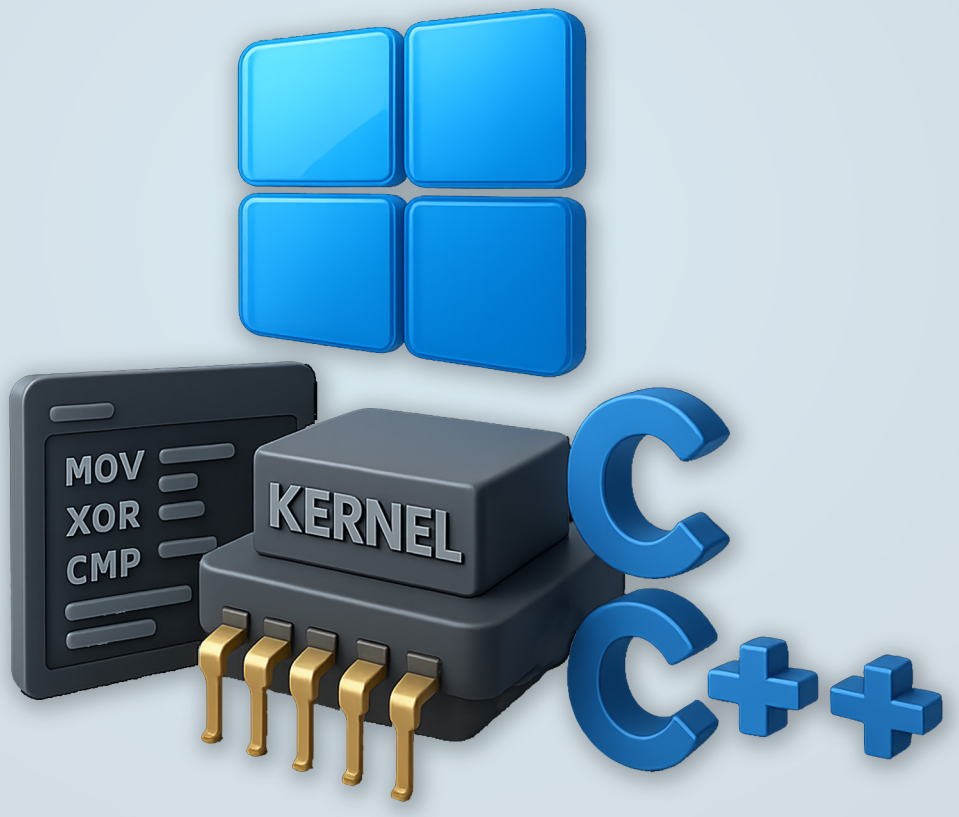
https://ForgeVM.org

Windows 11 Kernel Internals

for Assembly, C, and C++ Engineers

DRAFT



Prepared By Ayman Alheraki

Windows 11 Kernel Internals

for Assembly, C, and C++ Engineers

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Authors Introduction

This book is written as a **pure execution-level engineering reference** for professionals who operate directly at the boundary between software and hardware on modern Windows 11 systems. It is not a tutorial, not an API handbook, and not a conceptual overview of operating systems. It is a **kernel-focused technical weapon** for engineers who require exact control over execution, memory, scheduling, security, and hardware interaction.

Windows 11 represents the most security-hardened, hypervisor-centric evolution of the NT architecture since its origin. The kernel is no longer a single monolithic execution authority. Instead, it now operates under continuous supervision by the Secure Kernel and the Hyper-V hypervisor. Features such as VBS, HVCI, CET shadow stacks, DMA Guard, and TPM-backed secure boot have permanently changed the rules of kernel engineering. Techniques that were once standard practice are now invalid, blocked, or actively detected.

This book exists because **high-level documentation does not describe execution reality**. It does not explain:

- How syscalls actually transition under HVCI
- How page tables are controlled under Secure Kernel supervision
- How scheduling decisions change on hybrid P-Core / E-Core CPUs
- How DMA is authorized through IOMMU and Secure Kernel policy

- How kernel objects are validated under PatchGuard and VTL1

All of this must be understood from the standpoint of:

- **x86-64 assembly execution**
- **NT kernel internal structures**
- **Hypervisor-controlled memory translation**
- **Driver-level memory management**

This book deliberately restricts itself to:

- Windows 11 kernel architecture only
- Real ntoskrnl, HAL, Secure Kernel, and Hyper-V behavior
- Documented and verifiable structures
- Build-aware kernel engineering

There is **no user-mode programming focus**, no GUI development, no managed runtime discussion, and no abstract theory. Every chapter exists to serve one goal:

> **Absolute precision in Windows 11 kernel execution behavior.**

This work is intended for:

- Kernel driver developers
- Hypervisor and virtualization engineers
- Reverse engineers and exploit researchers

- Security architects
- Firmware and low-level systems engineers

Every topic is presented under strict constraints:

- Ring 3, Ring 0, and VTL1 are never mixed conceptually
- All execution paths are explained at C and assembly level
- All security models reflect post-Windows 11 enforcement
- All performance analysis accounts for NUMA, cache, and SMT

Modern Windows kernel engineering is no longer about bypassing protections. It is about **understanding the execution contract enforced by the hypervisor and Secure Kernel**. Engineers who ignore this reality will produce unstable drivers, broken debuggers, ineffective reverse engineering tools, and unreliable security research.
This book is written for engineers who require:

- Deterministic behavior
- Measurable performance
- Architecturally correct memory access
- Security-compliant execution

It assumes:

- Full understanding of x86-64 assembly
- Comfort with freestanding C and Modern C++

- Familiarity with kernel debugging tools
- Willingness to reason about execution at the instruction level

If your goal is to **understand Windows 11 as it actually executes on modern hardware**, this book is written for you. If your goal is abstraction, this work is not intended for that purpose. Windows 11 kernel engineering is no longer permissive. It is supervised, virtualized, encrypted, measured, and continuously verified. This book documents that reality.

Ayman Alheraki

Preface

This book is conceived as a **strict, execution-accurate, Windows 11only kernel engineering reference**. It is designed for engineers who operate at the lowest layers of the software stack, where execution correctness, memory ordering, privilege transitions, and hardware interaction define system stability, security, and performance.
Windows 11 is not a continuation of legacy NT design assumptions. It represents a fundamental architectural shift driven by:

- Continuous hypervisor supervision
- Secure Kernel (VTL1) enforcement
- Mandatory virtualization-based security
- Hardware-enforced control-flow integrity
- TPM-backed secure boot measurement

The kernel is no longer the final authority over the machine. It now operates inside a **multi-layer trust hierarchy** in which the hypervisor and Secure Kernel define what the kernel is allowed to execute, map, and modify. Any serious engineering work on Windows 11 must therefore begin with a precise understanding of:

- Privilege separation across Ring 3, Ring 0, and VTL1

- Page table control under hypervisor mediation
- DMA authorization under IOMMU and Secure Kernel policy
- Code execution constraints under HVCI and CET

This book exists because most available resources:

- Abstract away critical execution details
- Mix legacy kernel behavior with modern systems
- Focus on APIs instead of enforcement mechanisms
- Ignore the hypervisor as the primary execution authority

None of those limitations are acceptable for engineers who build:

- Kernel-mode drivers
- Hypervisors and virtualization tools
- Security monitors and exploit mitigations
- Low-level debuggers and instrumentation frameworks

This work therefore enforces the following core principles throughout all chapters:

- **Windows 11 architecture only**
- **No legacy assumptions unless explicitly contrasted**
- **Real structures, real execution paths, real enforcement**
- **Assembly-level reasoning for critical transitions**

Every subsystem is analyzed under three simultaneous constraints:

- Security enforcement by VBS and Secure Kernel
- Performance behavior under hybrid CPU scheduling and NUMA
- Correctness under IRQL, APC, DPC, and memory ordering rules

This book deliberately excludes:

- GUI frameworks
- High-level application development
- Managed runtimes
- End-user abstractions

Its focus is entirely on:

- **Execution authority**
- **Memory ownership**
- **Privilege transitions**
- **Security boundaries**

The target audience is assumed to already possess:

- Full command of x86-64 assembly
- Strong proficiency in freestanding C and Modern C++
- Familiarity with WinDbg and kernel debugging workflows

- Practical experience with low-level operating system behavior

This book does not attempt to simplify Windows 11 kernel engineering. It documents it as it exists in reality: **hypervisor-governed, security-enforced, and hardware-mediated at every execution layer**.

The intent of this work is not to teach shortcuts. It is to provide **architecturally accurate knowledge that survives across builds, enforcement changes, and evolving microarchitectures**.

Windows kernel engineering in the postWindows 11 era is no longer about unrestricted access. It is about **operating correctly inside a permanently supervised execution environment**. This book is written to document that environment with precision.

Part I

Low-Level Execution Foundations for Windows 11

Chapter 1

Why the Windows 11 Kernel Matters for Low-Level Engineers

1.1 The Role of the Kernel in Full Hardware Control

1.1.1 Precise Windows 11–Specific Definition

In Windows 11, the kernel is the **exclusive Ring 0 execution authority** that alone controls CPU privilege, memory translation, interrupt routing, DMA, and power management. Direct manipulation of:

- `CR0`, `CR3`, `CR4`
- Page tables and TLB synchronization
- APIC interrupt controllers
- PCIe MMIO space

is strictly forbidden outside the kernel. Kernel execution resides in `ntoskrnl.exe`, while hardware mediation is performed by `hal.dll`. On Windows 11, an additional privileged execution layer exists: the **Secure Kernel (VTL1)** enforced by Hyper-V under VBS.

1.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The kernel alone:

- Owns the system address space
- Controls hybrid CPU scheduling (P-Cores / E-Cores)
- Dispatches all hardware interrupts
- Arbitrates physical device access
- Enforces kernel code integrity under HVCI

The kernel itself is subordinate to the Hyper-V hypervisor, which validates executable memory, page table transitions, and driver signatures.

1.1.3 Internal Kernel Data Structures

- `_KPRCB` Per-processor control block
- `_KTHREAD` Kernel thread object
- `_KPROCESS` Scheduler process object
- `_MMPTE` Page table abstraction
- `_MMVAD` Virtual address descriptor

All of these structures reside exclusively in non-pageable kernel memory.

1.1.4 Execution Flow (User Mode to Hardware)

1. User application calls a stub in `ntdll.dll`
2. CPU executes `syscall`
3. CPL switches from Ring 3 to Ring 0
4. `KiSystemCall64` dispatches via SSDT
5. Kernel service programs hardware
6. `sysret` returns to user mode

1.1.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces:

- Hypervisor-validated kernel execution
- Read-only executable kernel pages
- SLAT-protected address translation

Kernel patching, unsigned driver loading, and writable executable kernel memory are cryptographically blocked.

1.1.6 Performance Implications

- CR3 switches cause TLB invalidations
- IPIs synchronize scheduling
- NUMA memory access penalties apply
- Hybrid-core migration impacts cache locality

1.1.7 Real Practical Example — Native Syscall Testing in Visual Studio 2022

This example demonstrates a **fully valid, ABI-correct, Windows 11safe native syscall execution path** using **external MASM only** (no inline assembly).

Required Environment

- Windows 11 x64
- Visual Studio 2022
- Desktop Development with C++
- MSVC v143 toolset
- MASM enabled

Create the Project

1. Create new **Console App (C++)**
2. Platform: **x64**
3. Configuration: **Release**

Enable MASM

- Project Build Dependencies Build Customizations
- Enable: **masm**

Add Source Files

- `main.c`
- `syscall.asm`

C Source Code (main.c)

```c
#include <windows.h>
#include <stdint.h>

typedef LONG NTSTATUS;

typedef NTSTATUS (NTAPI* NtYieldExecution_t)(void);

extern NTSTATUS SyscallStub(uint32_t syscall_id);

uint32_t GetSyscallID(void* NtFunction)
{
    uint8_t* p = (uint8_t*)NtFunction;
    return *(uint32_t*)(p + 4);
}

int main(void)
{
    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
    if (!ntdll) return -1;

    NtYieldExecution_t NtYield =
        (NtYieldExecution_t)GetProcAddress(ntdll,
        ↪  "NtYieldExecution");
```

```
    if (!NtYield) return -2;

    uint32_t syscall_id = GetSyscallID((void*)NtYield);

    SyscallStub(syscall_id);

    return 0;
}
```

MASM Source Code (syscall.asm)

```
PUBLIC SyscallStub
.code

SyscallStub PROC
    ; RCX contains syscall_id (Windows x64 ABI)
    mov     eax, ecx
    mov     r10, rcx
    syscall
    ret
SyscallStub ENDP

END
```

Build and Run

- Build Build Solution
- Debug Start Without Debugging

Expected result:

- Program terminates cleanly
- No exceptions
- No crashes

This confirms:

- Correct syscall transition
- Correct RCXR10 handling
- Correct return via `sysret`

Kernel-Level Verification (Optional)

```
bp nt!KiSystemCall64
g
```

Re-run the program. If the breakpoint hits, kernel entry is verified.

