC2: Direct Syscalls, Heaven's Gate, and Related Techniques

LummaC2 (also known as Lumma Stealer) is a modern and sophisticated information stealing malware. Its anti-monitoring approach targets the most common sandbox instrumentation technique: user-mode API hooking.

LummaC2 resolves critical APIs dynamically and invokes them via direct `SYSCALL` instructions rather than through `ntdll.dll` stubs, explicitly bypassing the function prologues where user-mode hooks (inline detours placed by AVs, EDRs, and sandbox agents on `Nt*` / `Zw*` exports) are inserted. Some variants also employ Heaven's Gate: a CPU mode-switching mechanism that lets a 32-bit process issue a far jump to segment `0x33`, transitioning to 64-bit execution. Once in 64-bit mode, the malware can either call 64-bit `ntdll.dll` functions directly (bypassing the 32-bit WoW64 instrumentation layer entirely) or skip `ntdll.dll` altogether and issue a raw `SYSCALL`, bypassing both layers at once.

The same goal of avoiding hooked `ntdll.dll` stubs has produced a family of widely used techniques: Hell's Gate and its extensions (Halo's Gate, Tartarus Gate), SysWhispers/SysWhispers2/SysWhispers3, etc. The underlying logic is consistent across all variants: avoid `ntdll.dll` function prologues entirely, because that is where every layer of user-mode instrumentation is commonly placed.

Why neither technique affects our architecture: SSDT hooks operate at Ring 0, intercepting the syscall at the kernel transition point: below the entire user-mode layer. LummaC2's direct `SYSCALL` instructions still transition to Ring 0 and still hit our SSDT hooks, because the intercept is not in `ntdll.dll` and cannot be bypassed by avoiding `ntdll.dll`. Heaven's Gate transitions are no different: regardless of which execution mode issues the `SYSCALL`, all user-mode syscalls enter the same kernel dispatch path that our hooks intercept. LummaC2's anti-monitoring logic passes harmlessly. Our hooks remain active and invisible.

This is a category-level advantage over user-mode monitoring. Not a better implementation of the same approach, but a fundamentally different intercept location that user-mode hook scanning cannot reach by design.

xLoader and SmokeLoader: Private ntdll Copies as Hook Bypass

xLoader (versions 6 and 7) loads a private copy of `ntdll.dll` into its own memory space and resolves API functions through that copy instead of the system-loaded instance. If a monitoring tool has placed breakpoints or inline hooks on the exported functions of the original `ntdll.dll`, xLoader's private copy remains unmodified. Its API calls route through the clean copy, bypassing user-mode instrumentation entirely.

SmokeLoader takes the same approach with variant-specific mechanics. Int latest variants it creates a copy of `ntdll.dll` in memory using `CreateFileMappingW` and `MapViewOfFile`. In its early stages, the malware dropped a copy of the library to disk in the `%TEMP%` folder and loaded it from there via `LdrLoadDll`. In all cases, the private copy is unhooked, allowing SmokeLoader to execute its API calls without triggering any instrumentation placed on the system `ntdll.dll`.

Why loading a private ntdll copy fails against kernel-level monitoring: `ntdll.dll` is a user-mode wrapper. Its exported functions ultimately issue `SYSCALL` instructions that transition to Ring 0. Regardless of which copy of `ntdll.dll` a process loads (the system copy, a private memory-mapped copy, or one extracted from disk), every call requiring OS services must eventually execute a `SYSCALL` instruction. That transition is the point where our SSDT hooks intercept. There is no user-space path that bypasses this transition; it is architecturally mandated. The private ntdll technique defeats user-mode instrumentation precisely because user-mode instrumentation sits above the syscall boundary. Our monitoring sits below it.

Multi-Stage Loaders: The Persistence-Based Staging Gap

An important blind spot in some sandbox analysis platforms is not any individual evasion technique, it is the structural failure to observe final-stage payloads that only deploy after persistence is confirmed.

Consider the following execution chain, representative of how contemporary loaders operate:

1. Stage 1 (dropper) arrives via phishing. It performs environment checks and, finding the environment acceptable, drops a stage 2 binary and writes a registry Run key.
2. Stage 1 exits. The sandbox records: file written, registry key written, process exited. Analysis complete.
3. Stage 2 (the actual payload) never executes, because nothing triggers the Run key.

