How Kernel Anti-Cheats Work: A Deep Dive into Modern Game Protection

s4dbrd.github.io/posts/how-kernel-anti-cheats-work

February 22, 2026

Modern kernel anti-cheat systems are, without exaggeration, among the most sophisticated pieces of software running on consumer Windows machines. They operate at the highest privilege level available to software, they intercept kernel callbacks that were designed for legitimate security products, they scan memory structures that most programmers never touch in their entire careers, and they do all of this transparently while a game is running. If you have ever wondered how BattlEye actually catches a cheat, or why Vanguard insists on loading before Windows boots, or what it means for a PCIe DMA device to bypass every single one of these protections, this post is for you.

This is not a comprehensive or authoritative reference. It is just me documenting what I found and trying to explain it clearly. Some of it comes from public research and papers I have linked at the bottom, some from reading kernel source and reversing drivers myself. If something is wrong, feel free to reach out. The post assumes some familiarity with Windows internals and low-level programming, but I have tried to explain each concept before using it.

1. Introduction

Why Usermode Protections Are Not Enough

The fundamental problem with usermode-only anti-cheat is the trust model. A usermode process runs at ring 3, subject to the full authority of the kernel. Any protection implemented entirely in usermode can be bypassed by anything running at a higher privilege level, and in Windows that means ring 0 (kernel drivers) or below (hypervisors, firmware). A usermode anti-cheat that calls `ReadProcessMemory` to check game memory integrity can be defeated by a kernel driver that hooks `NtReadVirtualMemory` and returns falsified data. A usermode anti-cheat that enumerates loaded modules via `EnumProcessModules` can be defeated by a driver that patches the PEB module list. The usermode process is completely blind to what happens above it.

Cheat developers understood this years before most anti-cheat engineers were willing to act on it. The kernel was, for a long time, the exclusive domain of cheats. Kernel-mode cheats could directly manipulate game memory without going through any API that a usermode anti-cheat could intercept. They could hide their presence from usermode enumeration APIs trivially. They could intercept and forge the results of any check a usermode anti-cheat might perform.

The response was inevitable: move the anti-cheat into the kernel.

The Arms Race

The escalation has been relentless. Usermode cheats gave way to kernel cheats. Kernel anti-cheats appeared in response. Cheat developers began exploiting legitimate, signed drivers with vulnerabilities to achieve kernel execution without loading an unsigned driver (the BYOVD attack). Anti-cheats responded with blocklists and stricter driver enumeration. Cheat developers moved to hypervisors, running below the kernel and virtualizing the entire OS. Anti-cheats added hypervisor detection. Cheat developers began using PCIe DMA devices to read

game memory directly through hardware without ever touching the OS at all. The response to that is still being developed.

Each escalation requires the attacking side to invest more capital and expertise, which has an important effect: it filters out casual cheaters. A $30 kernel cheat subscription is accessible to many people. A custom FPGA DMA setup costs hundreds of dollars and requires significant technical knowledge to configure. The arms race, while frustrating for anti-cheat engineers, does serve the practical goal of making cheating expensive and difficult enough that most cheaters do not bother.

Major Anti-Cheat Systems

Four systems dominate the competitive gaming landscape:

BattlEye is used by PUBG, Rainbow Six Siege, DayZ, Arma, and dozens of other titles. Its kernel component is `BEDaisy.sys`, and it has been the subject of detailed public reverse engineering work, most notably by the secret.club researchers and the back.engineering blog.

EasyAntiCheat (EAC) is now owned by Epic Games and used in Fortnite, Apex Legends, Rust, and many others. Its architecture is broadly similar to BattlEye in its three-component design but differs significantly in implementation details.

Vanguard is Riot Games' proprietary anti-cheat used in Valorant and League of Legends. It is notable for loading its kernel component (`vgk.sys`) at system boot rather than at game launch, and for its aggressive stance on driver allowlisting.

FACEIT AC is used for the FACEIT competitive platform for Counter-Strike. It is a kernel-level system with a well-regarded reputation in the competitive community for effective cheat detection, and has been the subject of academic analysis examining the architectural properties of kernel anti-cheat software more broadly.

The ARES 2024 Paper on Kernel Anti-Cheat Architecture

The 2024 paper "If It Looks Like a Rootkit and Deceives Like a Rootkit" (presented at ARES 2024) analyzed FACEIT AC and Vanguard through the lens of rootkit taxonomy, noting that both systems share technical characteristics with that class of software: kernel-level operation, system-wide callback registration, and broad visibility into OS activity. The authors are careful to distinguish between technical classification and intent, explicitly acknowledging that these systems are legitimate software serving a defensive purpose. The paper's contribution is primarily taxonomic rather than accusatory.

The underlying observation is simply that effective kernel anti-cheat requires the same OS primitives that malicious kernel software uses, because those primitives are what provide the visibility needed to detect cheats. Any sufficiently capable kernel anti-cheat will look like a rootkit under static behavioral analysis, because capability and intent are orthogonal at the kernel API level. This is a constraint imposed by Windows architecture, not a design choice unique to any particular anti-cheat vendor.

2. Architecture of a Kernel Anti-Cheat

The Three-Component Model

Modern kernel anti-cheats universally follow a three-layer architecture:

1. **Kernel driver**: Runs at ring 0. Registers callbacks, intercepts system calls, scans memory, enforces protections. This is the component that actually has the power to do anything meaningful.
2. **Usermode service**: Runs as a Windows service, typically with `SYSTEM` privileges. Communicates with the kernel driver via IOCTLs. Handles network communication with backend servers, manages ban enforcement, collects and transmits telemetry.
3. **Game-injected DLL**: Injected into (or loaded by) the game process. Performs usermode-side checks, communicates with the service, and serves as the endpoint for protections applied to the game process specifically.

The separation of concerns here is both architectural and security-motivated. The kernel driver can do things no usermode component can, but it cannot easily make network connections or implement complex application logic. The service can do those things but cannot directly intercept system calls. The in-game DLL has direct access to game state but runs in an untrustworthy ring-3 context.

Communication Channels

IOCTLs (I/O Control Codes) are the primary communication mechanism between usermode and a kernel driver. A usermode process opens a handle to the driver's device object and calls `DeviceIoControl` with a control code. The driver handles this in its `IRP_MJ_DEVICE_CONTROL` dispatch routine. The entire communication is mediated by the kernel, which means a compromised usermode component cannot forge arbitrary kernel operations - it can only make requests that the driver is programmed to service.

Named pipes are used for IPC between the service and the game-injected DLL. A named pipe is faster and simpler than routing everything through the kernel, and it allows the service to push notifications to the game component without polling.

Shared memory sections created with `NtCreateSection` and mapped into both the service process and the game process via `NtMapViewOfSection` allow high-bandwidth, low-latency data sharing. Telemetry data (input events, timing data) can be written to a shared ring buffer by the game DLL and read by the service without the overhead of IPC per event.

Boot-time vs Runtime Driver Loading

The distinction between boot-time and runtime driver loading is more significant than it might appear.

BattlEye and EAC load their kernel drivers when the game is launched. `BEDaisy.sys` and its EAC equivalent are registered as demand-start drivers and loaded via `ZwLoadDriver` from the service when the game starts. They are unloaded when the game exits.

Vanguard loads `vgk.sys` at system boot. The driver is configured as a boot-start driver (`SERVICE_BOOT_START` in the registry), meaning the Windows kernel loads it before most of the system has initialized. This gives Vanguard a critical advantage: it can observe every driver that loads after it. Any driver that loads after `vgk.sys` can be inspected before its code runs in a meaningful way. A cheat driver that loads at the normal driver initialization phase is loading into a system that Vanguard already has eyes on.

The practical implication of boot-time loading is also why Vanguard requires a system reboot to enable: the driver must be in place before the rest of the system initializes, which means it cannot be loaded after the fact without a restart.

Driver Signing Requirements

Windows enforces **Driver Signature Enforcement (DSE)** on 64-bit systems, which requires that kernel drivers be signed with a certificate that chains to a trusted root and that the driver's code integrity be verified at load time. This is implemented through `CiValidateImageHeader` and related functions in `ci.dll`. The kernel also enforces that driver certificates meet certain Extended Key Usage (EKU) requirements.

Anti-cheats handle signing in the obvious way: they pay for extended validation (EV) code signing certificates, go through Microsoft's WHQL process for some components, or use cross-signing. The certificate requirements have tightened significantly over the years; Microsoft now requires EV certificates for new kernel drivers, and the kernel driver signing portal requires WHQL submission for drivers targeting Windows 10 and later in many cases.

The reason this matters for cheats is that DSE is a significant barrier. Without a signed driver or a way to bypass DSE, a cheat author cannot load arbitrary kernel code. BYOVD attacks (covered in section 7) are the primary mechanism for bypassing this restriction.

BattlEye Component Breakdown

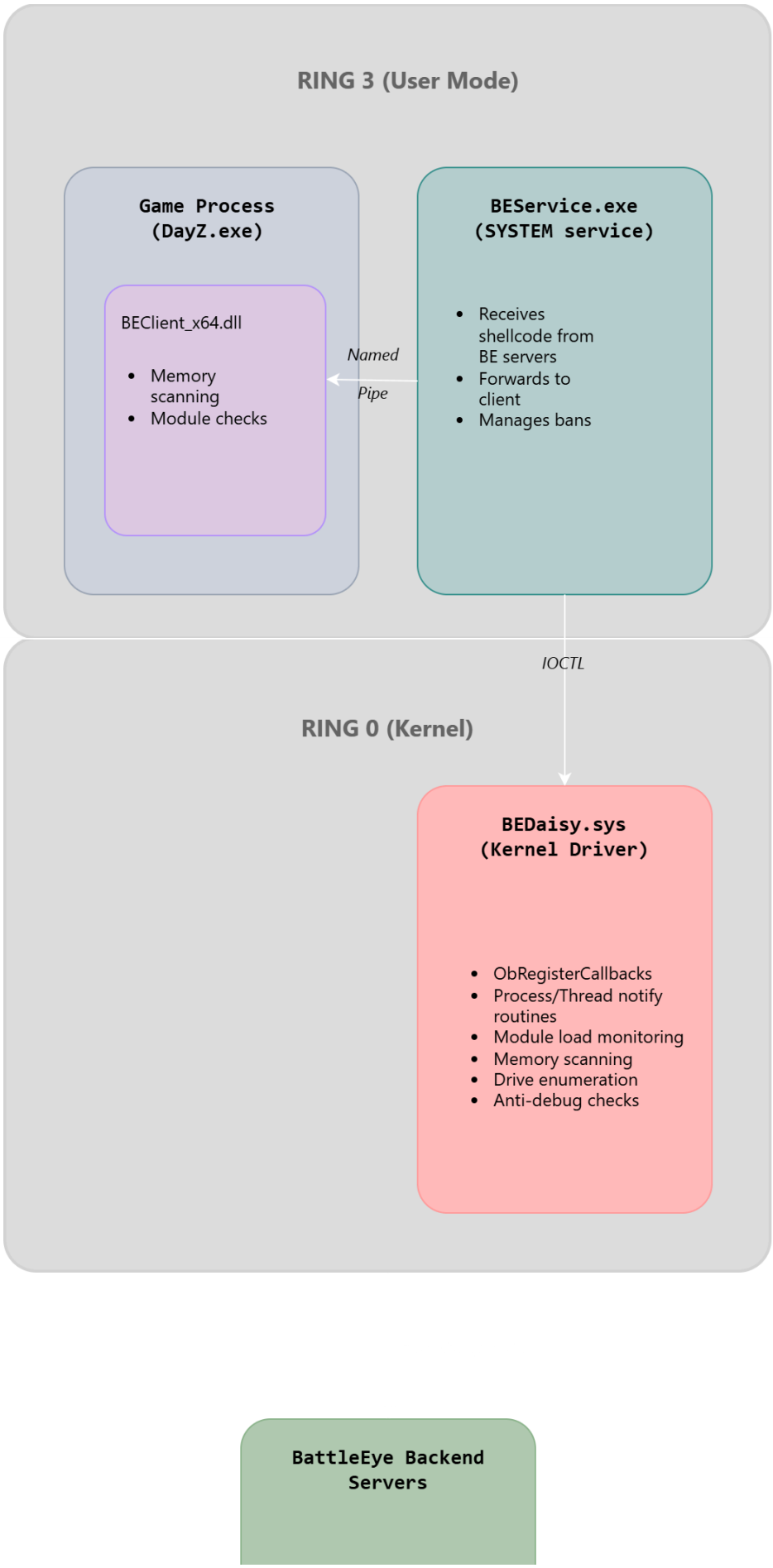
BattlEye's architecture is well-documented through reverse engineering:

`BEDaisy.sys` is the kernel driver. It registers callbacks for process creation, thread creation, image loading, and object handle operations. It implements the actual scanning and protection logic.