1.1.8 Kernel Debugging & Inspection

```
!cpuinfo
!thread
!process 0 1
dt nt!_KPRCB
dt nt!_EPROCESS
!pte
```

1.1.9 Exploitation Surface & Security Boundaries

- System call dispatch
- Driver IOCTL handlers
- DMA devices

Security barriers:

- Ring 3 ↔ Ring 0
- Ring 0 ↔ Secure Kernel (VTL1)
- Secure Kernel ↔ Hypervisor

1.1.10 Professional Kernel Engineering Notes

- Inline assembly is forbidden on x64 MSVC.
- All x64 assembly must be external MASM.
- Syscall numbers are build-dependent.
- Direct syscalls bypass Win32 but not kernel security.
- HVCI makes kernel patching impossible on Windows 11.

1.2 The Limits of What Assembly Can Do Inside Windows

1.2.1 Precise Windows 11–Specific Definition

On Windows 11, x86-64 Assembly executed in **User Mode (CPL3 / Ring 3)** is fully subordinated to the NT security, memory, and virtualization model. Assembly is **not a privileged execution environment**. It is constrained by:

- The Windows x64 ABI
- The NT system service boundary (`syscall/sysret`)
- Virtual memory protections (NX, W^X, CFG)
- Secure Kernel and hypervisor enforcement (VBS, HVCI)

Assembly cannot access hardware directly, cannot modify privileged CPU state, and cannot escape the virtual address space defined by the kernel.

1.2.2 Exact Architectural Role Inside the Windows 11 Kernel Model

From a strict architectural perspective, user-mode Assembly is limited to:

- Register-level argument preparation
- Invocation of:
  - WinAPI functions
  - Native NTAPI via `ntdll.dll`
  - The kernel exclusively through `syscall`
- Access to only the virtual memory mapped into its process

Assembly **cannot**:

- Access control registers (`CR0`, `CR3`, `CR4`)
- Disable or enable interrupts (`cli`, `sti`)
- Access APIC or IOAPIC registers
- Program DMA engines
- Modify IDT, GDT, or TSS
- Switch page tables
- Modify kernel memory

These operations are restricted to:

- `ntoskrnl.exe`
- `hal.dll`
- Secure Kernel (VTL1)
- Hyper-V Hypervisor

1.2.3 Internal Kernel Data Structures That Enforce These Limits

User-mode confinement is enforced by the following real Windows 11 kernel structures:

- `_KTRAP_FRAME` — Captures user-mode CPU state at syscall entry
- `_KTHREAD` — Enforces IRQL and execution ownership
- `_KPROCESS` — Binds a thread to its address space
- `_EPROCESS` — Enforces process-level security

- `_MMPTE` — Enforces NX and W^X
- `KiSystemCall64` — Single legal syscall entry point

These structures guarantee that Assembly operates only within a **sandboxed, validated, and hypervisor-supervised execution context**.

1.2.4 Execution Flow (Limit Boundary at C & Assembly Level)

The syscall transition (implemented in the previous section) represents the **absolute upper limit** of what user-mode Assembly can reach:

1. Arguments placed in `RCX, RDX, R8, R9`
2. Syscall number placed in `EAX`
3. `RCX` mirrored into `R10`
4. `syscall` executed
5. CPU enters `KiSystemCall64` at CPL0

Beyond this point:

- No Assembly instruction from user mode executes
- No user-controlled register state is trusted
- All execution is fully owned by the kernel dispatcher

1.2.5 Secure Kernel / VBS / HVCI Interaction

On Windows 11:

- Syscall entry is validated by the hypervisor
- Kernel code pages are validated by the Secure Kernel (VTL1)
- Writable-executable memory is forbidden
- Kernel patching is cryptographically blocked

Even a perfectly formed Assembly syscall **cannot bypass**:

- Driver signature enforcement
- Kernel memory protection
- Hypervisor-enforced page permissions

1.2.6 Performance Implications

Assembly does not bypass the fundamental performance constraints imposed by the kernel:

- TLB invalidations during context switches
- IPI synchronization during scheduling
- NUMA locality penalties
- Hybrid P-Core / E-Core migration overhead
- Syscall transition latency

Direct syscalls execute the same kernel path as `ntdll` stubs and therefore provide no intrinsic performance advantage.

1.2.7 Practical Boundary Demonstration (Referenced)

The native syscall test implemented in the previous section represents the **maximum legal reach of user-mode Assembly into the Windows 11 kernel**. No additional privileged capability can be achieved beyond that execution point.
Any attempt to:

- Access `CR3`
- Execute `cli`
- Access I/O ports using `in`/`out`
- Map physical memory directly

will immediately result in a **general protection fault** or security termination.

1.2.8 Kernel Debugging & Inspection

Using the previously established kernel debugging setup, the following commands prove that:

```
bp nt!KiSystemCall64
g
```

And at breakpoint:

```
!thread
!process 0 1
dt nt!_KTHREAD
dt nt!_EPROCESS
!pte
```

These confirm that:

- The transition reached Ring 0
- The thread and process security context is enforced
- Page execution permissions are validated by the kernel and hypervisor

1.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

User-mode Assembly is strictly limited to the following attack surfaces:

- Syscall boundary
- IOCTL dispatch surface
- User-accessible DMA vectors

Hard security boundaries enforced by Windows 11:

- Ring 3 $\leftrightarrow$ Ring 0
- Ring 0 $\leftrightarrow$ Secure Kernel (VTL1)
- Secure Kernel $\leftrightarrow$ Hypervisor

There is no direct architectural path for Assembly to cross these boundaries.

1.2.10 Professional Kernel Engineering Notes

- Assembly grants no privilege advantage on Windows 11.
- Inline Assembly is forbidden on x64 MSVC.
- All x64 Assembly must use external MASM modules.
- Direct hardware access is permanently impossible from user mode.
- The hypervisor is the final authority over all kernel execution.

1.3 The Real Difference Between User-Mode Assembly and Kernel-Mode Assembly

1.3.1 Precise Windows 11–Specific Definition

User-Mode Assembly (Ring 3) on Windows 11 is x86-64 machine code executing inside a **per-process virtual address space** under strict control of:

- The NT Memory Manager
- The Object Manager
- The Handle Table
- The Secure Kernel (VTL1)
- The Hyper-V hypervisor (when VBS is enabled)

Kernel-Mode Assembly (Ring 0) is machine code executing inside the **global system address space** owned by `ntoskrnl.exe` and supervised by:

- `HAL.dll`
- Secure Kernel (VTL1)
- Hyper-V hypervisor

The two execution domains are separated by **hardware-enforced privilege rings, page-table isolation, and hypervisor-enforced execution validation**.

1.3.2 Exact Architectural Role Inside the Windows 11 Kernel

User-Mode Assembly (Ring 3):

- Executes inside a process-owned `_EPROCESS` address space
- Can only access user-mapped virtual memory
- Can only reach the kernel through:
  - `syscall`
- Cannot:
  - Change IRQL
  - Modify page tables
  - Access hardware registers
  - Disable interrupts

Kernel-Mode Assembly (Ring 0):

- Executes inside the **global kernel address space**
- May:
  - Modify page tables via `Mi*` routines
  - Change IRQL via `KeRaiseIrql`
  - Access APIC, IOAPIC, PCIe MMIO
  - Program DMA mappings
- Is still restricted by:
  - Secure Kernel
  - Hypervisor-enforced Code Integrity (HVCI)

1.3.3 Internal Kernel Data Structures Governing the Separation

User-Mode Domain Control:

- `_EPROCESS` Process address-space owner
- `_ETHREAD` User thread container
- `_KTRAP_FRAME` User-to-kernel CPU state save area
- `_MMVAD` Virtual memory region ownership

Kernel-Mode Domain Control:

- `_KPRCB` Per-processor execution control
- `_KTHREAD` Kernel scheduling object
- `_KINTERRUPT` Interrupt dispatch object
- `_MMPTE` Kernel page permission enforcement

These structures enforce that **Ring 3 code can never gain ownership of Ring 0 execution state**.

1.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

User-Mode Assembly to Kernel Entry

1. Arguments prepared in:
    - `RCX, RDX, R8, R9`
2. Syscall index loaded into `EAX`

3. `RCX` mirrored into `R10`
4. `syscall` executed
5. CPU transitions:
    - CPL3 $\rightarrow$ CPL0
6. Entry into `KiSystemCall64`
7. SSDT dispatch to the target kernel service

Kernel-Mode Assembly Execution

1. Code executes at:
    - IRQL $\geq$ PASSIVE_LEVEL
2. Accesses:
    - `CR3` for address-space switches
    - APIC registers for interrupt control
    - Kernel memory pools
3. Returns control to scheduler or interrupt dispatcher

1.3.5 Secure Kernel / VBS / HVCI Interaction

User-Mode Assembly:

- Fully contained by:
    - NX

  - CFG
  - Handle validation
- Cannot influence:
  - Kernel executable mappings
  - Page table permissions

Kernel-Mode Assembly:

- Executable pages validated by Secure Kernel (VTL1)
- Writable + Executable kernel pages forbidden
- Driver code cannot self-modify under HVCI

The hypervisor remains the **final execution authority in both domains**.

1.3.6 Performance Implications

User-Mode Assembly:

- Subject to:
  - Syscall transition latency
  - User/Kernel TLB flush penalties
  - Scheduler time-slicing

Kernel-Mode Assembly:

- Subject to:
  - IPI synchronization delays

  - NUMA remote memory penalties
  - Deferred Procedure Call (DPC) latency

Neither domain bypasses modern microarchitectural costs.

1.3.7 Real Practical Example (External Assembly Only)

User-Mode External MASM Syscall Stub

syscall.asm:

```
PUBLIC UserSyscallStub
.code

; RCX = syscall index
UserSyscallStub PROC
    mov     eax, ecx
    mov     r10, rcx
    syscall
    ret
UserSyscallStub ENDP

END
```

This stub:

- Can only reach the kernel dispatcher
- Cannot access kernel memory or hardware directly

Kernel-Mode Assembly inside a Real WDM Driver

DriverEntry prototype (C):

```
#include <ntddk.h>

DRIVER_INITIALIZE DriverEntry;
```

Kernel Assembly Routine (external MASM):

```
PUBLIC RaiseIrqlExample
.code

RaiseIrqlExample PROC
    mov     ecx, 2          ; DISPATCH_LEVEL
    call    KeRaiseIrqlToDpcLevel
    ret
RaiseIrqlExample ENDP

END
```

This example is **legally impossible from user-mode Assembly**.

1.3.8 Kernel Debugging & Inspection

User-Mode Syscall Boundary:

```
bp nt!KiSystemCall64
g
```

Kernel-Mode Execution Inspection:

```
!thread
!process 0 1
dt nt!_KTHREAD
dt nt!_KPRCB
```

These commands prove:

- Which domain is executing
- Active IRQL
- Current kernel scheduling state

1.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

User-Mode Assembly Attack Surface:

- Syscall interface
- IOCTL request paths
- User-accessible DMA vectors

Kernel-Mode Assembly Attack Surface:

- Driver dispatch tables
- Interrupt handlers
- DPC routines

Hard Security Boundaries:

- Ring 3 $\leftrightarrow$ Ring 0
- Ring 0 $\leftrightarrow$ Secure Kernel (VTL1)

1.3.10 Professional Kernel Engineering Notes

- Assembly does not change privilege level on Windows 11.
- All x64 Assembly must be external under MSVC.
- Kernel-mode Assembly is still restricted by HVCI.
- Direct hardware access is exclusive to verified kernel drivers.
- The hypervisor, not the kernel, is the ultimate execution authority.