Against a conventional sandbox with a fixed analysis window and no mechanism to trigger persistence, this loader produces no observable malicious payload behavior. Detection rate for the final payload in conventional automated sandboxes: near zero.

Against our platform: when Stage 1 writes the Run key, our registry callback's `RfPreSetValueKey` handler captures the registered binary path and triggers execution of Stage 2 (simulating the reboot or login event that would normally trigger it). The full behavioral record of Stage 2, including any further stages it deploys, is captured within the same analysis session.

By detecting persistence establishment in real time and immediately triggering next-stage execution, the platform collapses what would otherwise require a multi-stage, multi-session manual analysis workflow into a single automated session. Importantly, the platform also **dynamically extends** the analysis time budget when persistence events are detected (instead of running for a fixed window and stopping before next-stage payloads complete their

behavior), the session continues until all triggered stages have run to completion. This ensures that even heavily staged loaders produce a complete behavioral record, from the initial dropper through the final payload.

Why Architecture, Not Signatures

As we've seen throughout this section, the common thread is that these evasion techniques are not advanced nation-state tradecraft (they are documented, commoditized capabilities in widely used crimeware kits). GuLoader's `RDTSC` trick has been described in public research since at least 2020. SmokeLoader and xLoader anti-hook mechanisms are widely documented (xLoader technical analysis, SmokeLoader history).

The sandbox that defeats these techniques is not one with better signatures for each specific check. It is one whose architecture operates at the correct layer to make the checks irrelevant. `RDTSC` timing is irrelevant when the hypervisor controls the result. User-mode hook scanning is irrelevant when there are no user-mode hooks. Registry VM artifact checks are irrelevant when a Ring 0 minifilter intercepts them first. Persistence-based staging is irrelevant when the platform triggers the persistence mechanism itself.

This is the architectural argument: not that it catches specific malware families, but that it eliminates entire *categories* of evasion technique by operating at the layer where those techniques have no effect.

Conclusion: The Future of Malware Analysis. What We Built and Why It Matters

The driver described here addresses a problem that has become increasingly urgent: sophisticated malware routinely defeats conventional analysis environments, delivering nothing observable in sandboxes while operating fully against real targets. The result is a systematic blind spot at the center of many organizations' threat intelligence pipelines.

Our kernel-level analysis driver closes that blind spot. It provides:

- **Syscall interception at Ring 0**, below any user-mode detection mechanism, with full access to caller context and result data.
- **Active detonation assistance**: Returning curated responses to every environment probe, ensuring malware concludes the environment is real and executes its payload.
- **Multi-layer anti-evasion** addressing time manipulation, hardware fingerprinting, registry and filesystem artifact discovery, and process enumeration in cooperation with hypervisor-level protections that cover what the kernel cannot reach.
- **Persistence detection with forced execution**: Capturing the complete infection chain, including final-stage payloads that only deploy after persistence is confirmed.
- **Forensic preservation** of evidence that malware routinely destroys: files deleted before analysts can examine them, memory state at process termination, dropped payload chains.
- **Automatic injection chain tracking** following the infection through every spawned child process and injection target without manual configuration.
- **Self-protection** against tampering by kernel-level malware samples, with detection capability for kernel-aware adversaries.

None of the individual techniques here are novel in isolation and we want to be precise about that. SSDT hooking is documented in depth in public literature. Code cave discovery is a well-known technique in rootkit development. Minifilters are a standard Windows driver pattern covered in the WDK documentation and countless driver development resources. What is novel is not the techniques, it is what the integration of these techniques *achieves* that no subset of them can achieve independently.

Complete infection chain capture in a single session: The combination of real-time persistence detection with forced execution of registered payloads inside a continuously monitored analysis session is what enables full infection chain visibility (from the initial dropper through every subsequent stage), within a single analysis run. This is not simply a matter of better evasion resistance at any single layer; it comes from the kernel-level persistence monitor and the payload triggering mechanism being co-designed with full awareness of each other.

Cross-layer consistency under adversarial cross-checking: Sophisticated samples do not rely on a single environment check, they cross-check multiple independent sources. A representative example: a sample that queries total physical memory via `NtQuerySystemInformation` with `SystemBasicInformation`, then reads `KUSER_SHARED_DATA.NumberOfPhysicalPages` directly (a technique requiring no syscall, and therefore invisible to SSDT-only solutions), and finds any inconsistency between them, aborts. Our architecture manages both sources simultaneously: the `NtQuerySystemInformation` SSDT hook adjusts `NumberOfPhysicalPages` in the returned structure, while `KUSER_SHARED_DATA.NumberOfPhysicalPages` is patched directly from kernel mode at initialization (using the same constant, ensuring the values are identical). The coherence across these sources is not incidental, it's explicitly enforced. That kind of cross-layer consistency management is not present in any single-layer solution.