`BEService.exe` (or `BEService_x64.exe`) is the usermode service. It communicates with `BEDaisy.sys` via a device object that the driver exposes. It handles network communication with BattlEye's backend servers, receives detection results from the driver, and is responsible for ban enforcement (kicking the player from the game server).

`BEClient_x64.dll` is injected into the game process. BattlEye does not inject this via `CreateRemoteThread` in the traditional sense - it is loaded as part of game initialization, with the game's cooperation. This DLL is responsible for performing usermode-side checks within the game process context: it verifies its own integrity, performs various environment checks, and serves as the target for protections that the kernel driver applies specifically to the game process.

The communication flow goes: `BEDaisy.sys` detects something suspicious, signals `BEService.exe` via an IOCTL completion or a shared memory notification, `BEService.exe` reports to BattlEye's servers, the server decides on an action (kick/ban), and `BEService.exe` instructs the game to terminate the connection.


RING 3 (User Mode)
Game Process
(DayZ.exe)
BEClient_x64.dll
Memory scanning
Module checks
Named
Pipe
BEService.exe
(SYSTEM service)
Receives shellcode from BE servers
Forwards to client
Manages bans
IOCTL
RING 0 (Kernel)
BEDaisy.sys
(Kernel Driver)
ObRegisterCallbacks
Process/Thread notify routines
Module load monitoring
Memory scanning
Drive enumeration
Anti-debug checks
BattleEye Backend Servers

BattlEye's three-component architecture: BEDaisy.sys at ring 0 communicates upward via IOCTLs to BEService.exe running as a SYSTEM service, which manages BEClient_x64.dll injected in the game process.

Vanguard's Architecture

`vgk.sys` is notably more aggressive than the BattlEye driver in its scope. Because it loads at boot, it can intercept the driver load process itself. Vanguard maintains an internal allowlist of drivers that are permitted to co-exist with a protected game. Any driver not on this list, or any driver that fails integrity checks, can result in Vanguard refusing to allow the game to launch. This is an allowlist model rather than a blocklist model, which is architecturally much stronger.

`vgauth.exe` is the Vanguard service, which handles the communication between `vgk.sys` and Riot's backend infrastructure.

3. Kernel Callbacks and Monitoring

This is the foundation of everything a kernel anti-cheat does. The Windows kernel exposes a rich set of callback registration APIs intended for security products, and anti-cheats use every one of them.

ObRegisterCallbacks

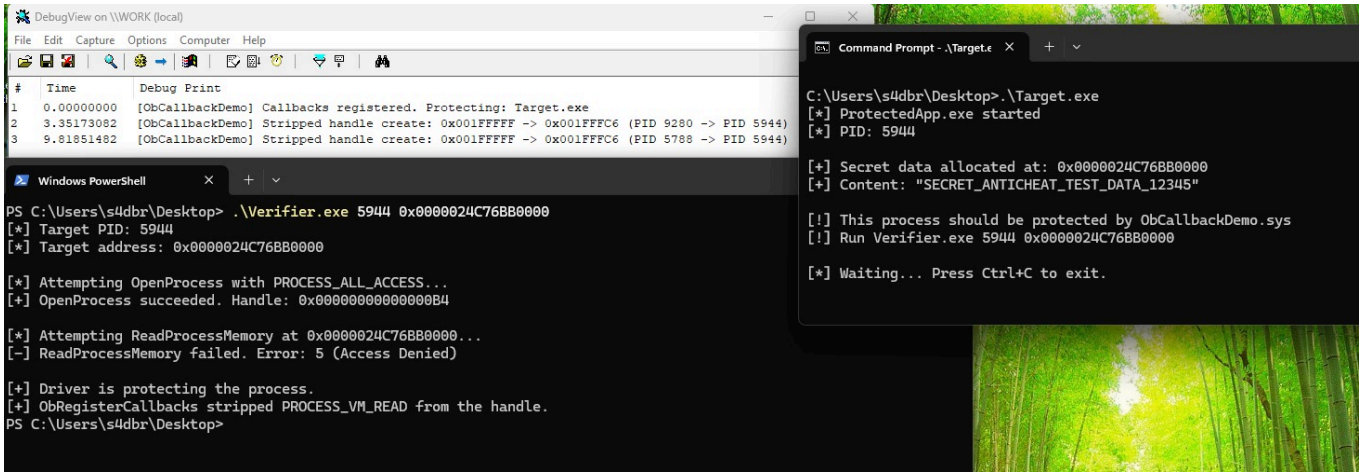
`ObRegisterCallbacks` is perhaps the single most important API for process protection. It allows a driver to register a callback that is invoked whenever a handle to a specified object type is opened or duplicated. For anti-cheat purposes, the object types of interest are `PsProcessType` and `PsThreadType`.

```
OB_CALLBACK_REGISTRATION callbackReg = {0};
OB_OPERATION_REGISTRATION opReg[2] = {0};

// Altitude string is required - must be unique per driver
UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"31001");

// Monitor handle opens to process objects
opReg[0].ObjectType = PsProcessType;
opReg[0].Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICA
opReg[0].PreOperation = ObPreOperationCallback;
opReg[0].PostOperation = ObPostOperationCallback;

// Monitor handle opens to thread objects
opReg[1].ObjectType = PsThreadType;
opReg[1].Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICA
opReg[1].PreOperation = ObPreOperationCallback;
opReg[1].PostOperation = NULL;

callbackReg.Version = OB_FLT_REGISTRATION_VERSION;
callbackReg.OperationRegistrationCount = 2;
callbackReg.Altitude = altitude;
callbackReg.RegistrationContext = NULL;
callbackReg.OperationRegistration = opReg;

NTSTATUS status = ObRegisterCallbacks(&callbackReg, &gCallbackHandle);
```

The pre-operation callback receives a `POB_PRE_OPERATION_INFORMATION` structure. The critical field is `Parameters->CreateHandleInformation.DesiredAccess`. The callback can strip access rights from the desired access by modifying `Parameters->CreateHandleInformation.DesiredAccess` before the handle is created. This is how anti-cheats prevent external processes from opening handles to the game process with `PROCESS_VM_READ` or `PROCESS_VM_WRITE` access.

When a cheat calls `OpenProcess(PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, gameProcessId)`, the anti-cheat's `ObRegisterCallbacks` pre-operation callback fires. The callback checks whether the target process is the protected game process. If it is, it strips `PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `PROCESS_VM_OPERATION`, and `PROCESS_DUP_HANDLE` from the desired access. The cheat receives a handle, but the handle is useless for reading or writing game memory. The cheat's `ReadProcessMemory` call will fail with `ERROR_ACCESS_DENIED`.

The IRQL for `ObRegisterCallbacks` pre-operation callbacks is `PASSIVE_LEVEL`, which means the callback can call pageable code and perform blocking operations (within reason).



ObCallbackDemo.sys in action. DebugView shows the driver stripping handle access rights, Target.exe is running

with secret data in memory, and Verifier.exe fails to read it with Access Denied.

PsSetCreateProcessNotifyRoutineEx

PsSetCreateProcessNotifyRoutineEx allows a driver to register a callback that fires on every process creation and termination event system-wide. The callback receives a `PEPROCESS` for the process, the PID, and a PPS_CREATE_NOTIFY_INFO structure containing details about the process being created (image name, command line, parent PID).

```
VOID ProcessNotifyCallback(
    PEPROCESS Process,
    HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo
)
{
    if (CreateInfo == NULL) {
        // Process is terminating
        HandleProcessTermination(Process, ProcessId);
        return;
    }

    // Process is being created
    if (CreateInfo->ImageFileName != NULL) {
        // Check if this is a known cheat process
        if (IsKnownCheatProcess(CreateInfo->ImageFileName)) {
            // Set an error status to prevent the process from launch
            CreateInfo->CreationStatus = STATUS_ACCESS_DENIED;
        }
    }
}

PsSetCreateProcessNotifyRoutineEx(ProcessNotifyCallback, FALSE);
```

Notably, the `Ex` variant (introduced in Windows Vista SP1) provides the image file name and command line, which the original PsSetCreateProcessNotifyRoutine does not. The callback is called at `PASSIVE_LEVEL` from a system thread context.

Anti-cheats use this callback to detect cheat tool processes spawning on the system. If a known cheat launcher or injector process is created while the game is running, the anti-cheat can immediately flag this. Some implementations also set `CreateInfo->CreationStatus` to a failure code to outright prevent the process from launching.

PsSetCreateThreadNotifyRoutine

PsSetCreateThreadNotifyRoutine fires on every thread creation and termination system-wide. Anti-cheats use it specifically to detect thread creation in the protected game process. When a new thread is created in the game process, the callback fires and the anti-cheat can inspect the thread's start address.

```
VOID ThreadNotifyCallback(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create)
{
    if (!Create) return;

    if (IsProtectedProcess(ProcessId)) {
        PETHREAD Thread;
        PsLookupThreadByThreadId(ThreadId, &Thread);

        // Get the thread start address - this is stored in ETHREAD
        PVOID StartAddress = PsGetThreadWin32StartAddress(Thread);

        // Check if start address is within a known module
        if (!IsAddressInKnownModule(StartAddress)) {
            // Thread started at an address with no backing module - suspici
            FlagSuspiciousThread(Thread, StartAddress);
        }

        ObDereferenceObject(Thread);
    }
}
```

The call to `PsLookupThreadByThreadId` retrieves the `ETHREAD` pointer for the new thread. `PsGetThreadWin32StartAddress` returns the Win32 start address as seen by the process, which is distinct from the kernel-internal start address. Once finished with the thread object, `ObDereferenceObject` releases the reference acquired by `PsLookupThreadByThreadId`.

A thread created in the game process whose start address does not fall within any loaded module's address range is a strong indicator of injected code. Legitimate threads start inside module code. An injected thread typically starts in shellcode or manually mapped PE code that has no module backing.