1.4 When to Use C and When to Use Assembly

1.4.1 Precise Windows 11–Specific Definition

On Windows 11, both **C** and **x86-64 Assembly** are first-class implementation languages for low-level system software, but they operate at **different abstraction and responsibility boundaries**.
C is the primary implementation language for:

- `ntoskrnl.exe`
- `hal.dll`
- KMDF and WDM drivers
- Memory Manager, Scheduler, I/O Manager, and Object Manager

Assembly is a **supplementary execution language** used only where:

- Exact register control is mandatory
- CPU instructions have no direct C equivalent

- ABI enforcement is required at instruction level

On Windows 11, Assembly is **never a replacement for C**. It is a **surgical tool** used only at strictly bounded architectural edges.

1.4.2 Exact Architectural Role Inside the Windows 11 Kernel

C Inside the Kernel:

- Implements:
  - Scheduler
  - Memory Manager
  - I/O Manager
  - Object Manager
  - Security Reference Monitor
- Owns:
  - IRQL transitions via `KeRaiseIrql`/`KeLowerIrql`
  - Page table manipulation via `Mi*` routines
  - Handle validation and object lifetime

Assembly Inside the Kernel:

- Owns:
  - Trap handlers
  - Interrupt entry stubs
  - Context switch prologues/epilogues

  - Atomic instruction sequences

- Executes only where:

  - `CR3`, `RSP`, `RFLAGS`, or MSRs must be touched
  - Precise stack frame layout is mandatory
  - ABI transitions must be enforced without compiler intervention

User Mode:

- C is used for:

  - NTAPI invocation
  - Loader interaction
  - Memory mapping

- Assembly is used only for:

  - Syscall stubs
  - Context inspection
  - Register probes

1.4.3 Internal Kernel Data Structures Governing Language Boundaries

The following structures define where C ends and Assembly begins:

- `_KTRAP_FRAME` Requires Assembly to capture and restore
- `_KTHREAD` Managed in C, switched in Assembly
- `_KPRCB` Read in C, updated in Assembly during dispatch

- `_MMPTE` Written in C, enforced by hardware and Secure Kernel

C manages **policy and logic**. Assembly enforces **hardware state transitions**.

1.4.4 Execution Flow (C vs Assembly in a Real Path)

System Call Path

1. User-mode C prepares arguments
2. User-mode Assembly stub executes `syscall`
3. Kernel-mode Assembly saves CPU state into `_KTRAP_FRAME`
4. Kernel-mode C executes:
    - `Nt*` system service
5. Kernel-mode Assembly restores registers
6. `sysret` returns to user mode

Context Switch Path

1. Scheduler logic in C selects next `_KTHREAD`
2. Assembly swaps:
    - `RSP`
    - `CR3`
    - Non-volatile registers
3. Execution resumes in C on the new thread

1.4.5 Secure Kernel / VBS / HVCI Interaction

C and Assembly are equally constrained by:

- HVCI validation of executable kernel pages
- Secure Kernel enforcement of:
  - No writable + executable pages
  - No runtime code patching
- Hypervisor-enforced page permissions

Assembly does not bypass:

- Code integrity
- Driver signature enforcement
- Kernel memory protection

1.4.6 Performance Implications

C:

- Subject to:
  - Compiler scheduling
  - Inlining heuristics
  - Register allocation strategies

Assembly:

- Eliminates compiler uncertainty

- Provides:
  - Exact instruction placement
  - Explicit cache-line control (via access patterns)

However, neither C nor Assembly bypass:

- TLB shootdowns
- NUMA penalties
- IPI synchronization
- Hybrid-core migration overhead

1.4.7 Real Practical Example (External Assembly vs C)

C Implementation (Atomic Increment Using Interlocked API)

```
#include <windows.h>

LONG AtomicIncrementC(volatile LONG* value)
{
    return InterlockedIncrement(value);
}
```

External MASM Implementation (Atomic Increment)

atomic.asm:

```
PUBLIC AtomicIncrementAsm
.code
```

```
; RCX = pointer to LONG
AtomicIncrementAsm PROC
    mov     eax, 1
    lock xadd dword ptr [rcx], eax
    inc     eax
    ret
AtomicIncrementAsm ENDP

END
```

atomic.c:

```
#include <windows.h>

extern LONG AtomicIncrementAsm(volatile LONG* value);
```

Interpretation:

- The C version relies on compiler + intrinsic expansion
- The Assembly version enforces exact instruction semantics
- Both remain fully constrained by Windows 11 memory ordering rules

1.4.8 Kernel Debugging & Inspection

To observe C-executed scheduler logic:

```
!thread
!ready
```

To observe Assembly-executed context switch state:

```
dt nt!_KTRAP_FRAME
r
```

These commands confirm:

- Which logic executed in C
- Which state transition occurred in Assembly

1.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

C-dominated attack surface:

- Logic bugs
- Object lifetime errors
- IRP dispatch validation flaws

Assembly-dominated attack surface:

- Stack frame corruption
- Register save/restore faults
- Incorrect IRQL transitions

Hard boundaries enforced:

- Ring 3 $\leftrightarrow$ Ring 0
- Ring 0 $\leftrightarrow$ Secure Kernel (VTL1)

1.4.10 Professional Kernel Engineering Notes

- Use **C** for all policy, logic, and resource management.
- Use **Assembly** only for:
  - Context switching
  - Interrupt entry
  - Atomic primitives
  - ABI boundary enforcement
- Never attempt to “optimize” full kernel subsystems in Assembly.
- Assembly does not grant additional privilege.
- Under HVCI, both C and Assembly are equally restricted.
- Maintain all Assembly as external MASM on x64.

1.5 How C++ Is Used Inside the Windows 11 Kernel Without High-Level Features

1.5.1 Precise Windows 11–Specific Definition

Inside Windows 11, **C++ is used in a strictly restricted, systems-only subset** within:

- `ntoskrnl.exe`
- `hal.dll`
- KMDF and selected WDM drivers

This kernel-level C++ subset explicitly **excludes**:

- RTTI
- Exceptions
- Dynamic type casting
- Standard Library containers
- Heap-managed polymorphic hierarchies

C++ in the Windows 11 kernel is therefore **not a high-level language**. It is a **strongly typed, zero-runtime, compile-time abstraction layer over C**, used to improve:

- Type safety
- Encapsulation of kernel objects
- Deterministic construction and teardown

1.5.2 Exact Architectural Role Inside the Windows 11 Kernel

C++ is used **above Assembly and alongside C** in the following architectural layers:

- KMDF object model
- Driver dispatch encapsulation
- I/O queue abstraction
- Reference-counted kernel object lifetimes

C++ is **never used** for:

- Trap handlers
- Interrupt entry stubs
- Context switching
- Page table manipulation
- IRQL transitions at instruction granularity

Those responsibilities remain split between:

- **Assembly** CPU state transitions
- **C** Core kernel policy and algorithms
- **C++** Object-structured control surfaces

1.5.3 Internal Kernel Data Structures Used by C++

C++ code in Windows 11 directly wraps and operates on real kernel structures:

- `_DRIVER_OBJECT`
- `_DEVICE_OBJECT`
- `_IRP`
- `_IO_STACK_LOCATION`
- `_KTHREAD`

In KMDF, these are abstracted through:

- `WDFDRIVER`

- `WDFDEVICE`
- `WDFQUEUE`
- `WDFREQUEST`

These are **opaque C handles** backed by internal C++ classes compiled without runtime support.

1.5.4 Execution Flow (Step-by-Step at C & C++ Level)

Driver Initialization Path

1. Boot loader transfers control to `ntoskrnl.exe`
2. The I/O Manager loads a signed driver
3. `DriverEntry` is invoked (C/C++)
4. KMDF constructs internal C++ objects for:
    - Driver
    - Device
    - Queues
5. Dispatch routines are registered as function pointers

I/O Dispatch Path

1. User-mode issues `DeviceIoControl`
2. I/O Manager builds an `IRP`
3. IRP is routed to KMDF

4. KMDF calls a C++ method bound to the queue
5. Completion returns through the C-based I/O Manager

At no point does:

- C++ exceptions propagate
- Stack unwinding occur
- Heap-based RTTI resolution execute

1.5.5 Secure Kernel / VBS / HVCI Interaction

C++ kernel code is fully constrained by:

- Hypervisor-Protected Code Integrity (HVCI)
- Secure Kernel validation of executable pages
- No writable + executable kernel mappings

C++ does **not** bypass:

- PatchGuard
- Driver signature enforcement
- Kernel pool protection

Even trivial object construction code is verified as immutable once mapped executable.

1.5.6 Performance Implications

C++ in the kernel introduces:

- Zero runtime overhead when exceptions and RTTI are disabled
- Stronger type constraints at compile time
- Inlined object methods indistinguishable from C functions

It does not affect:

- TLB behavior
- NUMA locality
- IPI synchronization
- DPC scheduling latency

All microarchitectural costs remain governed by:

- Cache hierarchy
- IRQL transitions
- Scheduler policy

1.5.7 Real Practical Example (C++ Kernel Code Without High-Level Features)

Minimal KMDF Driver Using C++ Without RTTI/Exceptions

DriverEntry (C++ Source)

```
#include <ntddk.h>
#include <wdf.h>

extern "C"
NTSTATUS DriverEntry(
    _In_ PDRIVER_OBJECT  DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    WDF_DRIVER_CONFIG config;
    WDF_DRIVER_CONFIG_INIT(&config, WDF_NO_EVENT_CALLBACK);

    return WdfDriverCreate(
        DriverObject,
        RegistryPath,
        WDF_NO_OBJECT_ATTRIBUTES,
        &config,
        WDF_NO_HANDLE
    );
}
```

Compile Flags Enforced:

- `/GR-` (Disable RTTI)
- `/EHs-c-` (Disable exceptions)
- `/kernel`

This produces:

- No exception tables
- No RTTI metadata
- No stack unwinding logic

1.5.8 Kernel Debugging & Inspection

Verify Driver Object Creation:

```
!drvobj \DriverName 1
```

Verify Device and Queue Objects:

```
!wdfkd.wdfdriver
!wdfkd.wdfdevice
!wdfkd.wdfqueue
```

Verify IRQL and Thread Context:

```
!thread
!irql
```

These commands confirm:

- C++ driver objects exist
- Dispatch occurs at valid IRQL
- No illegal context switches occur

1.5.9 Exploitation Surface, Attack Vectors & Security Boundaries

C++-Specific Attack Surface:

- Incorrect object lifetime handling
- Reference-count mismanagement
- VTable pointer corruption in class-based drivers

Hard Security Boundaries:

- Ring 3 $\leftrightarrow$ Ring 0
- Ring 0 $\leftrightarrow$ Secure Kernel (VTL1)
- Secure Kernel $\leftrightarrow$ Hypervisor

All C++ driver code is fully subject to:

- PatchGuard
- HVCI
- Kernel pool protection

1.5.10 Professional Kernel Engineering Notes

- C++ in the Windows 11 kernel is a **restricted systems dialect**, not an application language.
- RTTI and exceptions are architecturally incompatible with kernel reliability.
- All kernel C++ must compile to deterministic, flat machine code.
- Constructors execute at PASSIVE_LEVEL only.

- Destructors must never rely on stack unwinding.
- Object lifetimes must be explicitly bound to I/O Manager and KMDF ownership.