Zero semantic gap with complete interception: A hypervisor below the OS must infer which process made which syscall, what the arguments meant in OS terms, and whether a particular memory write corresponds to a persistence event. Our kernel driver has all of this natively, without translation, at the point of every syscall. That semantic immediacy is what makes behavioral monitoring precise enough to be useful at production scale, and undetectable enough to not be a detection signal itself.

In short, the platform described here is distinguished not by any single capability but by the combination: Ring 0 syscall interception that cannot be bypassed by user-mode techniques, persistence detection and forced execution that closes the staging gap, forensic preservation that survives malware cleanup routines, injection tracking that follows the infection across process boundaries, and cross-layer consistency enforcement that defeats multi-source environment fingerprinting. Each of these properties depends on operating at the kernel level; none of them is achievable from user-mode or hypervisor-only approaches alone.

The Mixed-Layer Advantage

As described in Section 3, the combination of hypervisor-level and kernel-level monitoring is what closes the full coverage gap. The mixed-layer architecture delivers capabilities that neither layer could provide on its own. Hardware-level probes are answered at the hypervisor before they reach the OS. OS-level syscalls are intercepted at Ring 0 with full semantic context. Persistence mechanisms are detected and triggered in real time. Injected code is tracked across process boundaries. Forensic evidence is preserved before malware can destroy it. Taken

together, these cover the full spectrum of what evasive malware does, from the first environment check to the final payload, within a single analysis session.

A sample performing both hardware-level and OS-level environment checks encounters a consistent picture of a physical workstation at every layer it probes. A sample attempting to evade monitoring through persistence-based staging encounters forced execution of its next stage. A sample attempting to destroy evidence encounters a driver that has already preserved it.

This is not two systems working in parallel. It is a single, integrated analysis platform with each layer doing what it does best.

Where the Industry Is Heading

The trajectory is clear, and the data supports it. MITRE ATT&CK's Virtualization/Sandbox Evasion category (T1497) has been among the top-ten most observed techniques in real-world incident reporting for multiple consecutive years. What used to be advanced is now easily accessible in a short time: techniques that required resources in 2018 ship in commercial crimeware kits today, available to any threat actor willing to pay a subscription fee.

For CISOs and security program owners, the practical implication is straightforward. If your threat intelligence pipeline depends on automated sandbox analysis (as most do, given the volume of samples requiring triage) and if that sandbox cannot handle evasive samples, then your intelligence is systematically incomplete for exactly the threats that are most likely to be used against you. A loader that bypasses your sandbox does not appear in your behavioral detection baseline, does not contribute to your IOC feeds, and does not generate the telemetry that would train your ML-based detection models. The evasion is not just a single missed detection, it is a persistent blind spot in your defensive posture.

Evasion techniques that were sophisticated APT tradecraft five years ago are commodity crimeware today. Multi-stage loaders, behavioral checks, hardware fingerprinting, persistence-based staging; these ship in off-the-shelf crimeware kits sold on underground markets. Analysis platforms that cannot counter them are operating with progressively declining visibility into real-world threats.

Kernel-level analysis has already become a baseline expectation for enterprise malware analysis platforms, following the same pattern by which behavioral analysis replaced signature detection a decade ago. Enterprise-grade sandboxes and analysis platforms increasingly rely on kernel ETW providers, kernel callbacks, and minifilter-based monitoring precisely because user-mode instrumentation is insufficient against modern threats. The same architectural reasoning that made kernel-level EDR more effective than user-mode EDR has driven the same transition in malware analysis platforms. The question today is not whether to operate at the kernel level, but how deep and how precisely that kernel-level instrumentation is integrated and whether it is paired with the persistence detection, forensic capture, and anti-evasion capabilities that turn raw syscall visibility into actionable behavioral intelligence.

If evasive malware does not detonate in your analysis environment, and if you cannot see what happens after it establishes persistence, you are making security decisions based on incomplete information. At Zynap Labs, we built the infrastructure to change that.

Source: https://www.zynap.com/blog/defensive-rootkits-engineering-kernel-level-malware-analysis-ring-0/