PsSetLoadImageNotifyRoutine

`PsSetLoadImageNotifyRoutine` fires whenever an image (DLL or EXE) is mapped into any process. It provides the image file name and a `PIMAGE_INFO` structure containing the base address and size.

```
VOID LoadImageCallback(
    PUNICODE_STRING FullImageName,
    HANDLE ProcessId,
    PIMAGE_INFO ImageInfo
)
{
    if (IsProtectedProcess(ProcessId)) {
        // A DLL was loaded into the protected game process
        // Verify it's on the allowlist or check its signature
        if (!IsAllowedModule(FullImageName)) {
            // Log and potentially act on this
            ReportSuspiciousModule(FullImageName, ImageInfo->ImageBas
        }
    }
}
```

This is IRQL `PASSIVE_LEVEL`. The callback fires after the image is mapped but before its entry point executes, which gives the anti-cheat an opportunity to scan the image before any of its code runs.

CmRegisterCallbackEx

CmRegisterCallbackEx registers a callback for registry operations. Anti-cheats use this to monitor for registry modifications that might indicate cheats configuring themselves or attempting to modify anti-cheat settings.

```
NTSTATUS RegistryCallback(
    PVOID CallbackContext,
    PVOID Argument1,  // REG_NOTIFY_CLASS
    PVOID Argument2   // Operation-specific data
)
{
    REG_NOTIFY_CLASS notifyClass = (REG_NOTIFY_CLASS)(ULONG_PTR)Argumen

    if (notifyClass == RegNtPreSetValueKey) {
        PREG_SET_VALUE_KEY_INFORMATION info =
            (PREG_SET_VALUE_KEY_INFORMATION)Argument2;
        // Check if someone is modifying anti-cheat registry keys
        if (IsProtectedRegistryKey(info->Object)) {
            return STATUS_ACCESS_DENIED;
        }
    }
    return STATUS_SUCCESS;
}
```

MiniFilter Drivers for Filesystem Monitoring

A minifilter driver sits in the file system filter stack and intercepts IRP requests going to and from file system drivers. Anti-cheats use minifilters to monitor for cheat file drops (writing known cheat executables or DLLs to disk), to detect reads of their own driver files (which might indicate attempts to patch the on-disk driver binary before it is verified), and to enforce file access restrictions.

```
FLT_PREOP_CALLBACK_STATUS PreOperationCallback(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
)
{
    if (Data->Iopb->MajorFunction == IRP_MJ_WRITE) {
        // Check if the target file is a known cheat file name
        PFLT_FILE_NAME_INFORMATION nameInfo;
        FltGetFileNameInformation(Data, FLT_FILE_NAME_NORMALIZED, &nameInf

        if (IsKnownCheatFileName(&nameInfo->Name)) {
            Data->IoStatus.Status = STATUS_ACCESS_DENIED;
            FltReleaseFileNameInformation(nameInfo);
            return FLT_PREOP_COMPLETE;
        }
        FltReleaseFileNameInformation(nameInfo);
    }
    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

FltGetFileNameInformation retrieves the normalized file name for the target of the operation. FltReleaseFileNameInformation must be called to release the reference when done. Minifilter callbacks typically run at `APC_LEVEL` or `PASSIVE_LEVEL`, depending on the operation and the file system. This is important because many operations (like allocating paged pool or calling pageable functions) are not safe at `DISPATCH_LEVEL` or above.

4. Memory Protection and Scanning

The kernel driver can do far more than just register callbacks. It can actively scan the game process's memory and the system-wide memory pool for artifacts of cheats.

Blocking Handle Access

As covered in the `ObRegisterCallbacks` section, the primary mechanism for protecting game memory from external reads and writes is stripping `PROCESS_VM_READ` and `PROCESS_VM_WRITE` from handles opened to the game process. This is effective against any cheat that uses standard Win32 APIs (`ReadProcessMemory`, `WriteProcessMemory`) because these ultimately call `NtReadVirtualMemory` and `NtWriteVirtualMemory`, which require appropriate handle access rights.

However, a kernel-mode cheat can bypass this entirely. It can call `MmCopyVirtualMemory` directly (an unexported but locatable kernel function) or manipulate page table entries directly to access game memory without going through the handle-based access control system. This is why handle protection alone is insufficient and why kernel-level cheats require kernel-level anti-cheat responses.

Periodic Memory Integrity Hashing

Anti-cheats periodically hash the code sections (`.text` sections) of the game executable and its core DLLs. A baseline hash is computed at game start, and periodic re-hashes are compared against the baseline. If the hash changes, someone has written to game code, which is a strong indicator of code patching (commonly used to enable no-recoil, speed, or aimbot functionality by patching game logic).

```
// Pseudocode for code section integrity checking
BOOLEAN VerifyCodeSectionIntegrity(PEPROCESS Process, PVOID ModuleBase)
{
    // Attach to process context to read its memory
    KAPC_STATE apcState;
    KeStackAttachProcess(Process, &apcState);

    // Parse PE headers to find .text section
    PIMAGE_NT_HEADERS ntHeaders = RtlImageNtHeader(ModuleBase);
    PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(ntHeaders);

    for (USHORT i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++, section++) {
        if (memcmp(section->Name, ".text", 5) == 0) {
            PVOID sectionBase = (PVOID)((ULONG_PTR)ModuleBase + section->VirtualAddres
            ULONG sectionSize = section->Misc.VirtualSize;

            // Compute hash of current code section contents
            UCHAR currentHash[32];
            ComputeSHA256(sectionBase, sectionSize, currentHash);

            // Compare against stored baseline hash
            if (memcmp(currentHash, gBaselineHash, 32) != 0) {
                KeUnstackDetachProcess(&apcState);
                return FALSE; // Code modification detected
            }
        }
    }

    KeUnstackDetachProcess(&apcState);
    return TRUE;
}
```

The KeStackAttachProcess / KeUnstackDetachProcess pattern is used to temporarily attach the calling thread to the target process's address space, allowing the driver to read memory that is mapped into the game process without going through handle-based access controls. RtlImageNtHeader parses the PE headers from the in-memory image base.

Heuristic Scanning: Detecting Manually Mapped Code

The most interesting memory scanning is the heuristic detection of manually mapped code. When a legitimate DLL loads, it appears in the process's PEB module list, in the `InLoadOrderModuleList`, and has a corresponding `VAD_NODE` entry with a `MemoryAreaType` that indicates the mapping came from a file. Manual mapping bypasses the normal loader, so the mapped code appears in memory as an anonymous private mapping or as a file-backed mapping with suspicious characteristics.

The key heuristic is: find all executable memory regions in the process, then cross-reference each one against the list of loaded modules. Executable memory that does not correspond to any loaded module is suspicious.

```
// Walk the VAD tree to find executable anonymous mappings
VOID ScanForManuallyMappedCode(PEPROCESS Process)
{
    KAPC_STATE apcState;
    KeStackAttachProcess(Process, &apcState);

    PVOID baseAddress = NULL;
    MEMORY_BASIC_INFORMATION mbi;

    while (NT_SUCCESS(ZwQueryVirtualMemory(
        ZwCurrentProcess(),
        baseAddress,
        MemoryBasicInformation,
        &mbi,
        sizeof(mbi),
        NULL)))
    {
        if (mbi.State == MEM_COMMIT &&
            (mbi.Protect & PAGE_EXECUTE_READ ||
             mbi.Protect & PAGE_EXECUTE_READWRITE ||
             mbi.Protect & PAGE_EXECUTE_WRITECOPY) &&
            mbi.Type == MEM_PRIVATE)  // Private, not file-backed
        {
            // Executable private memory - not associated with any file mapping
            // This is a strong indicator of manually mapped or shellcode
            ReportSuspiciousRegion(mbi.BaseAddress, mbi.RegionSize,
                                   "Executable private memory without file backing
        }

        baseAddress = (PVOID)((ULONG_PTR)mbi.BaseAddress + mbi.RegionSize);
        if ((ULONG_PTR)baseAddress >= 0x7FFFFFFFFFFF) break; // User space limit
    }

    KeUnstackDetachProcess(&apcState);
}
```

ZwQueryVirtualMemory iterates through committed memory regions, returning a MEMORY_BASIC_INFORMATION structure for each. The `Type` field distinguishes private allocations (`MEM_PRIVATE`) from file-backed mappings (`MEM_IMAGE`, `MEM_MAPPED`). BattlEye's scanning approach, as documented by the secret.club and back.engineering analyses, involves scanning all memory regions of the protected process and specifically flagging executable regions without file backing. It also scans external processes' memory pages looking for execution bit anomalies,

specifically targeting cases where page protection flags have been changed programmatically to make otherwise non-executable memory executable (a common technique when shellcode is staged).

VAD Tree Walking

The VAD (Virtual Address Descriptor) tree is a kernel-internal structure that the memory manager uses to track all memory regions allocated in a process. Each `VAD_NODE` (which is actually a `MMVAD` structure in kernel terms) contains information about the region: its base address and size, its protection, whether it is file-backed (and if so, which file), and various flags.

Anti-cheats walk the VAD tree directly rather than relying on `ZwQueryVirtualMemory`, because the VAD tree cannot be trivially hidden from kernel mode in the same way that module lists can be manipulated. Walking the VAD:

```
// Simplified VAD walker - actual offsets are version-specific
VOID WalkVAD(PEPROCESS Process)
{
    // VadRoot is at a version-specific offset in EPROCESS
    // On Windows 11 23H2, this is at EPROCESS+0x7D8 (https://www.vergiliusproject.com/kernels/x64/windows-11/23h2/_EPROCE
    PMM_AVL_TABLE vadRoot = (PMM_AVL_TABLE)((ULONG_PTR)Process +
                                            EPROCESS_VAD_ROOT_OFFSET);

    WalkAVLTree(vadRoot->BalancedRoot.RightChild);
}

VOID WalkAVLTree(PMMADDRESS_NODE node)
{
    if (node == NULL) return;

    PMMVAD vad = (PMMVAD)node;

    // Check the VAD flags for suspicious characteristics
    // u.VadFlags.PrivateMemory = 1 and executable protection = suspicious
    if (vad->u.VadFlags.PrivateMemory && IsExecutableProtection(vad->u.VadFlags.Protection)) {
        // Check for file-backed backing
        if (vad->Subsection == NULL) {
            ReportSuspiciousVAD(vad);
        }
    }

    WalkAVLTree(node->LeftChild);
    WalkAVLTree(node->RightChild);
}
```

We can observe this detection in practice using WinDbg's `!vad` command on a process with injected code.