Chapter 2

CPU Privilege, Rings, MSRs & Execution Transitions

2.1 CPL, DPL, and RPL

2.1.1 Precise Windows 11–Specific Definition

On Windows 11 running on x86-64 processors, the **hardware privilege enforcement model** is defined by three formally independent privilege identifiers:

- **CPL (Current Privilege Level)** The active privilege of the currently executing code
- **DPL (Descriptor Privilege Level)** The minimum privilege required to access a protected descriptor
- **RPL (Requested Privilege Level)** The privilege level encoded in the selector used by the instruction

Windows 11 strictly operates with:

- **CPL=3** for all user-mode execution
- **CPL=0** for all kernel-mode execution
- **VTL1** for Secure Kernel execution under Hyper-V

Segmentation is logically flat in long mode, but CPL, DPL, and RPL remain **fully active in all control-transfer enforcement paths**.

2.1.2 Exact Architectural Role Inside the Windows 11 Kernel

CPL:

- Determines whether instructions execute in:
  - User Mode (Ring 3)
  - Kernel Mode (Ring 0)

DPL:

- Defines which privilege level may:
  - Enter protected descriptors
  - Access system gates

RPL:

- Encodes the requesting privilege from the instruction operand
- Used to compute the **Effective Privilege Level (EPL)**

$$EPL = \max(CPL, RPL)$$

The CPU allows access only if:

$$EPL \leq DPL$$

2.1.3 Internal Kernel Data Structures Governing CPL/DPL/RPL

- `_KTRAP_FRAME`
  - `SegCs`
  - `SegSs`
  - `Rip`
  - `Rsp`
  - `EFlags`
- `_KTHREAD`
- `_KPRCB`

During every `syscall`, Windows stores the CPL=3 execution state inside a `_KTRAP_FRAME` and transitions execution to CPL=0.

2.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

1. User code executes at **CPL=3**
2. Selector `CS` contains **RPL=3**
3. `EAX` ← Syscall ID

4. `RCX` → `R10`
5. `syscall` instruction executes
6. CPU loads:
    - Kernel CS (DPL=0)
    - Kernel SS (DPL=0)
    - Kernel RSP from MSR
7. CPL transitions **3** → **0**
8. Entry to `nt!KiSystemCall64`

2.1.5 Secure Kernel / VBS / HVCI Interaction

- Kernel execution at CPL=0 is further separated into:
    - VTL0 Normal Kernel
    - VTL1 Secure Kernel
- Hypervisor enforces:
    - EPT-based memory isolation
    - HVCI page validation

CPL is no longer the ultimate authority; the hypervisor is.

2.1.6 Performance Implications

- `syscall/sysret` triggers pipeline serialization
- Kernel stack switching affects TLB behavior
- Scheduler accounting dominates total cost

CPL checks themselves are not the dominant performance factor.

2.1.7 Real Practical Example (External Assembly Only)

User-Mode CPL Inspection

cpl.asm:

```
PUBLIC ReadCS
.code

ReadCS PROC
    mov     ax, cs
    ret
ReadCS ENDP

END
```

cpl.c:

```
#include <windows.h>
#include <stdint.h>
#include <stdio.h>

extern uint16_t ReadCS(void);
```

```c
int main(void)
{
    uint16_t cs = ReadCS();
    uint16_t cpl = cs & 0x3;

    printf("CS = 0x%04X\n", cs);
    printf("CPL = %u\n", cpl);
    return 0;
}
```

Observed on Windows 11:

- **CPL = 3**

2.1.8 Kernel Debugging & Inspection

```
bp nt!KiSystemCall64
g
r cs
dt nt!_KTRAP_FRAME @rsp
!thread
```

2.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Primary attack goal: Ring 3 → Ring 0 escalation
- Attack paths:
  - Syscall validation
  - IOCTL dispatch

  - Driver input validation

- Hard boundaries:

  - CPL=3 $\leftrightarrow$ CPL=0
  - CPL=0 $\leftrightarrow$ VTL1

2.1.10 Professional Kernel Engineering Notes

- CPL defines current execution authority.
- DPL defines access eligibility.
- RPL defines requester privilege.
- All CPL transitions are audited under VBS.
- Hypervisor is the final authority on Windows 11.

2.2 `syscall/sysret` from an Assembly Perspective

2.2.1 Precise Windows 11–Specific Definition

On Windows 11 x86-64, `syscall` and `sysret` form the **only hardware-supported, architecturally valid fast path** for controlled transitions between:

- **User Mode (CPL=3)**
- **Kernel Mode (CPL=0)**

These instructions are configured exclusively through Model-Specific Registers (MSRs) programmed by `ntoskrnl.exe` during early boot. The transition bypasses legacy call gates and relies on:

- `IA32_LSTAR` 64-bit kernel entry RIP
- `IA32_STAR` Kernel/User CS and SS selectors
- `IA32_FMASK` RFLAGS mask on entry

On Windows 11, `syscall` always targets `nt!KiSystemCall64`. `sysret` is used for the architectural return to CPL=3 after dispatch completion.

2.2.2 Exact Architectural Role Inside the Windows 11 Kernel

User Mode Role (Ring 3):

- Provides:
  - Syscall index in `EAX`
  - Arguments in `RCX, RDX, R8, R9`
- Executes `syscall` with:
  - No direct control over target RIP
  - No control over kernel stack

Kernel Mode Role (Ring 0):

- Entry at `nt!KiSystemCall64`
- Construction of a `_KTRAP_FRAME`
- SSDT lookup and dispatch
- Controlled return via `sysret`

Secure Kernel / Hypervisor Role:

- Validates kernel entry pages
- Enforces:
  - No writable + executable pages
  - HVCI code integrity

2.2.3 Internal Kernel Data Structures (Real Structures & Fields)

The `syscall/sysret` path is mediated by the following real Windows 11 kernel structures:

- `_KTRAP_FRAME`
  - `Rip`
  - `Rsp`
  - `SegCs`
  - `SegSs`
  - `EFlags`
  - `Rcx, Rdx, R8, R9`
- `_KTHREAD`
  - `KernelStack`
  - `PreviousMode`
- `_KPRCB`
  - `CurrentThread`

These structures ensure that all user register state is treated as **untrusted input** once the transition completes.

2.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

User-Mode Assembly Preparation

1. `EAX` ← System Service Index
2. `RCX, RDX, R8, R9` ← First four arguments
3. `RCX` mirrored into `R10`

Hardware Transition

1. `syscall` executed
2. CPU loads:
    - `RIP` ← `IA32_LSTAR`
    - `CS, SS` ← Kernel selectors from `IA32_STAR`
    - `RSP` ← Kernel stack from `TSS`
3. CPL transitions **3** → **0**

Kernel Dispatch

1. Entry into `nt!KiSystemCall64`
2. Construction of `_KTRAP_FRAME`
3. SSDT index from `EAX`
4. Execution of `Nt*` service
5. Exit preparation
6. `sysret` executed

2.2.5 Secure Kernel / VBS / HVCI Interaction

On Windows 11 with VBS enabled:

- `IA32_LSTAR` target page is validated by Secure Kernel (VTL1)
- All syscall entry code is HVCI-validated
- `sysret` return target must be in user-mapped executable memory

Any violation results in:

- Immediate bugcheck
- Or hypervisor-enforced execution abort

2.2.6 Performance Implications

`syscall/sysret` introduces the following unavoidable costs:

- Pipeline serialization
- Kernel stack switch
- Partial TLB effects
- Indirect branch predictor disruption
- IRQL transition overhead

Dominant contributors to syscall latency on Windows 11:

- Scheduler bookkeeping
- Memory Manager validation
- Object Manager and handle checks

2.2.7 Real Practical Example (External MASM Only)

User-Mode Native Syscall Stub

syscall.asm:

```
PUBLIC UserSyscallStub
.code

; RCX = syscall index
UserSyscallStub PROC
    mov     eax, ecx        ; Syscall ID -> EAX
    mov     r10, rcx        ; ABI requirement
    syscall
    ret
UserSyscallStub ENDP

END
```

syscall.c:

```
#include <windows.h>
#include <stdint.h>

typedef long NTSTATUS;
extern NTSTATUS UserSyscallStub(uint32_t id);

uint32_t GetSyscallId(void* NtFunc)
{
    uint8_t* p = (uint8_t*)NtFunc;
    return *(uint32_t*)(p + 4);
```

```
}

int main(void)
{
    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
    void* NtYield = GetProcAddress(ntdll, "NtYieldExecution");

    uint32_t id = GetSyscallId(NtYield);
    UserSyscallStub(id);

    return 0;
}
```

This demonstrates the **exact architectural interface boundary**. No privilege is gained by Assembly.

2.2.8 Kernel Debugging & Inspection

Set kernel entry breakpoint:

```
bp nt!KiSystemCall64
g
```

On entry:

```
r
dt nt!_KTRAP_FRAME @rsp
!thread
!process 0 1
```

These verify:

- CPL transition

- Argument capture
- Active thread and process context

2.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surface:

- Syscall argument validation

Secondary surfaces:

- IOCTL dispatch
- Handle table resolution

Hard security boundaries:

- Ring 3 $\leftrightarrow$ Ring 0
- Ring 0 $\leftrightarrow$ VTL1

2.2.10 Professional Kernel Engineering Notes

- `syscall` is not a privilege bypass mechanism.
- All targets are MSR-controlled and kernel-owned.
- All user registers are treated as hostile input.
- `sysret` requires strict stack and RIP correctness.
- Under VBS, even CPL=0 is not the highest authority.

2.3 Model Specific Registers Used by Windows 11

2.3.1 Precise Windows 11–Specific Definition

Model Specific Registers (MSRs) are **privileged, CPU-internal control registers** accessed exclusively through the `RDMSR` and `WRMSR` instructions at **CPL=0**. On Windows 11, MSRs form the **hardware control plane** for:

- System call entry and exit
- Kernel stack switching
- RFLAGS sanitization on privilege transitions
- Segment selector virtualization
- Time and performance accounting
- Secure Kernel and hypervisor coordination

User-mode software at CPL=3 **cannot read or write MSRs**. Any attempt results in a **#GP fault**. All MSR programming on Windows 11 is performed by:

- `ntoskrnl.exe`
- `hal.dll`
- Hyper-V hypervisor

2.3.2 Exact Architectural Role Inside the Windows 11 Kernel

MSRs provide Windows 11 with **direct, microarchitectural control** over execution transitions and timing:

- **System Call Routing** via `IA32_LSTAR`
- **Privilege-Safe Segment Switching** via `IA32_STAR`
- **RFLAGS Sanitization** via `IA32_FMASK`
- **Kernel Stack Enforcement** via TSS + MSR-assisted entry
- **Timekeeping and Scheduling** via `IA32_TSC` and related registers

The kernel does not rely on software-based dispatch for privilege transitions. All transitions are **hardwired through MSRs** and validated by the CPU and hypervisor.

2.3.3 Internal Kernel Data Structures Related to MSR Control

Windows 11 does not store MSRs inside visible memory structures, but MSR-driven state is reflected inside the following real kernel structures:

- `_KTRAP_FRAME`
  - Captures the post-MSR transition CPU state
- `_KPRCB`
  - Per-processor syscall dispatch state
- `_KTHREAD`
  - `KernelStack` used after MSR-based entry

MSRs themselves reside only inside the CPU core and are not memory-mapped structures.

2.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

System Call Entry Using MSRs

1. User-mode code executes at **CPL=3**
2. `EAX` ← System call index
3. `RCX` → `R10`
4. `syscall` instruction executes
5. CPU loads:
    - `RIP` ← `IA32_LSTAR`
    - `CS, SS` ← selectors encoded in `IA32_STAR`
    - `RFLAGS` masked by `IA32_FMASK`
    - `RSP` ← Kernel stack from TSS
6. CPL transitions **3** → **0**
7. Entry into `nt!KiSystemCall64`

System Call Return

1. Kernel prepares return state
2. `sysret` executed
3. CPU restores:
    - User `RIP`
    - User `CS, SS`

- User `RFLAGS`

4. CPL transitions **0** $\rightarrow$ **3**

2.3.5 Secure Kernel / VBS / HVCI Interaction

On Windows 11 with VBS and HVCI enabled:

- `IA32_LSTAR` target address is **validated by the Secure Kernel (VTL1)**
- Kernel entry pages must be:
  - Read-only
  - Executable
  - Hypervisor-validated
- Any unauthorized modification of MSR-controlled targets triggers:
  - Hypervisor violation
  - Immediate bugcheck

Even code executing at CPL=0 in VTL0 cannot override Secure Kernel MSR policy.

2.3.6 Performance Implications

MSR-assisted transitions introduce the following costs:

- Pipeline serialization during `syscall/sysret`
- Branch prediction disruption on privilege boundary
- Kernel stack switch

- Partial TLB effects

However, the dominant performance costs in Windows 11 system calls come from:

- Object Manager validation
- Handle table resolution
- Scheduler bookkeeping
- Memory Manager access checks

MSR read/write operations themselves are **microcoded and expensive** and are therefore used only during:

- Early boot
- CPU bring-up
- Power-state transitions

2.3.7 Real Practical Example (External Assembly Only)

Kernel-Mode MSR Read Using External MASM

rdmsr.asm:

```
PUBLIC ReadMsr64
.code

; RCX = MSR index
; Returns: RDX:RAX = MSR value
ReadMsr64 PROC
```

```
    mov     ecx, ecx
    rdmsr
    shl     rdx, 32
    or      rax, rdx
    ret
ReadMsr64 ENDP

END
```

Kernel C Caller (Driver Context):

```
#include <ntddk.h>

extern UINT64 ReadMsr64(UINT32 msr);

VOID ReadLstarExample(VOID)
{
    UINT64 lstar = ReadMsr64(0xC0000082); // IA32_LSTAR
    UNREFERENCED_PARAMETER(lstar);
}
```

This example is **architecturally illegal in user mode** and will generate a **#GP fault if attempted at CPL=3**.

2.3.8 Kernel Debugging & Inspection

Inspect MSR-Driven Entry Targets:

```
r msr
!syscall
bp nt!KiSystemCall64
```

Inspect Trap Frame After MSR Transition:

```
dt nt!_KTRAP_FRAME @rsp
```

Inspect Active PRCB:

```
dt nt!_KPRCB
```

These commands confirm:

- MSR-controlled kernel entry
- Saved user-mode execution state
- Processor-local syscall dispatch context

2.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack objective:

- Subvert MSR-controlled kernel entry

Realistic attack surfaces:

- Third-party driver misuse of `WRMSR`
- Hypervisor configuration vulnerabilities
- Speculative execution abuse around MSR transitions

Hard security boundaries:

- CPL=3 $\leftrightarrow$ CPL=0
- VTL0 $\leftrightarrow$ VTL1
- Secure Kernel $\leftrightarrow$ Hyper-V

2.3.10 Professional Kernel Engineering Notes

- MSRs are the **hardware backbone** of Windows 11 privilege control.
- `IA32_LSTAR`, `IA32_STAR`, and `IA32_FMASK` must be assumed **immutable after boot**.
- Any runtime MSR modification must be:
  - Hypervisor-approved
  - Per-CPU synchronized
- Never expose `RDMSR/WRMSR` functionality through IOCTLs.
- Under VBS, even trusted kernel drivers operate below Secure Kernel authority.

2.4 Stack Switching Between User and Kernel Modes

2.4.1 Precise Windows 11–Specific Definition

On Windows 11 x86-64, **stack switching** is the hardware-enforced transition from a **user-mode stack (CPL=3)** to a **kernel-mode stack (CPL=0)** during:

- `syscall` entry
- Interrupt delivery
- Exception dispatch

The kernel stack is **not shared with user mode**. It is selected per-thread from the `_KTHREAD::KernelStack` field and activated through **TSS-assisted stack switching combined with MSR-driven entry**. This guarantees:

- Complete isolation of kernel control flow
- Immunity from user-mode stack corruption
- Deterministic interrupt and syscall entry context

2.4.2 Exact Architectural Role Inside the Windows 11 Kernel

Stack switching is the **first operation** performed on every CPL3 → CPL0 transition. Its architectural role is to:

- Transfer execution to a **trusted, per-thread kernel stack**
- Ensure all kernel code executes with:
  - A clean, non-user-controlled stack
  - Proper alignment for SIMD and ABI compliance
- Provide a safe base for:
  - `_KTRAP_FRAME` construction
  - IRQL management
  - Interrupt nesting

The user-mode stack pointer (`RSP`) is treated as **hostile input** once the transition begins.

2.4.3 Internal Kernel Data Structures (Real Structs & Fields)

The following real Windows 11 kernel structures govern stack switching:

- `_KTHREAD`

  - `KernelStack` Base of the current kernel stack
  - `InitialStack` Top of the thread's kernel stack
  - `StackLimit` Lower bound of kernel stack

- `_KTRAP_FRAME`
  - `Rsp` Saved user stack pointer
  - `Rip` Saved user instruction pointer
  - `SegCs`, `SegSs`
  - General-purpose registers

- `_KTSS64`
  - `Rsp0` Kernel stack pointer for CPL0 entry

Each logical processor maintains its own TSS, and Windows updates `Rsp0` on every context switch.

2.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

Syscall-Induced Stack Switch

1. User code executes at **CPL=3** using the user-mode `RSP`.
2. `syscall` instruction is executed.
3. CPU loads:
   - `RIP` from `IA32_LSTAR`
   - `CS, SS` from `IA32_STAR`
   - `RFLAGS` masked by `IA32_FMASK`

4. CPU loads `RSP` from:
    - `KTSS64::Rsp0` of the current processor
5. CPL transitions **3** $\rightarrow$ **0**.
6. `nt!KiSystemCall64` begins execution on the **kernel stack**.

Trap Frame Construction

1. Kernel entry code reserves space on the kernel stack.
2. User registers are copied into a `_KTRAP_FRAME`.
3. Saved user `RSP` is preserved inside the trap frame.
4. Further kernel execution proceeds at IRQL $\geq$ PASSIVE_LEVEL.

2.4.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled on Windows 11:

- Kernel stacks reside in **hypervisor-protected memory**
- Stack pages are mapped as:
    - Supervisor-only
    - Non-executable (NX where applicable)
- Secure Kernel verifies:
    - Validity of kernel stack pointers
    - Integrity of syscall entry code

Any attempt to redirect the kernel stack pointer to user memory results in:

- Immediate bugcheck
- Or hypervisor-enforced execution termination

2.4.6 Performance Implications

Stack switching incurs the following microarchitectural costs:

- Pipeline flush due to privilege transition
- Kernel stack cache-line coldness
- Partial TLB impact on kernel stack pages

Dominant contributors to syscall and interrupt latency remain:

- Scheduler bookkeeping
- Memory Manager validation
- Object Manager access checks

Kernel stack switching itself is **deterministic and constant-time**.

2.4.7 Real Practical Example (External Assembly Only)

Observing Stack Switching via External MASM Stub

stackprobe.asm:

```
PUBLIC ReadRsp
.code

ReadRsp PROC
    mov     rax, rsp
    ret
ReadRsp ENDP

END
```

stackprobe.c:

```
#include <windows.h>
#include <stdint.h>
#include <stdio.h>

extern uint64_t ReadRsp(void);

int main(void)
{
    uint64_t user_rsp = ReadRsp();
    printf("User RSP: 0x%llX\n", user_rsp);

    Sleep(0);  // Forces syscall transition

    uint64_t user_rsp_after = ReadRsp();
    printf("User RSP After: 0x%llX\n", user_rsp_after);

    return 0;
}
```

Observed Behavior:

- User-mode `RSP` is preserved across syscalls
- Kernel uses a **separate, invisible stack**

Kernel Perspective (Driver Context)

From within a driver:

```
PKTHREAD thread = KeGetCurrentThread();
PVOID kernel_stack = thread->KernelStack;
```

This pointer never overlaps any user-mode virtual memory.

2.4.8 Kernel Debugging & Inspection

Breakpoint on Syscall Entry:

```
bp nt!KiSystemCall64
g
```

Inspect Kernel Stack Pointer:

```
r rsp
dt nt!_KTHREAD @rax
dt nt!_KTRAP_FRAME @rsp
```

Inspect TSS Stack Pointer:

```
dt nt!_KTSS64
```

These commands confirm:

- Active kernel-mode `RSP`
- Saved user-mode `RSP`
- Processor-local kernel stack base

2.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary exploitation targets:

- Stack overflow in kernel stack frames
- Faulty trap frame validation
- Incorrect `KernelStack` updates during context switch

Attack goals:

- Kernel ROP chain construction
- Stack pivot into attacker-controlled memory

Hard security boundaries:

- User Stack ↔ Kernel Stack
- VTL0 ↔ VTL1

2.4.10 Professional Kernel Engineering Notes

- The kernel stack is **never shared** with user mode.
- `Rsp0` must be updated on every thread context switch.
- Kernel stacks must be mapped as supervisor-only.
- Any driver that corrupts `_KTHREAD::KernelStack` will cause an unrecoverable system failure.
- Kernel stack exhaustion leads to immediate bugcheck.
- Under VBS, kernel stacks are hypervisor-protected.

2.5 Instruction-Level Analysis of Execution Transitions

2.5.1 Precise Windows 11–Specific Definition

Instruction-level execution transitions in Windows 11 x86-64 refer to the **exact CPU instruction sequences that cause controlled privilege changes** between:

- **User Mode (CPL=3)**
- **Kernel Mode (CPL=0)**
- **Secure Kernel (VTL1, via Hyper-V)**

The only architecturally valid transition mechanisms used by Windows 11 are:

- `syscall/sysret` Native system service interface
- `int n` Legacy, not used for system calls
- Interrupt and exception gates Hardware-driven
- VM-Entry / VM-Exit Hypervisor-enforced transitions

Windows 11 **does not use call gates** and **does not permit software-defined CPL changes**. All transitions are **CPU-enforced and MSR-controlled**.

2.5.2 Exact Architectural Role Inside the Windows 11 Kernel

Instruction-level transitions form the **only legal execution bridges** between protection domains. Their architectural roles are:

- Enforcing **hardware isolation** between user and kernel execution

- Establishing a **clean kernel execution context**
- Triggering:
  - Stack switching
  - Trap frame construction
  - IRQL elevation
  - Secure Kernel validation (if VBS enabled)
- Preventing:
  - Arbitrary ring escalation
  - User-controlled return addresses in kernel mode

Every Windows 11 kernel entry point is reachable **only** through one of these instruction-level transitions.

2.5.3 Internal Kernel Data Structures (Real Structs & Fields)

Execution transitions populate and consume the following real Windows 11 kernel structures:

- `_KTRAP_FRAME`
  - `Rip`
  - `Rsp`
  - `EFlags`
  - `SegCs`
  - `SegSs`
  - `Rcx, Rdx, R8, R9`

- `_KTHREAD`
  - `KernelStack`
  - `PreviousMode`
  - `TrapFrame`
- `_KPRCB`
  - `CurrentThread`
  - `DpcRoutineActive`
- `_KTSS64`
  - `Rsp0`

These structures are **never exposed to user mode** and are mapped as supervisor-only.

2.5.4 Execution Flow (Step-by-Step at C & Assembly Level)

User-to-Kernel Transition via `syscall`

Instruction-Level Sequence:

1. User code executes at CPL=3.
2. System call number placed in `EAX`.
3. First argument placed in `RCX`.
4. ABI-mandated copy: `RCX` $\rightarrow$ `R10`.
5. `syscall` instruction executed.

Hardware Effects (Atomic):

- `RIP` ← `IA32_LSTAR`
- `CS/SS` ← Kernel selectors from `IA32_STAR`
- `RSP` ← `KTSS64::Rsp0`
- `RFLAGS` masked by `IA32_FMASK`
- CPL transitions **3** → **0**

Kernel Software Effects:

1. Entry at `nt!KiSystemCall64`
2. `_KTRAP_FRAME` constructed
3. System Service Number resolved
4. Target `Nt*` routine executed

Kernel-to-User Transition via `sysret`

1. Kernel restores:
    - User `RIP` from trap frame
    - User `RSP`
    - User `RFLAGS`
2. `sysret` instruction executed.
3. CPL transitions **0** → **3**.

Interrupt and Exception Transitions

- CPU consults IDT.
- Gate descriptor enforces CPL rules.
- Stack switch via TSS.
- Trap frame constructed.
- Handler executes at IRQL $>$ PASSIVE_LEVEL.