```
1: kd> !process 0 0 Target.exe
PROCESS ffffe38ab722a080
    SessionId: 1  Cid: 12c0    Peb: 90851d3000  ParentCid: 03e4
    DirBase: 1279e000  ObjectTable: ffff840c185020c0  HandleCount:  51.
    Image: Target.exe

1: kd> .process /i ffffe38ab722a080
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
1: kd> !vad
VAD             Level         Start             End              Commit
ffffe38ab91c8730  5             310             310                1 Private      EXECUTE_READWRITE
ffffe38ab90031c0  4            7ffe0           7ffe0               1 Private      READONLY
ffffe38ab90088a0  3            7ffe4           7ffe4               1 Private      READONLY
ffffe38ab9008cb0  4         9084ed0         9084fcf               6 Private      READWRITE
ffffe38ab9008990  2         9085000         90851ff               3 Private      READWRITE
ffffe38ab8d63b00  3        2826f310        2826f31f               0 Mapped       READWRITE          Pagefile section, shared commit 0x10
ffffe38ab8d64aa0  4        2826f320        2826f322               0 Mapped       READONLY           \Windows\System32\l_intl.nls
ffffe38ab8d63ce0  1        2826f330        2826f34e               0 Mapped       READONLY           Pagefile section, shared commit 0x1f
ffffe38ab8d64140  4        2826f350        2826f353               0 Mapped       READONLY           Pagefile section, shared commit 0x4
ffffe38ab8d640a0  3        2826f360        2826f360               0 Mapped       READONLY           Pagefile section, shared commit 0x1
ffffe38ab9008e40  4        2826f370        2826f371               2 Private      READWRITE
ffffe38ab8d641e0  2        2826f380        2826f390               0 Mapped       READONLY           \Windows\System32\C_1252.NLS
ffffe38ab8d64b40  3        2826f3a0        2826f3b0               0 Mapped       READONLY           \Windows\System32\C_437.NLS
```

The first entry is a Private EXECUTE_READWRITE region with no backing file, injected by our test tool. Every legitimate module shows as Mapped Exe with a full file path.

The power of VAD walking is that it catches manually mapped code even if the cheat has manipulated the PEB module list or the `LDR_DATA_TABLE_ENTRY` chain to hide itself. The VAD is a kernel structure that usermode code cannot modify directly.

5. Anti-Injection Detection

CreateRemoteThread Injection

The classic injection technique: call `CreateRemoteThread` in the target process with `LoadLibraryA` as the thread start address and the DLL path as the argument. This is trivially detectable via `PsSetCreateThreadNotifyRoutine`: the new thread's start address will be `LoadLibraryA` (or rather its address in kernel32.dll), and the caller process is not the game itself.

A more subtle check is the `CLIENT_ID` of the creating thread. When `CreateRemoteThread` is called, the kernel records which process created the thread. The anti-cheat can check whether a thread in the game process was created by an external process, which is a reliable indicator of injection.

APC Injection

`QueueUserAPC` and the underlying `NtQueueApcThread` allow queuing an Asynchronous Procedure Call to a thread in any process for which the caller has `THREAD_SET_CONTEXT` access. When the target thread enters an alertable wait, the APC fires and executes arbitrary code in the target thread's context.

Detection at the kernel level leverages the `KAPC` structure. Each thread has a kernel APC queue and a user APC queue. Anti-cheats can inspect the pending APC queue of game process threads to detect suspicious APC targets:

```
// Check for suspicious pending APCs on a thread
VOID InspectThreadAPCQueue(PETHREAD Thread)
{
    // The user APC queue is at a version-specific offset in ETHREAD
    // ETHREAD::ApcState::ApcListHead[1] (index 1 = user APC list)
    PLIST_ENTRY apcList = (PLIST_ENTRY)((ULONG_PTR)Thread +
                                        ETHREAD_APC_STATE_OFFSET +
                                        KAPC_STATE_USER_APC_LIST_OFFSE

    PLIST_ENTRY entry = apcList->Flink;
    while (entry != apcList) {
        PKAPC apc = CONTAINING_RECORD(entry, KAPC, ApcListEntry);

        // Check if the normal routine (user APC function)
        // points to an address without module backing
        if (apc->NormalRoutine != NULL &&
            !IsAddressInLoadedModule((PVOID)apc->NormalRoutine)) {
            ReportSuspiciousAPC(Thread, apc->NormalRoutine);
        }

        entry = entry->Flink;
    }
}
```

NtMapViewOfSection-Based Injection

A sophisticated injection technique maps a shared section object (backed by a file or created with `NtCreateSection`) into the target process using `NtMapViewOfSection`. This bypasses `CreateRemoteThread`-based detection heuristics because no remote thread is created initially. The injected code is then typically triggered via APC or by modifying an existing thread's context.

Detection is via the VAD: a section mapping that appears in the game process but was created by an external process will have a distinct pattern in the VAD. Specifically, the `MMVAD::u.VadFlags.NoChange` and related flags, combined with the file object backing the section (or lack thereof), can reveal this technique.

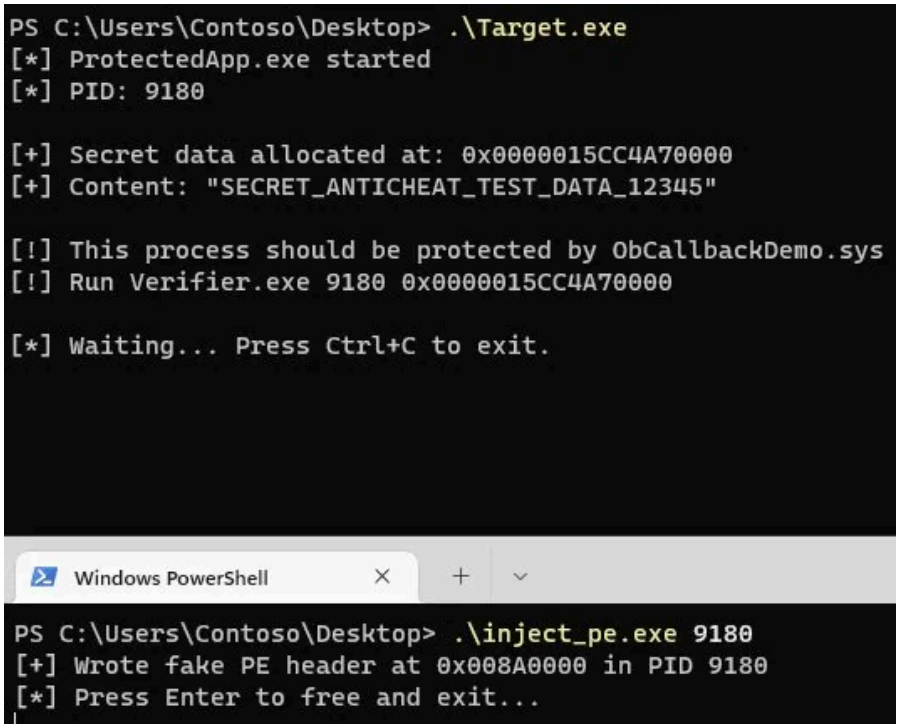
Reflective DLL Injection and Manual Mapping

Reflective DLL injection embeds a reflective loader inside the DLL that, when executed, maps the DLL into memory without using `LoadLibrary`. The DLL parses its own PE headers, resolves imports, applies relocations, and calls `DllMain`. The result is a fully functional DLL in memory that never appears in the `InLoadOrderModuleList`.

Detection: executable memory with a valid PE header (check for the `MZ` magic bytes and the `PE\0\0` signature at the offset specified by `e_lfanew`) but no corresponding module list entry. This is a reliable indicator.

```
// Check for PE headers in executable private memory
BOOLEAN HasValidPEHeader(PVOID base, SIZE_T size)
{
    if (size < sizeof(IMAGE_DOS_HEADER)) return FALSE;

    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)base;
    if (dosHeader->e_magic != IMAGE_DOS_SIGNATURE) return FALSE;

    if (dosHeader->e_lfanew >= size - sizeof(IMAGE_NT_HEADERS)) return FAL

    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)
        ((ULONG_PTR)base + dosHeader->e_lfanew);
    if (ntHeaders->Signature != IMAGE_NT_SIGNATURE) return FALSE;

    return TRUE;
}
```

We can observe this in practice using a simple test tool that allocates an RWX region and writes a minimal PE header into a running process:



Walking the VAD tree with `!vad` reveals the injected region immediately. The first entry at `0x8A0` is a `Private EXECUTE_READWRITE` region with no backing file. Compare this to the legitimate `Target.exe` image at the bottom, which is `Mapped Exe EXECUTE_WRITECOPY` with a full file path. Dumping the legitimate module's base with `db` confirms a complete PE header with the DOS stub:

```
1: kd> !vad
VAD            Level         Start             End              Commit
ffffe38ab7fe3c50  5             8a0               8a0              1 Private      EXECUTE_READWRITE
ffffe38ab8445120  4          7ffe0             7ffe0              1 Private      READONLY
ffffe38ab8445350  3          7ffe4             7ffe4              1 Private      READONLY
ffffe38ab84454e0  4         5babe70           5babf6f             6 Private      READWRITE
ffffe38ab8444e00  2         5bac000           5bac1ff             3 Private      READWRITE
ffffe38ab71e9b10  3        15cc48a0          15cc48af             0 Mapped       READWRITE          Pagefile section, shared commit 0x10
ffffe38ab8d63880  4        15cc48b0          15cc48b2             0 Mapped       READONLY           \Windows\System32\l_intl.nls
ffffe38ab71d34f0  1        15cc48c0          15cc48de             0 Mapped       READONLY           Pagefile section, shared commit 0x1f
ffffe38ab71d4cb0  4        15cc48e0          15cc48e3             0 Mapped       READONLY           Pagefile section, shared commit 0x4
ffffe38ab71e9d90  3        15cc48f0          15cc48f0             0 Mapped       READONLY           Pagefile section, shared commit 0x1
ffffe38ab8445210  4        15cc4900          15cc4901             2 Private      READWRITE
ffffe38ab71e97f0  2        15cc4910          15cc4920             0 Mapped       READONLY           \Windows\System32\C_1252.NLS
ffffe38ab71e9930  3        15cc4930          15cc4940             0 Mapped       READONLY           \Windows\System32\C_437.NLS
ffffe38ab71e9a70  0        15cc4950          15cc4952             0 Mapped       READONLY           \Windows\System32\l_intl.nls
ffffe38ab8444ef0  4        15cc4960          15cc4979             1 Private      READWRITE
ffffe38ab8d63100  3        15cc4980          15cc4a4d             0 Mapped       READONLY           \Windows\System32\locale.nls
ffffe38ab8d646e0  4        15cc4a50          15cc4a60             0 Mapped       READONLY           \Windows\System32\C_1252.NLS
ffffe38ab8445440  5        15cc4a70          15cc4a70             1 Private      READWRITE
ffffe38ab8444c70  2        15cc4a80          15cc4b7f            15 Private      READWRITE
ffffe38ab8d641e0  5        15cc4b80          15cc4b90             0 Mapped       READONLY           \Windows\System32\C_437.NLS
ffffe38ab8d5b7c0  4      7ff419110         7ff41920f              0 Mapped       READONLY           Pagefile section, shared commit 0x5
ffffe38ab84455d0  3      7ff419210         7ff51922f              0 Private      READWRITE
ffffe38ab8445170  4      7ff519230         7ff51b230              1 Private      READWRITE
ffffe38ab7905b40  1      7ff51b240         7ff51b240              0 Mapped       READONLY           Pagefile section, shared commit 0x1
ffffe38ab7905280  3      7ff6a7a50         7ff6a7a77              4 Mapped  Exe  EXECUTE_WRITECOPY  \Users\Contoso\Desktop\Target.exe
ffffe38ab71eb410  4      7ffc61590         7ffc61935              9 Mapped  Exe  EXECUTE_WRITECOPY  \Windows\System32\KernelBase.dll
ffffe38ab71e9e30  2      7ffc634a0         7ffc63563              7 Mapped  Exe  EXECUTE_WRITECOPY  \Windows\System32\kernel32.dll
ffffe38ab7905460  3      7ffc63e70         7ffc64086             16 Mapped  Exe  EXECUTE_WRITECOPY  \Windows\System32\ntdll.dll