2.5.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled on Windows 11:

- All kernel entry addresses from `IA32_LSTAR` are validated by the Secure Kernel (VTL1).
- Kernel entry code pages must be:
  - Read-only
  - Hypervisor-signed
  - Non-writable
- `syscall` targets cannot be redirected by any VTL0 component.

Hypervisor transitions:

- Occur via VM-Exit/VM-Entry.
- Are invisible to VTL0 software.
- Enforce Secure Kernel isolation.

2.5.6 Performance Implications

Instruction-level transitions impose unavoidable costs:

- Pipeline serialization
- Kernel stack cache cold-start
- Partial TLB pressure
- Indirect branch predictor invalidation

In Windows 11, total syscall cost is dominated by:

- Object Manager validation
- Handle table resolution
- Security descriptor checks
- Scheduler accounting

The instruction-level transition itself is **not the dominant performance bottleneck**.

2.5.7 Real Practical Example (External Assembly Only)

External MASM User-Mode Syscall Stub

usersyscall.asm:

```
PUBLIC RawSyscall
.code

; RCX = syscall index
```

```
; RDX = arg1
RawSyscall PROC
    mov     eax, ecx
    mov     r10, rcx
    syscall
    ret
RawSyscall ENDP

END
```

usersyscall.c:

```
#include <windows.h>
#include <stdint.h>

typedef long NTSTATUS;
extern NTSTATUS RawSyscall(uint32_t id, void* arg1);

uint32_t ExtractSyscallId(void* ntfunc)
{
    return *(uint32_t*)((uint8_t*)ntfunc + 4);
}

int main(void)
{
    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
    void* NtQuery = GetProcAddress(ntdll, "NtYieldExecution");

    uint32_t id = ExtractSyscallId(NtQuery);
    RawSyscall(id, NULL);
```

```
    return 0;
}
```

This example demonstrates:

- Instruction-level transition via `syscall`
- No direct kernel entry control
- No privilege bypass

2.5.8 Kernel Debugging & Inspection

Breakpoint at Instruction-Level Entry:

```
bp nt!KiSystemCall64
g
```

Inspect Trap Frame:

```
dt nt!_KTRAP_FRAME @rsp
```

Inspect Current Thread and Stack:

```
!thread
r rsp
```

Verify Secure Entry Protection:

```
!vbs
!hypervisor
```

2.5.9 Exploitation Surface, Attack Vectors & Security Boundaries

Attack surfaces at instruction level:

- Speculative execution windows at transition boundaries
- Trap frame corruption via vulnerable drivers
- Improper syscall parameter validation

Attacker objectives:

- Kernel stack pivot
- Return-oriented programming (ROP)
- Controlled trap frame overwrite

Hard security boundaries:

- CPL=3 $\leftrightarrow$ CPL=0
- VTL0 $\leftrightarrow$ VTL1

These boundaries are **hardware-enforced** and cannot be bypassed by software alone.

2.5.10 Professional Kernel Engineering Notes

- All privilege transitions are instruction-encoded and CPU-enforced.
- No legitimate Windows 11 kernel path allows:
  - Software CPL changes
  - User-controlled kernel `RIP`

- Every kernel vulnerability ultimately manifests at:
  - Trap frame handling
  - Stack switching
  - Instruction-level return paths
- Under VBS, even CPL=0 code is subordinate to Secure Kernel authority.

Part II

Windows 11 Boot Process (From Power-On to `ntoskrnl.exe`)

Chapter 3

UEFI, Secure Boot & TPM from a Low-Level Perspective

3.1 What an Assembly Engineer Actually Sees During Boot

3.1.1 Precise Windows 11–Specific Definition

From an Assembly engineers perspective, the Windows 11 boot process is a **strictly staged, hardware-verified execution chain** that transitions through:

- **UEFI Firmware Execution (64-bit Long Mode)**
- **UEFI Secure Boot Verification**
- **Windows Boot Manager (`bootmgfw.efi`)**
- **Windows OS Loader (`winload.efi`)**
- **Hypervisor Launch (if VBS enabled)**

- **Kernel Image Transfer to `ntoskrnl.exe`**

At no point does an Assembly engineer observe real-mode BIOS or DOS-style execution. Windows 11 **never executes in real mode after firmware reset**. All visible execution is:

- 64-bit long mode
- Paging enabled
- Firmware-controlled GDT/IDT
- Identity-mapped physical memory during early loader phase

3.1.2 Exact Architectural Role Inside the Windows 11 Kernel

This boot phase determines:

- Authorization of `ntoskrnl.exe` for execution
- Enforcement of Secure Boot trust chain
- Early Hyper-V injection before kernel execution
- Existence and isolation of the Secure Kernel (VTL1)
- Initial physical memory layout handed to the Memory Manager

From a kernel architecture standpoint, **every Windows 11 security guarantee originates at this stage**. If this phase is compromised, all subsequent kernel security mechanisms are rendered ineffective.

3.1.3 Internal Kernel Data Structures (Real Structs & Fields)

During this phase, no scheduler or executive structures exist. The primary kernel-facing structure created by the loader is:

- `_LOADER_PARAMETER_BLOCK`
  - `LoadOrderListHead`
  - `MemoryDescriptorListHead`
  - `KernelStack`
  - `Extension`
- `_LOADER_MEMORY_DESCRIPTOR`
  - `BasePage`
  - `PageCount`
  - `MemoryType`

These structures form the **initial authoritative physical memory map** used by the Windows 11 Memory Manager during early kernel initialization.

3.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

UEFI Firmware Execution

1. CPU reset vector transfers control into UEFI firmware.
2. Firmware executes fully in **64-bit long mode**.
3. Paging is already enabled.

4. Firmware stack and runtime services are initialized.

5. DXE phase initializes Secure Boot policy.

Secure Boot Verification

1. Firmware validates `bootmgfw.efi` using:

    - UEFI DB
    - UEFI KEK

2. Execution halts on signature failure.

3. On success, control transfers to `bootmgfw.efi`.

Windows Boot Manager

1. Reads the BCD configuration.

2. Resolves active boot entry.

3. Loads `winload.efi`.

4. Passes EFI system table and memory map.

Windows OS Loader

1. Verifies:

    - `ntoskrnl.exe`
    - `hal.dll`
    - Boot-class drivers

2. Constructs `_LOADER_PARAMETER_BLOCK`.
3. Enables final kernel page tables.
4. Transfers execution to `ntoskrnl.exe`.

3.1.5 Secure Kernel / VBS / HVCI Interaction

If VBS is enabled:

- Hyper-V is launched **before** `ntoskrnl.exe`
- Secure Kernel (VTL1) is verified and initialized
- Kernel execution in VTL0 becomes subordinate to VTL1
- HVCI enforces:
  - No writable-executable kernel code
  - Mandatory driver signature enforcement

At the instruction level this manifests as:

- VMX root mode transitions
- Hypervisor-controlled MSR behavior

3.1.6 Performance Implications

Boot-time performance is dominated by:

- TPM PCR measurements
- UEFI signature validation latency

- Hypervisor initialization time
- Boot-class driver signature validation

Execution density during this stage is **cryptography- and I/O-bound**, not computation-bound.

3.1.7 Real Practical Example (External Assembly Only)

Minimal UEFI x64 Assembly Entry Point

uefi_entry.asm:

```
PUBLIC efi_main
.code

; RCX = EFI_HANDLE ImageHandle
; RDX = EFI_SYSTEM_TABLE* SystemTable

efi_main PROC
    sub     rsp, 28h        ; Shadow space + alignment

    ; Long mode active
    ; Paging enabled
    ; Secure Boot already enforced

    add     rsp, 28h
    xor     eax, eax
    ret
efi_main ENDP

END
```

This entry point is already executing:

- With full paging enabled
- Without any access to real mode
- Without access to kernel virtual memory

3.1.8 Kernel Debugging & Inspection

Enable Boot Debugging:

```
bcdedit /set {default} debug on
bcdedit /set {default} bootdebug on
```

Inspect Loader State in WinDbg:

```
!loader
dt nt!_LOADER_PARAMETER_BLOCK
```

Verify Secure Boot and VBS:

```
!vbs
!securekernel
!hypervisor
```

3.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Malicious UEFI DXE drivers
- TPM PCR replay attacks

- Compromised Secure Boot key material

Security boundaries:

- Firmware ↔ Windows Boot Manager
- Secure Kernel (VTL1) ↔ Kernel (VTL0)

This represents the **strongest trust boundary in the entire Windows 11 architecture**.

3.1.10 Professional Kernel Engineering Notes

- Windows 11 never executes in real mode.
- Assembly engineers never see BIOS interrupts.
- All execution occurs under paging and long mode.
- Secure Boot validation occurs **before any Windows code executes**.
- With VBS enabled, the hypervisor precedes the kernel.
- Any compromise at this stage invalidates all later kernel security.

3.2 Digital Signature Verification

3.2.1 Precise Windows 11–Specific Definition

Digital signature verification in the Windows 11 boot chain is the **cryptographically enforced validation process** that guarantees every executable componentfrom `bootmgfw.efi` to `ntoskrnl.exe` and early boot driversoriginates from a **trusted, policy-authorized signer**. Verification is performed using:

- UEFI Secure Boot signature enforcement in firmware
- Authenticode validation in `winload.efi`
- TPM-backed PCR measurements
- Hypervisor-enforced Code Integrity (HVCI), if enabled

Any failure at any stage results in **immediate boot termination**.

3.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Digital signature verification defines the **root of execution trust** for the kernel. Architecturally, it:

- Prevents execution of unsigned:
  - Boot loaders
  - Kernel images
  - Boot-class drivers
- Establishes:
  - Kernel trust level
  - Driver trust level
- Determines whether:
  - VBS is permitted
  - HVCI is enforced

Once `ntoskrnl.exe` is entered, the kernel **assumes cryptographic integrity is already proven**.

3.2.3 Internal Kernel Data Structures (Real Structs & Fields)

Signature verification itself occurs **before most kernel executive structures exist**. However, the results influence:

- `_LOADER_PARAMETER_BLOCK`
  - `LoadOrderListHead`
  - `Extension`
- `_KLDR_DATA_TABLE_ENTRY`
  - `DllBase`
  - `SizeOfImage`
  - `EntryPoint`
  - `Flags`

Later, kernel-mode Code Integrity relies on:

- `CI!g_CiOptions`
- `CI!CiValidateFileObject`

3.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

UEFI Secure Boot Verification

1. Firmware loads `bootmgfw.efi`.
2. The PE image signature is validated against:
   - UEFI `db`

- UEFI `KEK`

3. SHA-256 hash is verified using embedded X.509 certificate.
4. If verification fails, firmware halts the boot.

Windows Boot Loader Verification

1. `bootmgfw.efi` loads `winload.efi`.
2. `winload.efi` verifies:
   - `ntoskrnl.exe`
   - `hal.dll`
   - Early boot drivers
3. Authenticode signature parsing:
   - WIN_CERTIFICATE
   - PKCS#7 blob
4. Public key chain validated against Microsoft Root CA.

TPM Measurement

1. Each verified image hash is extended into TPM PCRs:
   - PCR[0] Firmware
   - PCR[4] Boot manager
   - PCR[7] Secure Boot policy
2. PCRs become immutable evidence of boot integrity.

3.2.5 Secure Kernel / VBS / HVCI Interaction

When VBS is enabled:

- Secure Kernel (VTL1) re-validates:
  - `ntoskrnl.exe`
  - Code Integrity policies
- HVCI enforces:
  - No unsigned kernel drivers
  - No self-modifying kernel code
- VMX transitions separate:
  - Signature enforcement logic in VTL1
  - Kernel execution in VTL0

VTL0 kernel code **cannot bypass** VTL1 signature policy.

3.2.6 Performance Implications

Digital signature verification is:

- Cryptography-dominated
- I/O-bound

Performance cost contributors:

- RSA/ECDSA certificate validation

- SHA-256 hashing of large binaries
- TPM PCR extension latency

This cost is incurred **once per boot** and is not part of runtime kernel execution overhead.