Total VADs: 28, average level: 4, maximum depth: 5
Total private commit: 0x44 pages (272 KB)
Total shared commit:  0x3a pages (232 KB)
1: kd> db 7ff6a7a50*0x1000
00007ff6`a7a50000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
00007ff6`a7a50010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
00007ff6`a7a50020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00007ff6`a7a50030  00 00 00 00 00 00 00 00-00 00 00 00 08 01 00 00  ................
00007ff6`a7a50040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
00007ff6`a7a50050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
00007ff6`a7a50060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
00007ff6`a7a50070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

Dumping the injected region at `0x008A0000` also shows a valid MZ signature, but the rest of the header is mostly zeroes with no DOS stub. This is characteristic of manually mapped code:

```
0: kd> .process /i ffffe38ab8fca0c0
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
0: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff807`2081c720 cc              int     3
1: kd> db 0x008A0000
00000000`008a0000  4d 5a 00 00 00 00 00 00-00 00 00 00 00 00 00 00  MZ..............
00000000`008a0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`008a0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`008a0030  00 00 00 00 00 00 00 00-00 00 00 00 80 00 00 00  ................
00000000`008a0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`008a0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`008a0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`008a0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Finally, `!peb` confirms that the injected region does not appear in any of the module lists. The PEB only contains `Target.exe`, `ntdll.dll`, `kernel32.dll`, and `KernelBase.dll`. The region at `0x008A0000` is completely invisible to any usermode API that enumerates loaded modules:

```
1: kd> !peb
PEB at 0000005bac0a7000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00007ff6a7a50000
    NtGlobalFlag:             0
    NtGlobalFlag2:            0
    Ldr                       00007ffc63ff63e0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 0000015cc4a82230 . 0000015cc4a870d0
    Ldr.InLoadOrderModuleList:           0000015cc4a823b0 . 0000015cc4a876a0
    Ldr.InMemoryOrderModuleList:         0000015cc4a823c0 . 0000015cc4a876b0
                    Base TimeStamp                     Module
            7ff6a7a50000 699cd216 Feb 23 14:17:58 2026 C:\Users\Contoso\Desktop\Target.exe
            7ffc63e70000 806d3ecf Apr 11 16:59:11 2038 C:\Windows\SYSTEM32\ntdll.dll
            7ffc634a0000 d05c34f8 Oct 09 05:50:00 2080 C:\Windows\System32\KERNEL32.DLL
            7ffc61590000 9223bda8 Sep 11 04:57:28 2047 C:\Windows\System32\KERNELBASE.dll
    SubSystemData:     0000000000000000
    ProcessHeap:       0000015cc4a80000
    ProcessParameters: 0000015cc4a86730
```

Stack Walking with RtlWalkFrameChain

When BEDaisy wants to inspect a thread's call stack, it uses an APC mechanism to capture the stack frames while the thread is in user mode. The APC fires in the game thread's context and calls `RtlWalkFrameChain` or `RtlCaptureStackBackTrace` to capture the return address chain.

The back.engineering analysis of BEDaisy (and the Aki2k/BEDaisy GitHub research) documents this specifically: BEDaisy queues kernel APCs to threads in the protected process. The APC kernel routine runs at `APC_LEVEL`, captures the thread's stack, and then analyzes each return address against the list of loaded modules. A return address pointing outside any loaded module is a strong indicator of injected code on the stack, which suggests the thread is currently executing injected code or returned from it.

```
// Pseudocode for APC-based stack inspection
VOID KernelApcRoutine(
    PKAPC Apc,
    PKNORMAL_ROUTINE *NormalRoutine,
    PVOID *NormalContext,
    PVOID *SystemArgument1,
    PVOID *SystemArgument2
)
{
    // We're now running at APC_LEVEL in the context of the inspected thr
    PVOID frames[64];
    ULONG capturedFrames = RtlWalkFrameChain(frames, 64, 0);

    for (ULONG i = 0; i < capturedFrames; i++) {
        if (!IsAddressInKnownModule(frames[i])) {
            // Stack frame points outside any loaded module
            ReportSuspiciousStackFrame(frames[i]);
        }
    }
}
```

6. Hook Detection

Hooks are the primary mechanism by which usermode cheats intercept and manipulate the game's interaction with the OS. Detecting them is a core anti-cheat function.

IAT Hook Detection

The Import Address Table (IAT) of a PE file contains the addresses of imported functions. When a process loads, the loader resolves these addresses by looking up each imported function in the exporting DLL and writing the function's address into the IAT. An IAT hook overwrites one of these entries with a pointer to attacker-controlled code.

Detection is straightforward: for each IAT entry, compare the resolved address against what the on-disk export of the correct DLL says the address should be.

```
VOID DetectIATHooks(PVOID moduleBase)
{
    PIMAGE_IMPORT_DESCRIPTOR importDesc =
        RtlImageDirectoryEntryToData(moduleBase, TRUE,
                                     IMAGE_DIRECTORY_ENTRY_IMPORT, NULL);

    while (importDesc->Name != 0) {
        PCHAR dllName = (PCHAR)((ULONG_PTR)moduleBase + importDesc->Name);
        PVOID importedDllBase = GetLoadedModuleBase(dllName);

        PULONG_PTR iat = (PULONG_PTR)((ULONG_PTR)moduleBase +
                                      importDesc->FirstThunk);
        PIMAGE_THUNK_DATA originalFirstThunk =
            (PIMAGE_THUNK_DATA)((ULONG_PTR)moduleBase +
                                importDesc->OriginalFirstThunk);

        while (originalFirstThunk->u1.AddressOfData != 0) {
            PIMAGE_IMPORT_BY_NAME importByName =
                (PIMAGE_IMPORT_BY_NAME)((ULONG_PTR)moduleBase +
                                        originalFirstThunk->u1.AddressOfData)

            // Get the expected address from the exporting DLL
            PVOID expectedAddr = GetExportedFunctionAddress(importedDllBase,
                                                            importByName->Nam
            PVOID actualAddr = (PVOID)*iat;

            if (expectedAddr != actualAddr) {
                ReportIATHook(dllName, importByName->Name,
                              expectedAddr, actualAddr);
            }

            iat++;
            originalFirstThunk++;
        }
        importDesc++;
    }
}
```

`RtlImageDirectoryEntryToData` locates the import directory from the PE headers. The `TRUE` parameter specifies that the image is mapped (as opposed to a raw file on disk), which is correct when working with in-memory modules. The outer loop walks the `IMAGE_IMPORT_DESCRIPTOR` array, terminating on a zero `Name` field. The inner loop compares each resolved IAT entry against the expected export address.

Inline Hook Detection

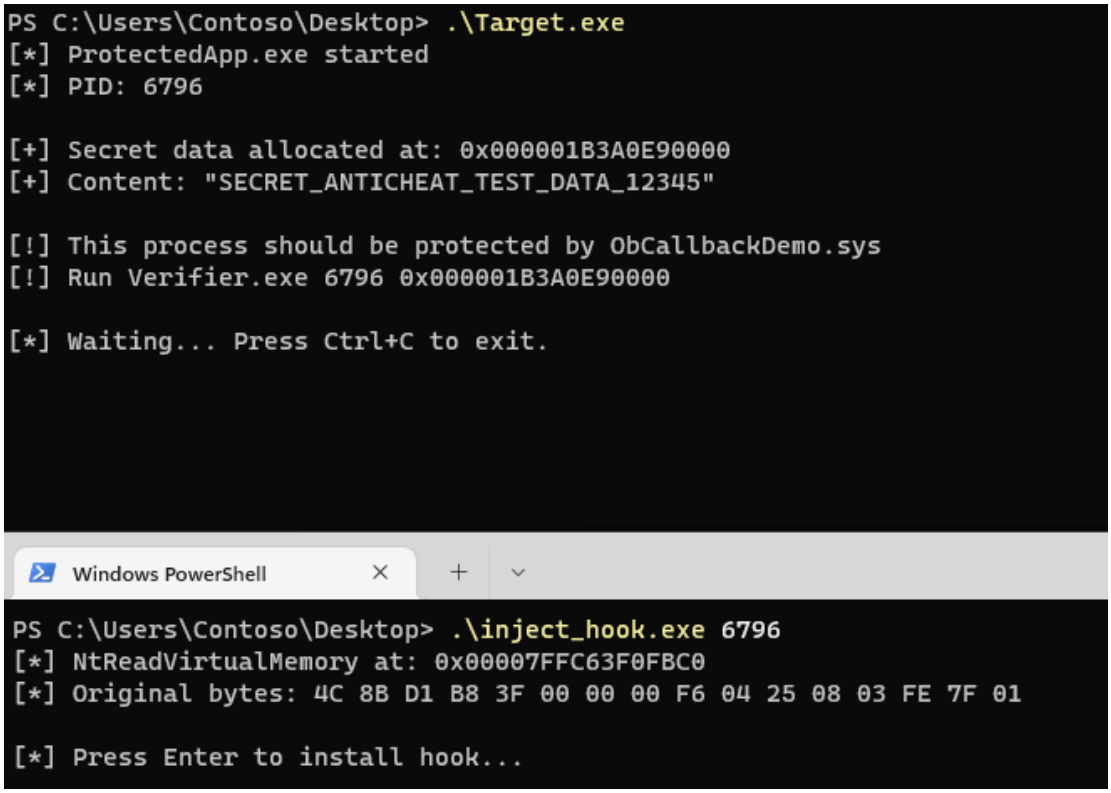
Inline hooks patch the first few bytes of a function with a `JMP` (opcode `0xE9` for relative near jump, or `0xFF 0x25` for indirect jump through a memory pointer) to redirect execution to attacker code, which typically performs its modifications and then jumps back to the original code (a "trampoline" pattern).

Detection involves reading the first 16-32 bytes of each monitored function and checking for:

- `0xE9` (JMP rel32)
- `0xFF 0x25` (JMP [rip+disp32]) - common for 64-bit hooks
- `0x48 0xB8 ... 0xFF 0xE0` (MOV RAX, imm64; JMP RAX) - an absolute 64-bit jump sequence
- `0xCC` (INT 3) - a software breakpoint, which can also be a hook point

The anti-cheat reads the on-disk PE file and compares the on-disk bytes of function prologues against what is currently in memory. Any discrepancy indicates patching.

To demonstrate inline hook detection, we use a test tool that patches `NtReadVirtualMemory` in a running process with a `MOV RAX; JMP RAX` hook:



Before patching, the function prologue shows a clean syscall stub. `mov r10, rcx` saves the first argument, `mov eax, 3Fh` loads the syscall number, and `syscall` transitions to kernel mode:

Clean NtReadVirtualMemory prologue

After the hook is installed, the first 12 bytes are overwritten with `mov rax, 0xDEADBEEFCAFEBABE; jmp rax`, redirecting execution to an attacker-controlled address. An anti-cheat comparing these bytes against the on-disk copy of ntdll would immediately flag the mismatch:

Hooked NtReadVirtualMemory

SSDT Integrity Checking

The System Service Descriptor Table (SSDT) is the kernel's dispatch table for syscalls. When a usermode process executes a syscall instruction, the kernel uses the syscall number (placed in EAX) to index into the SSDT and invoke the corresponding kernel function. Patching the SSDT redirects syscalls to attacker-controlled code.

SSDT hooking is a classic technique that became significantly harder after the introduction of PatchGuard (Kernel Patch Protection, KPP) in 64-bit Windows. PatchGuard monitors the SSDT (among many other structures) and triggers a `CRITICAL_STRUCTURE_CORRUPTION` bug check (0x109) if it detects modification. As a result, SSDT hooking is essentially dead in 64-bit Windows. However, anti-cheats still verify SSDT integrity as a defense in depth measure.

IDT and GDT Monitoring

The Interrupt Descriptor Table (IDT) maps interrupt vectors to their handler routines. The Global Descriptor Table (GDT) defines memory segments. Both are processor-level structures that cannot be easily protected by PatchGuard alone on all configurations.

A cheat operating at kernel level can attempt to replace IDT entries to intercept specific interrupts, which can be used for control flow interception or as a covert channel. Anti-cheats verify that IDT entries point to expected kernel locations:

```
VOID VerifyIDTIntegrity(void)
{
    IDTR idtr;
    __sidt(&idtr);  // Read IDTR register

    PIDT_ENTRY64 idt = (PIDT_ENTRY64)idtr.Base;

    for (int i = 0; i < 256; i++) {
        ULONG_PTR handler = GetIDTHandlerAddress(&idt[i]);

        // Verify handler is in ntoskrnl or a known driver address ra
        if (!IsKernelCodeAddress(handler)) {
            ReportIDTModification(i, handler);
        }
    }
}
```

Detecting Direct Syscall Usage by Cheats

A common evasion technique is for cheats to perform syscalls directly (using the `syscall` instruction with the appropriate syscall number) rather than going through `ntdll.dll` functions. This bypasses usermode hooks placed in ntdll. Anti-cheats detect this by monitoring threads within the game process for `syscall` instruction execution from unexpected code locations, and by checking whether ntdll functions that should be called are actually being called with expected frequency and patterns.

7. Driver-Level Protections

Detecting Unsigned and Test-Signed Drivers

On a properly configured Windows system with Secure Boot enabled, all kernel drivers must be signed by a certificate trusted by Microsoft. Test signing mode (enabled with `bcdedit /set testsigning on`) allows loading self-signed drivers and is a common development and cheat-deployment technique.

Anti-cheats detect test signing mode by reading the Windows boot configuration and by checking the kernel variable that reflects whether DSE is currently enforced. Some anti-cheats refuse to launch if test signing is enabled.

The `SeValidateImageHeader` and `SeValidateImageData` functions in the kernel validate driver signatures. Anti-cheats can inspect loaded driver objects and verify that their `IMAGE_INFO_EX` `ImageSignatureType` and `ImageSignatureLevel` fields reflect proper signing.

BYOVD Attacks

Bring Your Own Vulnerable Driver is the dominant technique for loading unsigned kernel code in 2024-2026. The attack works as follows:

1. The attacker finds a legitimate, signed driver with a vulnerability (typically a dangerous IOCTL handler that allows arbitrary kernel memory reads/writes, or that calls `MmMapIoSpace` with attacker-controlled parameters).
2. The attacker loads this legitimate driver (which passes DSE because it has a valid signature).
3. The attacker exploits the vulnerability in the legitimate driver to achieve arbitrary kernel code execution.
4. Using that kernel execution, the attacker disables DSE or directly maps their unsigned cheat driver.

Common BYOVD targets have included drivers from MSI, Gigabyte, ASUS, and various hardware vendors. These drivers often have IOCTL handlers that expose direct physical memory read/write capability, which is all an attacker needs.

Anti-Cheat Driver Blocklists

The primary defense against BYOVD is a blocklist of known-vulnerable drivers. The Microsoft Vulnerable Driver Blocklist (maintained in `DriverSiPolicy.p7b`) is built into Windows and distributed via Windows Update. Anti-cheats maintain their own, more aggressive blocklists.

Vanguard in particular is known for actively comparing the set of loaded drivers against its blocklist and refusing to allow the protected game to launch if a blocklisted driver is present. This is enforced because some BYOVD attacks involve loading the vulnerable driver and immediately using it before unloading it, so checking only at game launch with a pre-scan covers most cases.

PiDDBCache and PiDDBLock

This is one of the more interesting internals that kernel cheat developers and anti-cheat engineers both care deeply about.

`PiDDBCacheTable` is a kernel-internal AVL tree that caches information about previously loaded drivers. When a driver is loaded, the kernel stores an entry keyed by the driver's `TimeDateStamp` (from the PE header) and `SizeOfImage`. This cache is used to quickly look up whether a driver has been seen before. The structure is a `RTL_AVL_TABLE` protected by `PiDDBLock` (an `ERESOURCE` lock).

Cheat developers who manually map a driver without going through the normal load path try to erase or modify the corresponding `PiDDBCacheTable` entry to conceal that their driver was ever loaded. Anti-cheats detect this by:

1. Verifying the consistency of the `PiDDBCacheTable` - if a driver is in memory (found via pool tag scanning or other means) but has no `PiDDBCacheTable` entry, the entry was probably scrubbed.
2. Monitoring the `PiDDBLock` for unexpected acquisitions from non-kernel threads.
3. Comparing the timestamp/size combinations of all known loaded drivers against the `PiDDBCacheTable` entries.

```
// Locating PiDDBCacheTable (must locate via signature scan since not exported)
// This is version-specific and fragile; anti-cheats maintain multiple signatures
BOOLEAN FindPiDDBCacheTable(PVOID *TableAddress)
{
    // Pattern to locate PiDDBCacheTable in ntoskrnl
    // This is a simplified illustration - real implementations use robust pattern match
    PVOID ntoskrnl = GetKernelModuleBase("ntoskrnl.exe");
    PUCHAR pattern = "\x48\x8D\x0D"; // LEA RCX, [RIP+...]
    PVOID match = FindPattern(ntoskrnl, pattern, 3, NTOSKRNL_TEXT_RANGE);
    if (match) {
        // Extract the RIP-relative offset
        INT32 offset = *(INT32*)((ULONG_PTR)match + 3);
        *TableAddress = (PVOID)((ULONG_PTR)match + 7 + offset);
        return TRUE;
    }
    return FALSE;
}
```

Reversing PiCompareDDBCacheEntries

`PiDDBCacheTable` is not exported and `PiDDBCacheEntry` is not in public symbols. To interact with the cache, we need to reverse the entry structure. The compare routine is the best starting point since it directly accesses the fields used for ordering.


PiCompareDDBCacheEntries in Binary Ninja

The decompiled output reveals the structure layout. The function receives two `PiDDBCacheEntry` pointers and first compares their `DriverName` fields (a `UNICODE_STRING` at offset 0x10) using `RtlCompareUnicodeString`. If the names are equal and `TableContext` is non-zero, the entries are considered equal. Otherwise, it falls through to comparing the `TimeDateStamp` field (a `ULONG` at offset 0x20). This gives us the recovered structure:

```
struct PiDDBCacheEntry
{
    RTL_BALANCED_LINKS Links;    // 0x00 - AVL tree node pointers (0x20 bytes)
    UNICODE_STRING DriverName;   // 0x10 - driver filename (from compare routine offs
    ULONG TimeDateStamp;         // 0x20 - PE header timestamp (secondary sort key)
};
```

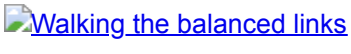
Walking the AVL Tree

`PiDDBCacheTable` is an `RTL_AVL_TABLE`, a self-balancing binary search tree. Each node in the tree starts with an `_RTL_BALANCED_LINKS` header containing `Parent`, `LeftChild`, and `RightChild` pointers. The actual `PiDDBCacheEntry` data sits immediately after this header.

To enumerate entries, we start by resolving the table address. `PiDDBCacheTable` is not exported, so anti-cheats locate it via signature scanning in `ntoskrnl.exe`. In WinDbg with symbols loaded, we can resolve it directly:


PiDDBCacheTable AVL table metadata

The table contains 151 cached driver entries with a tree depth of 9. The `CompareRoutine` points to `PiCompareDDBCacheEntries`, confirming this is the right table. The `BalancedRoot` is the entry point into the tree. Its `RightChild` gives us the first real node:


Walking the balanced links

From each node, the entry data starts at offset 0x20 (past the `_RTL_BALANCED_LINKS` header). Adding our recovered offsets: `DriverName` at node+0x30, `TimeDateStamp` at node+0x40. Following the `LeftChild` and `RightChild` pointers lets us walk the entire tree:


Cached driver entries

A cheat developer who manually maps a kernel driver would try to find and remove their entry from this tree to avoid detection. An anti-cheat that detects a driver in memory (via pool tag scanning or other means) but finds no corresponding `PiDDBCacheTable` entry knows the entry was scrubbed.

MmUnloadedDrivers

`MmUnloadedDrivers` is a kernel array (also not exported) that maintains a circular buffer of the last 50 unloaded drivers, storing their name, start address, end address, and unload timestamp. This structure allows debugging and forensics of driver activity.

Cheat developers who successfully load and then unload a kernel driver often try to zero out or corrupt their entry in `MmUnloadedDrivers` to hide traces. Anti-cheats detect this by:

1. Maintaining their own shadow copy of expected `MmUnloadedDrivers` entries.
2. Detecting anomalous zero-filled entries in the middle of the circular buffer (a signature of deliberate erasure).
3. Cross-referencing `MmUnloadedDrivers` against other kernel timestamps and logs.

BigPool Allocations

When a kernel allocation exceeds approximately 4KB (more precisely, when it exceeds a threshold managed by the pool allocator), it is managed as a “big pool allocation” tracked in the `PoolBigPageTable`. Anti-cheats scan this table to find memory allocations that were made by manually mapped drivers. A manually mapped driver typically makes large allocations for its code and data sections; these show up in the big pool table with the allocation address but without a corresponding loaded driver.

The technique is to enumerate all big pool entries, then cross-reference each allocation’s address against the list of loaded driver address ranges. Allocations in no driver’s range that are the right size to be driver code sections are suspicious.

8. Anti-Debug Protections

Anti-cheat code itself is a high-value target for reverse engineering. Reverse engineers analyzing the anti-cheat driver need to use kernel debuggers, which anti-cheats aggressively detect.

NtQueryInformationProcess Checks

At the usermode level (in the game-injected DLL), the anti-cheat uses `NtQueryInformationProcess` with multiple information classes:

- `ProcessDebugPort` (7): Returns a non-zero value if a debugger is attached via `DebugActiveProcess`. A kernel driver can spoof this by hooking `NtQueryInformationProcess`, but the check is done in the kernel driver itself as well.
- `ProcessDebugObjectHandle` (30): Returns a handle to the debug object if one exists.
- `ProcessDebugFlags` (31): The `NoDebugInherit` flag; checking for its inverse reveals debugger presence.

Kernel Debugger Detection

The kernel driver checks the kernel-exported variables `KdDebuggerEnabled` and `KdDebuggerNotPresent`. On a system with WinDbg (or any kernel debugger) attached, `KdDebuggerEnabled` is TRUE and `KdDebuggerNotPresent` is FALSE.

```
BOOLEAN IsKernelDebuggerPresent(void)
{
    // KD_DEBUGGER_ENABLED is a kernel export
    if (*KdDebuggerEnabled && !*KdDebuggerNotPresent) {
        return TRUE;
    }

    // Additional check: attempt a debug break and see if it's hand
    // More sophisticated: check specific kernel structures

    return FALSE;
}
```

Some anti-cheats go further and directly inspect the `KDDEBUGGER_DATA64` structure and the shared kernel data page (`KUSER_SHARED_DATA`) for debugger-related flags.

Thread Hiding Detection

`NtSetInformationThread` with `ThreadHideFromDebugger` (17) sets a flag in the thread's `ETHREAD` structure (`CrossThreadFlags.HideFromDebugger`). Once set, the kernel will not deliver debug events for that thread to any attached debugger. The thread becomes essentially invisible to WinDbg: breakpoints in the thread do not trigger debugger notification, exceptions are not forwarded.

Anti-cheats use this to protect their own threads. However, they also detect if cheats are using it to hide their own injected threads. The detection method is to enumerate all threads in the system via a kernel enumeration (not via usermode APIs that could be hooked) and check the `HideFromDebugger` bit in `CrossThreadFlags` for each thread. A hidden thread in the game process that the anti-cheat did not itself hide is a red flag.