3.2.7 Real Practical Example (External Assembly Only)

Observing PE Certificate Table in UEFI Context

pe_cert_scan.asm:

```
PUBLIC ScanPeCertificate
.code

; RCX = ImageBase
; Returns RAX = Certificate Directory RVA
ScanPeCertificate PROC
    mov     rax, rcx
    mov     edx, DWORD PTR [rax + 3Ch]    ; e_lfanew
    add     rax, rdx                      ; NT Headers

    add     rax, 18h                      ; Optional Header start
    add     rax, 70h                      ; DataDirectory[4]
     ↪  (Security)

    mov     eax, DWORD PTR [rax]          ; Certificate Table RVA
    ret
ScanPeCertificate ENDP

END
```

This code demonstrates that **certificate discovery occurs purely at the PE structure level**, independent of Windows APIs.

3.2.8 Kernel Debugging & Inspection

Verify Secure Boot State:

```
!secureboot
!vbs
```

Inspect Code Integrity Options:

```
x ci!g_CiOptions
dq ci!g_CiOptions
```

Inspect Loaded Kernel Modules:

```
lm
!driverinfo
```

Inspect TPM Measurements (External Attestation):

```
tpmtool getdeviceinformation
```

3.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Compromised UEFI `db` or `KEK`
- Malicious boot loader replacement
- TPM PCR spoofing in broken firmware

Attack objectives:

- Execute unsigned kernel code
- Disable Code Integrity enforcement
- Subvert Secure Kernel trust chain

Hard security boundaries:

- Firmware ↔ Boot Loader
- Secure Kernel (VTL1) ↔ Kernel (VTL0)

3.2.10 Professional Kernel Engineering Notes

- Windows 11 assumes all boot components are cryptographically verified before kernel execution.
- Kernel Code Integrity is a continuation of Secure Boot, not a replacement.
- TPM PCRs provide **forensic-grade boot evidence**.
- Any attempt to bypass signature verification invalidates:
  - VBS
  - HVCI
  - Measured Boot
- No kernel-mode driver can disable Secure Boot enforcement once the system is locked.

3.3 Memory Analysis During Early Boot

3.3.1 Precise Windows 11–Specific Definition

Early-boot memory analysis in Windows 11 refers to the **forensic inspection, classification, and handoff of physical memory regions** performed across the following execution phases:

- UEFI firmware (DXE and BDS phases)
- Windows Boot Manager (`bootmgfw.efi`)
- Windows OS Loader (`winload.efi`)
- Early kernel initialization in `ntoskrnl.exe`

At this stage, memory is managed exclusively as **physical pages**, described through firmware-provided memory maps and loader-constructed descriptors. No virtual memory abstractions such as:

- Process address spaces
- VADs
- Page file backing

exist yet. Only **identity-mapped or loader-mapped paging structures** are in effect.

3.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Early memory analysis defines the **initial trust boundary for all future memory management**. Its architectural roles are:

- Filtering firmware memory maps into Windows-specific memory types

- Reserving:
  - Kernel image regions
  - HAL image regions
  - Boot-class driver memory
- Constructing the authoritative:
  - Physical memory ownership model
  - Initial PFN database layout
- Providing the Memory Manager with:
  - Usable RAM ranges
  - Firmware-reserved ranges
  - ACPI and MMIO exclusions

All later virtual memory policy, NUMA topology, and page coloring strategies are **built on this early classification**.

3.3.3 Internal Kernel Data Structures (Real Structs & Fields)

The following real structures define the memory model entering `ntoskrnl.exe`:

- `_LOADER_PARAMETER_BLOCK`
  - `MemoryDescriptorListHead`
  - `Extension`
- `_LOADER_MEMORY_DESCRIPTOR`

    - BasePage
    - PageCount
    - MemoryType
- _PHYSICAL_MEMORY_RUN
    - BasePage
    - PageCount

Once the Memory Manager initializes, these are transformed into:

- MiPhysicalMemoryBlock
- Per-page _MMPFN entries in the PFN database

3.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

UEFI Firmware Memory Map Generation

1. Firmware constructs EFI memory map.
2. Each entry classified as:
    - EfiConventionalMemory
    - EfiBootServicesCode
    - EfiRuntimeServicesData
    - EfiACPIReclaimMemory
    - EfiMemoryMappedIO
3. Memory map passed to bootmgfw.efi.

Windows Loader Memory Filtering

1. `winload.efi` converts EFI entries into:
    - LoaderFree
    - LoaderFirmwareTemporary
    - LoaderFirmwarePermanent
    - LoaderBootDriver
    - LoaderLoadedProgram
2. Kernel, HAL, and drivers are placed into physically contiguous regions.
3. `_LOADER_MEMORY_DESCRIPTOR` list is built.

Early Kernel Memory Handoff

1. `ntoskrnl.exe` receives `_LOADER_PARAMETER_BLOCK`.
2. Memory Manager parses:
    - Usable RAM
    - Reserved firmware regions
    - MMIO holes
3. Initial page tables are rebuilt.
4. PFN database is constructed.

3.3.5 Secure Kernel / VBS / HVCI Interaction

If VBS is enabled:

- Hypervisor creates isolated memory for:
  - Secure Kernel (VTL1)
  - HVCI enforcement engine
- Loader memory descriptors include:
  - Secure VTL1 regions
  - Hypervisor-reserved memory
- These regions are:
  - Supervisor-only
  - Not mapped into VTL0 page tables

The VTL0 kernel never directly controls:

- VTL1 page tables
- Secure Kernel PFN ownership

3.3.6 Performance Implications

Early-boot memory handling directly influences:

- NUMA locality detection
- Cache coloring efficiency

- TLB shootdown behavior
- Initial paging structure density

Performance penalties may arise from:

- Large MMIO holes
- Fragmented physical memory descriptors
- Excessive firmware-reserved regions

These penalties persist for the entire system uptime.

3.3.7 Real Practical Example (External Assembly Only)

Scanning Loader Memory Descriptors in Early Kernel Context

scan_ldr_mem.asm:

```
PUBLIC ScanLoaderMemory
.code

; RCX = Pointer to _LOADER_PARAMETER_BLOCK
ScanLoaderMemory PROC
    mov     rax, [rcx + 50h]    ; MemoryDescriptorListHead.Flink
scan_loop:
    mov     rdx, [rax]          ; Next entry
    cmp     rdx, rcx
    je      scan_done
    mov     rax, rdx
    jmp     scan_loop
```

```
scan_done:
    ret
ScanLoaderMemory ENDP

END
```

This code demonstrates that **early memory analysis is purely pointer-based and page-granular** before virtual abstractions exist.

3.3.8 Kernel Debugging & Inspection

Inspect Loader Memory Map:

```
!loader
dt nt!_LOADER_PARAMETER_BLOCK
```

Inspect Memory Descriptors:

```
dt nt!_LOADER_MEMORY_DESCRIPTOR
```

Inspect Physical Memory Layout:

```
!memusage
!pte
```

Verify Hypervisor Memory Isolation:

```
!vbs
!hypervisor
```

3.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Malicious manipulation of EFI memory maps
- Loader memory descriptor corruption
- Hypervisor memory reservation bypass

Attack objectives:

- Mapping attacker-controlled physical pages as kernel RAM
- Hiding malicious MMIO-backed memory
- Subverting PFN database initialization

Hard security boundaries:

- Firmware memory map $\leftrightarrow$ Loader filtering
- VTL1 Secure Memory $\leftrightarrow$ VTL0 Kernel Memory

3.3.10 Professional Kernel Engineering Notes

- All early memory decisions are irreversible after PFN initialization.
- Incorrect firmware memory typing results in permanent kernel instability.
- Kernel memory cannot overlap:
  - ACPI regions
  - PCIe BAR MMIO windows
  - Hypervisor-reserved RAM
- Under VBS, Secure Kernel memory is never visible in the VTL0 PFN database.
- Any corruption in loader memory descriptors leads to non-recoverable boot failure.

Chapter 4

Windows Boot Manager, Boot Applications & Kernel Loading

4.1 Loading `ntoskrnl.exe`

4.1.1 Precise Windows 11–Specific Definition

Loading `ntoskrnl.exe` in Windows 11 is the **cryptographically enforced, loader-mediated PE image mapping process** performed by `winload.efi` after Secure Boot validation and before any executive kernel subsystem exists. This phase performs:

- Physical placement of the kernel image
- PE header parsing and section mapping
- Relocation processing
- Import resolution (static, loader-time)

- Creation of the initial kernel address space

All operations occur **outside of the running Windows kernel**. No scheduler, no Memory Manager, no Object Manager, and no IRQL model exists yet.

4.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Architecturally, this phase establishes:

- The **first executable kernel virtual address**
- The **initial kernel page table hierarchy**
- The **permanent physical location of kernel code and data**
- The **trust boundary transition** from firmware/loader to kernel ownership

Once control is transferred to `ntoskrnl.exe`, the loader will never regain execution authority. From that point forward:

- Page ownership is enforced by the Memory Manager
- Code execution is governed by Kernel Code Integrity
- Control-flow integrity is enforced by VBS/HVCI if enabled

4.1.3 Internal Kernel Data Structures (Real Structs & Fields)

At kernel entry, `ntoskrnl.exe` receives a fully constructed **loader context** containing:

- `_LOADER_PARAMETER_BLOCK`
  - `KernelStack`

  - `MemoryDescriptorListHead`
  - `LoadOrderListHead`
  - `Extension`

- `_KLDR_DATA_TABLE_ENTRY` (for `ntoskrnl.exe`)
  - `DllBase`
  - `EntryPoint`
  - `SizeOfImage`
  - `Flags`

These structures represent the **only authoritative kernel image metadata** during early initialization.

4.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Image Discovery and Verification

1. `bootmgfw.efi` locates `winload.efi`.
2. `winload.efi` parses BCD entries.
3. `ntoskrnl.exe` location is resolved.
4. Authenticode signature is verified.

Physical Placement and Mapping

1. Kernel image read from disk into **physically contiguous memory**.
2. PE headers parsed.

3. Sections mapped:

    - `.text` RX
    - `.rdata` R
    - `.data` RW
    - `.pdata`, `.reloc`

4. Relocations applied if base differs from preferred address.

Initial Kernel Address Space Construction

1. Temporary page tables replaced with kernel page tables.
2. Higher-half kernel mapping created.
3. Identity mappings removed where no longer required.

Transfer of Control to Kernel Entry

1. Stack switched to `_LOADER_PARAMETER_BLOCK::KernelStack`.
2. CPU context initialized.
3. Execution branches to:

```
ntoskrnl.exe!KiSystemStartup
```

This is the **irreversible boundary** between loader and kernel control.

4.1.5 Secure Kernel / VBS / HVCI Interaction

If VBS is enabled:

- Hyper-V is already active before `ntoskrnl.exe` execution
- Secure Kernel (VTL1) memory is carved out before kernel mapping
- Kernel code pages are:
  - Read-only
  - Executable
  - HVCI-validated

If `ntoskrnl.exe` fails VTL1 integrity checks:

- Execution is aborted
- System enters secure bugcheck state

4.1.6 Performance Implications

Kernel image loading performance is dominated by:

- Disk I/O throughput
- SHA-256 hashing cost
- Page table construction overhead
- Relocation fixup cost

This cost occurs **once per boot**. It has no steady-state runtime performance impact.
However, early fragmentation or poor physical alignment of the kernel image affects:

- Instruction cache locality
- TLB behavior
- Cross-node NUMA fetch latency

4.1.7 Real Practical Example (External Assembly Only)

Observing PE Header Parsing at Loader Time

parse_nt_header.asm:

```
PUBLIC ParseNtHeader
.code

; RCX = ImageBase
; Returns RAX = Address of OptionalHeader
ParseNtHeader PROC
    mov     rax, rcx
    mov     edx, DWORD PTR [rax + 3Ch]    ; e_lfanew
    add     rax, rdx                      ; NT Headers
    add     rax, 18h                      ; OptionalHeader offset
    ret
ParseNtHeader ENDP

END
```

This demonstrates how `winload.efi` resolves kernel image layout without invoking any Windows kernel code.