```
// Check CrossThreadFlags for HideFromDebugger
#define PS_CROSS_THREAD_FLAGS_HIDEFROMDEBUGGER 0x4

VOID CheckThreadDebugVisibility(PETHREAD Thread)
{
    // CrossThreadFlags is at a version-specific offset in ETHREAD
    ULONG crossFlags = *(ULONG*)((ULONG_PTR)Thread + ETHREAD_CROSS_THREAD_FLAGS_OFFSE

    if (crossFlags & PS_CROSS_THREAD_FLAGS_HIDEFROMDEBUGGER) {
        // Thread is hidden from debuggers
        // If we didn't hide it, flag it
        if (!IsAntiCheatOwnedThread(Thread)) {
            ReportHiddenThread(Thread);
        }
    }
}
```

To demonstrate this detection, we use a test tool that creates a remote thread in Target.exe and then sets `ThreadHideFromDebugger` on it:

Creating a hidden thread

In WinDbg, we convert the decimal TID to hex, locate the thread in the process, and inspect its `CrossThreadFlags`. Before setting the flag, the value is `0x5402` with bit 2 (HideFromDebugger) clear:


CrossThreadFlags before

After calling `NtSetInformationThread` with `ThreadHideFromDebugger`, the value changes to `0x5406`. Bit 2 is now set, making this thread invisible to any attached debugger:


CrossThreadFlags after

An anti-cheat enumerating threads in the game process would check this bit on every thread. A hidden thread that the anti-cheat did not create itself is a strong indicator of injected cheat code.

Timing-Based Anti-Debug

Single-step debugging (via the TF flag in EFLAGS) and hardware breakpoints dramatically increase the time between instruction executions. Anti-cheats use `RDTSC` instruction-based timing to detect this:

```
UINT64 before = __rdtsc();
// Execute a fixed number of operations
volatile ULONG dummy = 0;
for (int i = 0; i < 1000; i++) dummy += i;
UINT64 after = __rdtsc();

UINT64 elapsed = after - before;
if (elapsed > EXPECTED_MAXIMUM_CYCLES) {
    // Execution was slowed - likely single-stepping or a breakpc
    ReportDebuggerDetected();
}
```

The threshold `EXPECTED_MAXIMUM_CYCLES` is calibrated based on known CPU behavior. Single-stepping can add thousands of cycles per instruction (due to debug exception handling), making the timing discrepancy obvious.

Hardware Breakpoint Detection

The x86-64 debug registers (DR0-DR3 for breakpoint addresses, DR6 for status, DR7 for control) are accessible in kernel mode. Reading them allows detection of hardware breakpoints set by a debugger:

```
BOOLEAN HasHardwareBreakpoints(void)
{
    ULONG_PTR dr7 = __readdr(7); // Read DR7 (debug control registe

    // Check Local Enable bits (L0, L1, L2, L3) for each breakpoint
    // Bits 0, 2, 4, 6 of DR7 are the local enable bits for BP 0-3
    if (dr7 & 0x55) { // 0x55 = 01010101b - all four local enable b
        return TRUE;
    }
    return FALSE;
}
```

Anti-cheats scan all threads' saved debug register state (accessible via `CONTEXT` structure obtained with `KeGetContextThread` or directly from `KTHREAD::TrapFrame`) for active hardware breakpoints not set by the anti-cheat itself.

Hypervisor-Based Debugger Detection

Type-1 hypervisor-based debuggers (like a custom hypervisor running a Windows VM for isolated debugging) are significantly harder to detect. The primary detection vectors are:

CPUID checks: The hypervisor present bit (bit 31 of ECX when CPUID leaf 1 is executed) indicates a hypervisor is present. The hypervisor vendor can be queried with CPUID leaf `0x40000000`. VMware returns "VMwareVMware", VirtualBox returns "VBoxVBoxVBox". An unknown vendor string is suspicious.

MSR timing: Executing `RDMSR` in a VM introduces additional overhead compared to native execution. Anti-cheats time MSR reads and flag anomalies.

CPUID instruction timing: The `CPUID` instruction itself is a privileged instruction in virtualized environments and must be handled by the hypervisor, introducing measurable latency.

9. DMA Cheats and Detection

DMA cheats represent the current frontier of the anti-cheat arms race, and they are genuinely hard to address with software alone.

What DMA Cheats Are

A PCIe DMA (Direct Memory Access) cheat uses a PCIe-connected device - typically a development FPGA board - that can directly read the host system's physical memory via the PCIe bus without the CPU being involved. The `pcileech` framework and its `LeechCore` library provide the software stack for these devices. The device physically appears on the PCIe bus, acquires access to the host's physical memory via the PCIe TLP (Transaction Layer Packet) protocol, and reads game process memory by translating virtual addresses to physical addresses (using the page tables, which are also in physical memory and can be read by the device).

The attacking machine (running the cheat software) is physically separate from the victim machine (running the game). All cheat logic runs on the attacker's machine. The game machine has no processes, no drivers, no memory allocations from the cheat. From a pure software perspective, the game machine is completely clean.

DMA cheat architecture *The FPGA sits on the game PC's PCIe bus and reads physical memory via TLPs without CPU involvement. The cheat software runs entirely on a separate machine, leaving the game PC clean from a software perspective.*

PCIe Internals

PCIe communication is structured around TLPs. A memory read TLP from a DMA device contains the physical address to read and the requested byte count. The PCIe root complex services this request by reading the specified physical memory and returning the data in a completion TLP. This is entirely hardware-level and the CPU is not involved in servicing the request.

The device needs to be configured with a valid BAR (Base Address Register) that the BIOS assigns during PCIe enumeration. The device also needs the target system's IOMMU (if present) to either be disabled or to have the device's DMA transactions allowed through it.

IOMMU as a Defense

The IOMMU (Intel VT-d, AMD-Vi) is a hardware unit that translates DMA addresses from PCIe devices using a device-specific page table (analogous to the CPU's page tables for usermode address translation). If the IOMMU is enabled and properly configured, a PCIe device can only access physical memory that the OS has explicitly granted to it via the IOMMU page tables.

With IOMMU enabled, a DMA device that is not associated with any OS driver has no IOMMU mappings and cannot access arbitrary physical memory. This is theoretically a hardware-level defense against DMA attacks.

In practice, the IOMMU defense has significant gaps. Many gaming motherboards ship with IOMMU disabled by default. Even when enabled, the IOMMU configuration is complex and many systems have misconfigured IOMMU policies that leave large physical memory ranges accessible. And critically, DMA firmware that successfully impersonates a legitimate PCIe device (e.g., a USB controller or a network card that the OS has granted IOMMU access to) can potentially access memory through the IOMMU using the legitimate device's granted permissions.

DMA Firmware Mimicry

Sophisticated DMA cheat firmware is designed to mimic the PCIe device ID, vendor ID, subsystem ID, and BAR0 configuration of a legitimate device. A Xilinx FPGA running customized firmware can present itself to the BIOS and OS as, for example, a USB host controller. The OS loads the legitimate driver for that device (providing IOMMU coverage), and the FPGA firmware uses that coverage to perform DMA reads.

Anti-cheats attempt to detect this by enumerating all PCIe devices and validating that each device's reported characteristics match expected hardware. But without specific firmware-level attestation, it is very difficult to distinguish legitimate hardware from a properly mimicking FPGA.

Secure Boot and TPM as Partial Mitigations

Epic Games' requirement for Secure Boot and TPM 2.0 for Fortnite is directly related to the DMA threat. Secure Boot ensures that only signed bootloaders run, which prevents boot-time attacks that could disable IOMMU or install firmware-level cheats. TPM 2.0 enables measured boot (where each boot stage's hash is recorded in the TPM's PCR registers), providing an attestation chain that proves the system booted in a known-good state. Remote attestation using the TPM can allow a server to verify that the client system has not been tampered with at the firmware level.

This does not solve the DMA problem directly (a DMA attack device physically attached to the PCIe slot bypasses all of this), but it closes some of the software-assisted DMA attack paths.

10. Behavioral Detection and Telemetry

No static protection scheme is sufficient. Behavioral detection operating on game telemetry is the complementary layer that addresses what kernel protections cannot.

Mouse and Input Analysis

The kernel anti-cheat driver operates at a level where it can intercept raw input before it reaches the game. Drivers for HID (Human Interface Device) input, specifically for mice and keyboards, sit in the input driver stack. By installing a filter driver above `mouclass.sys` or `kbdclass.sys`, an anti-cheat can observe all input events with timestamps accurate to the system clock (microsecond resolution).

Aimbot detection targets the statistical properties of mouse movement. Human aiming exhibits specific properties: Fitts' Law governs approach trajectories, there is characteristic deceleration as the cursor approaches a target, velocity profiles have specific acceleration and deceleration curves, and there is measurement noise. An aimbot performing perfect linear interpolation to a target produces movement that violates these properties. A triggerbot (which fires automatically when the crosshair is on target but does not manipulate mouse movement) is detected via reaction time analysis: human reaction times to a target crossing the crosshair have a minimum physiological floor (approximately 150-200ms) with a characteristic distribution. Reaction times below this floor with high consistency indicate automation.

Machine Learning Detection

The Collins et al. (CheckMATE 2024) paper documents the application of CNNs to triggerbot detection, achieving approximately 99.2% accuracy on labeled datasets. The features fed to the network include mouse position time series, click timing relative to target position, and velocity profiles.

The AntiCheatPT paper (2025) applies transformer architectures to aimbot detection. Using 256-tick windows with 44 data points per tick (including position, velocity, acceleration, view angle rates, and click events), the model achieves 89.17% accuracy in distinguishing legitimate players from aimbot users. The transformer architecture is well-suited to this problem because aimbots often introduce temporal correlations in the input data (smooth tracking, periodic corrections) that attention mechanisms can exploit.

Graph neural networks are used for collusion detection (wallhack and communication-based cheating in team games) by modeling player interaction graphs and detecting anomalous patterns, such as players consistently targeting enemies through walls or demonstrating perfect awareness of enemy positions without line-of-sight.

Mouse trajectory comparison *Left: a legitimate player's mouse path follows Fitts' Law with a natural S-curve, overshoot, and micro-corrections. Right: an aimbot produces an idle phase followed by an instant linear snap to the target with no natural deceleration.*

Telemetry Pipeline

The flow from raw data to ban decision typically works as follows:

1. The kernel driver captures input events and timestamps them at the hardware interrupt level.
2. These events are written to a shared memory ring buffer.
3. The usermode service reads from the ring buffer, batches events, encrypts them, and transmits to backend servers.
4. Backend ML inference runs on the event stream and produces anomaly scores.
5. A manual review queue receives high-confidence flagged sessions.
6. Ban decisions are pushed back to the game client via the service.

The encryption of telemetry data is critical both for privacy (the data includes all mouse movements) and for anti-tamper (preventing cheats from identifying and falsifying telemetry).

11. Anti-VM and Environment Checks

CPUID-Based VM Detection

The most reliable VM detection is CPUID-based. When `CPUID` is executed with `EAX=1`, bit 31 of `ECX` is set if a hypervisor is present (this is the “Hypervisor Present” bit). With `EAX=0x40000000`, the hypervisor vendor string is returned in EBX, ECX, EDX:

```
BOOLEAN IsRunningInVM(void)
{
    int cpuInfo[4];
    __cpuid(cpuInfo, 1);

    // Check hypervisor present bit (ECX bit 31)
    if (cpuInfo[2] & (1 << 31)) {
        // Get hypervisor vendor
        __cpuid(cpuInfo, 0x40000000);

        char vendor[13];
        memcpy(vendor, &cpuInfo[1], 4);
        memcpy(vendor + 4, &cpuInfo[2], 4);
        memcpy(vendor + 8, &cpuInfo[3], 4);
        vendor[12] = '\0';

        // Known VM vendors
        if (strcmp(vendor, "VMwareVMware") == 0 ||
            strcmp(vendor, "VBoxVBoxVBox") == 0 ||
            strcmp(vendor, "Microsoft Hv") == 0 ||  // Hype
            strcmp(vendor, "KVMKVMKVM") == 0) {
            return TRUE;
        }

        return TRUE; // Unknown hypervisor is also suspicic
    }
    return FALSE;
}
```

Artifact-Based VM Detection

Each VM platform leaves characteristic artifacts in the registry and device enumeration:

- VMware: Registry key `HKLM\SOFTWARE\VMware, Inc.\VMware Tools`; PCI device `\Device\VMwareHGFS`; virtual devices appearing in `Win32_PnPEntity` with “VMware” in the name.
- VirtualBox: `HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions`; `VBoxMiniRdDN` driver; registry key `HKLM\HARDWARE\ACPI\DSDT\VBOX__`.
- Hyper-V: `HKLM\SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters`; presence of `vmbus` and `storvsc` driver objects.

Anti-cheats query these artifacts from kernel mode where they cannot be intercepted by usermode hooking. A system presenting any of these artifacts is likely running in a VM, and the anti-cheat can refuse to operate or flag the session.

Detecting Nested Hypervisors

Cheat developers sometimes use nested hypervisors to create a transparent analysis environment: they run the game in a VM, with the cheat running in the VM's host. Detection of nested hypervisors relies on timing anomalies: `CPUID` executed inside a nested VM is handled by two hypervisors in sequence, introducing double the overhead. `RDMSR` and `WRMSR` instructions similarly have amplified latency. Statistical analysis of hundreds of timing measurements can reliably distinguish native execution, single-level virtualization, and nested virtualization.

12. Hardware Fingerprinting and Ban Enforcement

Collected Identifiers

Anti-cheats collect multiple hardware identifiers to create a unique fingerprint that survives account banning:

- **SMBIOS data**: Manufacturer, product name, serial numbers, UUID. Accessed via `NtQuerySystemInformation(SystemFirmwareTableInformation, ...)` or directly via the firmware table.
- **Disk serial numbers**: Physical disk serial numbers via IOCTL `IOCTL_STORAGE_QUERY_PROPERTY`. These are stable identifiers that survive OS reinstallation.
- **GPU identifiers**: Device instance ID, adapter LUID.
- **MAC addresses**: NIC MAC addresses via NDIS or the registry. These are spoofable at the software level but often not changed by non-technical users.
- **Boot GUID**: The `MachineGuid` in `HKLM\SOFTWARE\Microsoft\Cryptography`, or more persistently, the UEFI firmware's platform UUID accessible via SMBIOS.

The Vanguard analysis from the `rhaym-tech` GitHub gist documents `vgk.sys` collecting BIOS information, including the system UUID and various SMBIOS fields, which are combined into a hardware fingerprint for ban enforcement.

HWID Spoofing and Detection

HWID spoofing involves modifying the identifiers that the anti-cheat reads to evade a hardware ban. Spoofing approaches include:

- Registry-based spoofing: Patching the registry entries that report disk serial numbers, MAC addresses, and SMBIOS data. This works against anti-cheats that query these through registry paths rather than directly.
- Driver-level spoofing: A kernel driver that intercepts IOCTL requests for hardware identifiers and returns spoofed values. This works against anti-cheats that use standard IOCTL paths but fails against anti-cheats that query hardware directly.
- Physical spoofing: Programming a different MAC address into NIC firmware, flashing new disk serial numbers (supported by some drives). This is rare and sometimes permanent.

Anti-cheats detect spoofing by cross-referencing multiple identifier sources. If the SMBIOS UUID is `FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF` (a common spoofed value), that is an immediate flag. If the reported disk model is "Samsung 970 EVO" but the disk serial number format does not match Samsung's format, that is a spoof indicator. If the UEFI firmware tables report one UUID and the registry reports a different one, the registry value has been tampered with.

13. The Arms Race: Current Trends and Future Directions

The Escalation Hierarchy

The escalation we have seen over the past decade follows a clear pattern:

1. Usermode cheats were countered by usermode anti-cheat.
2. Kernel cheats were countered by kernel anti-cheat.
3. Kernel cheats with BYOVD were countered by driver blocklists and stricter DSE enforcement.
4. Hypervisor-based cheats were countered by hypervisor detection.
5. DMA cheats are the current frontier, partially countered by IOMMU, Secure Boot, and TPM attestation.
6. The next level is firmware-based attacks, where the cheat is embedded in the SSD firmware, GPU firmware, or NIC firmware.

Firmware attacks are particularly concerning because they survive OS reinstallation, are invisible to all kernel-level inspection, and are extremely difficult to detect without physical access to the device for firmware verification. There is no widespread anti-cheat defense against firmware cheats today.

AI-Powered Cheats

The next-generation threat is aimbots powered by computer vision models that run on the GPU or a secondary computer. These systems use a camera or screen capture to analyze the game frame, identify targets, and move the mouse via hardware (a USB HID device, bypassing software input inspection entirely). The mouse movements they produce can be configured to mimic human motion patterns, making statistical detection much harder.

An AI aimbot operating via hardware HID is, from the game machine's perspective, completely indistinguishable from a human using a mouse. All input comes through legitimate hardware channels. No code runs in the game process. The kernel is entirely clean. The only detection surface is the behavioral profile: the accuracy, reaction times, and movement patterns that the AI produces.

This is why the behavioral ML approaches discussed in section 10 are not optional but increasingly central to effective anti-cheat.

The Privacy Debate

Kernel-level anti-cheat is deeply unpopular among privacy advocates. The criticisms are substantive:

A driver running at ring 0 with boot-time loading has access to everything on the system. While BattlEye, EAC, and Vanguard are not documented to abuse this access for surveillance, the technical capability exists. The ARES 2024 paper's analysis underscores that the trust model is identical to what we use for security-critical software, which means any vulnerability in these components is a local privilege escalation to ring 0.

The fact that games require installation of boot-time kernel drivers as a condition of play is also a significant attack surface concern. A vulnerability in `vgk.sys` is a local privilege escalation to ring 0. The anti-cheat software itself becomes an attack target.

Attestation-Based Approaches

The most technically promising direction for anti-cheat is remote attestation. Rather than running a ring-0 driver that actively fights cheats, the system proves to the game server that it is running in a known-good state. TPM-

based measured boot, combined with UEFI Secure Boot, can generate a cryptographically signed attestation that specific bootloaders, kernels, and drivers were loaded. The server refuses connections from systems that cannot provide valid attestation.

This is not a complete solution (a sufficiently sophisticated attacker can potentially manipulate attestation), but it significantly raises the bar. Attestation can coexist with traditional scanning to provide defense in depth.

Cloud Gaming as Anti-Cheat

Cloud gaming (GeForce Now, Xbox Cloud Gaming) is architecturally the ultimate anti-cheat for certain game categories. If the game runs in a data center and only video is streamed to the client, there is no game client code to exploit, no game memory to read, and no local environment to manipulate. The cheat attack surface reduces to input manipulation and video analysis, both of which have relatively straightforward detection approaches.

The constraint is latency: cloud gaming is unsuitable for competitive titles where single-digit millisecond reaction times matter. For casual and semi-competitive play, cloud delivery may increasingly be the answer.

14. Conclusion

Modern kernel anti-cheat systems represent a layered defensive architecture that operates across every available level of the Windows privilege model:

- **Kernel callbacks** (`ObRegisterCallbacks`, `PsSetCreateProcessNotifyRoutineEx`, `PsSetLoadImageNotifyRoutine`) provide real-time visibility into system events with the ability to actively block malicious operations.
- **Memory scanning** (VAD walking, big pool enumeration, code section hashing) provides periodic verification that game memory has not been tampered with and that no injected code is present.
- **Behavioral telemetry** (input analysis, statistical profiling, ML inference) catches cheats that are architecturally invisible to kernel scanning.
- **Hardware fingerprinting** enforces ban decisions across account resets.
- **Anti-debug and anti-VM protections** make reverse engineering and development significantly more difficult.

No single technique is sufficient. Kernel callbacks can be bypassed by DMA attacks. Memory scanning can be evaded by hypervisor-based cheats that intercept memory reads. Behavioral detection can be fooled by sufficiently human-mimicking AI. Hardware fingerprinting can be defeated by hardware spoofers. It is the combination of all these layers, continually updated in response to new evasion techniques, that provides meaningful protection.

The trajectory of this arms race points toward hardware attestation and server-side verification as the ultimate foundations of trustworthy game security. Software-only client-side protection will always be asymmetric: defenders must check everything, attackers need only find one gap. Hardware attestation shifts this asymmetry by making it extremely difficult to demonstrate a trustworthy state while operating a modified system.

Until that foundation is universally available and enforced, kernel anti-cheat remains the best practical defense available, with all the associated complexity, privacy implications, and attack surface that entails.

References

1. Collins, R. et al. “Anti-Cheat: Attacks and the Effectiveness of Client-Side Defences.” *CheckMATE 2024 (Workshop co-located with CCS 2024)*. https://tomchothia.gitlab.io/Papers/AntiCheat2024.pdf
2. Vella, R. et al. “If It Looks Like a Rootkit and Deceives Like a Rootkit: A Critical Analysis of Kernel-Level Anti-Cheat Systems.” *ARES 2024*. https://arxiv.org/pdf/2408.00500
3. Sousa, J. et al. “AntiCheatPT: A Transformer-Based Approach to Cheat Detection in First-Person Shooter Games.” *2025*. https://arxiv.org/html/2508.06348v1
4. secret.club. “Reversing BattlEye’s anti-cheat kernel driver.” *2019*. https://secret.club/2019/02/10/battleye-anticheat.html
5. secret.club. “Easy Anti-Cheat integrity check bypass.” *2020*. https://secret.club/2020/04/08/eac_integrity_check_bypass.html
6. back.engineering. “Reversing BEDaisy.sys.” *2020*. https://back.engineering/blog/2020/08/22/
7. Aki2k. “BEDaisy Reverse Engineering.” *GitHub*. https://github.com/Aki2k/BEDaisy
8. archie-osu. “Vanguard Dispatch Table Hooks Analysis.” *2025*. https://archie-osu.github.io/2025/04/11/vanguard-research.html
9. rhaym-tech. “Vanguard vgk.sys Analysis Gist.” *GitHub Gist*. https://gist.github.com/rhaym-tech/f636b76deeca15528e70304b5ee95980
10. donnaskiez. “ac: Open Source Kernel Anti-Cheat.” *GitHub*. https://github.com/donnaskiez/ac