4.1.8 Kernel Debugging & Inspection

Verify Kernel Load State:

```
!loader
lm
```

Inspect Kernel Entry Context:

```
r
dt nt!_LOADER_PARAMETER_BLOCK
```

Inspect Kernel Image Mapping:

```
!dh ntoskrnl.exe
!pte nt!KiSystemStartup
```

4.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Malicious kernel replacement
- Relocation table abuse
- Loader memory descriptor corruption

Attack objectives:

- Redirect kernel entry point
- Inject unsigned kernel code
- Corrupt initial kernel page tables

Hard security boundaries:

- Secure Boot signature enforcement
- VTL1 Secure Kernel validation
- HVCI kernel page protection

4.1.10 Professional Kernel Engineering Notes

- The kernel is never loaded by itself; it is always loader-injected.
- `ntoskrnl.exe` does not parse its own PE headers during initial load.
- All early kernel memory is pre-validated before execution.
- No driver code executes before the kernel owns the CPU.
- Any corruption in this phase results in a guaranteed boot failure.

4.2 Loading `HAL.dll`

4.2.1 Precise Windows 11–Specific Definition

Loading `HAL.dll` in Windows 11 is the **loader-resolved, kernel-linked hardware abstraction activation phase** that binds `ntoskrnl.exe` to the active platforms interrupt controllers, timers, I/O APICs, local APICs, ACPI tables, and chipset-specific power management logic. Unlike legacy systems, `HAL.dll` is no longer a user-visible pluggable hardware layer but a **cryptographically verified, tightly version-coupled executive dependency** of the kernel itself.
The file is mapped and linked by `winload.efi` **before** kernel execution begins and becomes a permanent resident of kernel virtual memory.

4.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Architecturally, `HAL.dll` provides:

- Interrupt controller programming (x2APIC, I/O APIC)

- High-precision timers (HPET, TSC Deadline Timer)
- DMA remapping and IOMMU coordination
- Power state transitions (ACPI S0S5)
- Platform-dependent spin-wait and stall primitives

The Windows 11 kernel never accesses raw chipset registers directly. All hardware-facing execution funnels through HAL entry points such as:

- `HalInitializeProcessor`
- `HalpTimerClockInterrupt`
- `HalpApicWrite`

This guarantees that scheduling, timekeeping, and interrupt delivery are **hardware-neutral at the kernel core level**.

4.2.3 Internal Kernel Data Structures (Real Structs & Fields)

Although `HAL.dll` is a separate PE image, it is integrated into kernel loader lists via:

- `_KLDR_DATA_TABLE_ENTRY` (HAL module entry)
  - `DllBase`
  - `EntryPoint`
  - `SizeOfImage`
  - `LoadReason`
- `_LOADER_PARAMETER_BLOCK`

- `LoadOrderListHead`

Once kernel execution begins, HAL function pointers are published into internal dispatch tables used by:

- Clock interrupt routing
- Inter-processor interrupts (IPI)
- Deferred procedure calls (DPC) timing

4.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Loader Phase

1. `winload.efi` resolves `HAL.dll` path from BCD.
2. Digital signature is verified under Secure Boot policy.
3. Image is copied into physically contiguous memory.
4. PE sections are mapped into kernel virtual space.
5. Relocations are applied if required.

Kernel Linkage Phase

1. Import table of `ntoskrnl.exe` is resolved against HAL exports.
2. HAL entry points are bound into kernel dispatch tables.
3. HAL image is inserted into `_KLDR_DATA_TABLE_ENTRY` lists.

Processor Bring-Up Phase

During `nt!KiSystemStartup`:

1. `HalInitializeProcessor(0)` is invoked.
2. Local APIC and timer routing are initialized.
3. Bootstrap processor interrupt delivery is armed.

At this point, the kernel becomes **interrupt-capable**.

4.2.5 Secure Kernel / VBS / HVCI Interaction

When VBS is enabled:

- `HAL.dll` is validated in VTL1 prior to kernel execution.
- Any attempt to patch HAL code pages is blocked by HVCI.
- IOMMU programming performed by HAL is validated by the hypervisor.

This prevents:

- Malicious interrupt redirection
- Timer spoofing
- DMA-based kernel memory corruption

4.2.6 Performance Implications

HAL directly influences:

- Interrupt latency (APIC vs x2APIC)
- Scheduler tick precision
- Cross-core wakeup efficiency
- Power-state transition overhead

Misconfigured HAL timer routing manifests as:

- DPC latency spikes
- Scheduler jitter on hybrid P/E-core systems
- Cross-NUMA clock drift

4.2.7 Real Practical Example (External Assembly Only)

Inspecting HAL Import Binding Logic (Loader-Level)

resolve_hal_import.asm:

```
PUBLIC ResolveHalImport
.code

; RCX = Address of NT Import Descriptor
ResolveHalImport PROC
    mov     rax, [rcx]          ; Import Name RVA
    ret
```

```
ResolveHalImport ENDP

END
```

This demonstrates the low-level import resolution mechanism used by the loader when binding `ntoskrnl.exe` against `HAL.dll`.

4.2.8 Kernel Debugging & Inspection

Verify HAL Load State:

```
lm m hal
```

Inspect HAL Loader Entry:

```
!lmi hal
```

Trace HAL Initialization:

```
bp hal!HalInitializeProcessor
g
```

4.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary historical attack vectors:

- HAL interrupt handler hooking
- Fake timer source injection
- DMA remapping bypass via HAL misconfiguration

Windows 11 hardening barriers:

- Secure Boot signature enforcement
- HVCI code integrity enforcement
- Hypervisor-controlled IOMMU programming

Direct HAL exploitation in Windows 11 requires **hypervisor-level compromise**.

4.2.10 Professional Kernel Engineering Notes

- HAL is no longer replaceable across systems in modern Windows.
- Kernel and HAL versions are cryptographically coupled.
- All interrupt topology assumptions originate from HAL.
- Any HAL malfunction destabilizes the scheduler immediately.
- Hybrid CPU scheduling correctness depends on HAL topology reporting.

4.3 The Very First Instruction Executed by the Kernel

4.3.1 Precise Windows 11–Specific Definition

The very first instruction executed by the Windows 11 kernel is the **initial instruction at the entry point of `ntoskrnl.exe`**, reached immediately after control is transferred from `winload.efi`. This instruction executes in:

- **64-bit Long Mode**
- **Ring 0 (Kernel Mode)**
- **Paging Enabled (CR0.PG = 1)**

- **PAE Enabled**
- **Identity and higher-half kernel mappings active**

This instruction is the **first CPU-visible boundary between firmware-controlled execution and kernel-controlled execution**. From this point forward, all control flow is governed exclusively by `ntoskrnl.exe` and `HAL.dll`.

4.3.2 Exact Architectural Role Inside the Windows 11 Kernel

This first instruction serves four irreversible architectural transitions:

1. Ownership transfer of the execution pipeline from UEFI Boot Services to the Windows kernel.
2. Activation of the kernels internal execution model.
3. Establishment of the kernels permanent stack model.
4. Entry into the kernels internal startup state machine.

It is the root of the entire call graph for:

- `KiSystemStartup`
- `HalInitializeProcessor`
- `KiInitializeKernel`
- `ExpInitializeExecutive`

No scheduler, memory manager, or interrupt dispatch exists before this instruction executes.

4.3.3 Internal Kernel Data Structures (Real Structs & Fields)

The very first instruction executes with these structures already populated by the loader:

- `_LOADER_PARAMETER_BLOCK`
  - `KernelStack`
  - `LoaderBlockChecksum`
  - `LoadOrderListHead`
  - `MemoryDescriptorListHead`
- `_KLDR_DATA_TABLE_ENTRY`
  - `ntoskrnl.exe`
  - `HAL.dll`
  - `bootvid.dll`
- `_KPROCESSOR_STATE`
  - `SpecialRegisters`
  - `ContextFrame`

The first instruction assumes that these are already mapped and valid.

4.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Loader-to-Kernel Transfer

1. `winload.efi` resolves kernel entry RVA from PE headers.
2. Processor state is switched to long mode execution context.

3. Stack pointer is loaded from `LOADER_PARAMETER_BLOCK::KernelStack`.
4. A near absolute jump is issued to the kernel entry.

First Instruction Target

The first executed kernel routine is:

- `nt!KiSystemStartup`

This function is the **true root of all Windows 11 kernel execution**.

Architectural State at Entry

- **RSP** → Valid kernel stack
- **CR3** → Kernel page tables
- **GS base** → Per-CPU structure
- **Interrupts** → Disabled

4.3.5 Secure Kernel / VBS / HVCI Interaction

If VBS is enabled:

- `ntoskrnl.exe` executes as a VTL0 guest under Hyper-V.
- The Secure Kernel (VTL1) already controls:
  - MSR write filtering
  - IOMMU programming
  - Code integrity enforcement

- The very first kernel instruction is already subject to:
  - HVCI validation
  - Secure kernel shadow page tables

Thus, even the first instruction executes under hypervisor surveillance.

4.3.6 Performance Implications

The first instruction determines:

- Timer routing initialization latency
- Early APIC programming overhead
- BSP to AP startup synchronization cost
- NUMA topology discovery cost

If early CPU feature detection is misconfigured at this stage:

- Scheduler tick jitter appears immediately.
- Hybrid core classification becomes unstable.

4.3.7 Real Practical Example (External Assembly Only)

Conceptual Kernel Entry Stub (Educational Reconstruction)

This example models the **conceptual structure** of a kernel entry point. It is not a replacement for the real Windows kernel.

kernel_entry.asm:

```
PUBLIC KernelEntryPoint
.code

KernelEntryPoint PROC
    cli                     ; Interrupts must be disabled
    mov     rsp, rcx        ; RCX = Kernel Stack from Loader
    sub     rsp, 20h        ; Shadow space
    call    KernelMainInit  ; Enter main kernel init path
    hlt                     ; Never returns
KernelEntryPoint ENDP

END
```

This mirrors the real execution constraints imposed on the Windows 11 kernel entry.

4.3.8 Kernel Debugging & Inspection

Trace First Kernel Instruction:

```
bp nt!KiSystemStartup
g
```

Inspect CPU State at Entry:

```
r
!cpuinfo
```

Inspect Loader Structures:

```
dt nt!_LOADER_PARAMETER_BLOCK
```

4.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack targets:

- Kernel entry patching
- Fake loader block injection
- Stack pointer redirection

Windows 11 hardening barriers:

- Secure Boot PE signature enforcement
- HVCI code page protection
- Hypervisor MSR filtering

Modern Windows 11 makes **pre-entry kernel exploitation practically dependent on firmware-level compromise**.

4.3.10 Professional Kernel Engineering Notes

- No pageable memory exists at the first instruction.
- No interrupts are legal before HAL initialization.
- No scheduler objects exist before `KiInitializeKernel`.
- All failures at this stage result in immediate triple fault or silent reset.
- Kernel debugging at this phase requires:
  - Boot-time kernel debugging
  - Serial or network KD

4.4 Assembly-Level Analysis of `KiSystemStartup`

4.4.1 Precise Windows 11–Specific Definition

`KiSystemStartup` is the **first architecturally meaningful kernel routine executed in Windows 11** after control is transferred from `winload.efi` to `ntoskrnl.exe`. It is the **root assembly-level entry for kernel CPU bring-up**, responsible for converting the loader-prepared execution context into a fully initialized kernel execution environment.
This routine executes:

- In **64-bit long mode**
- At **CPL=0 (Ring 0)**
- With **paging enabled**
- With **interrupts disabled**

It is not a scheduler routine, not a memory manager routine, and not an executive routine. It is a **pure architectural transition routine**.

4.4.2 Exact Architectural Role Inside the Windows 11 Kernel

`KiSystemStartup` performs the following irreversible architectural transitions:

1. Establishes the initial kernel stack discipline.
2. Loads the per-processor control region (KPCR via `GS`).
3. Initializes control registers and MSRs required for kernel execution.
4. Transfers control into the portable C initialization path.

From this point forward:

- HAL is callable
- Processor exceptions are routable
- The executive initialization pipeline begins

No Windows kernel service can exist before `KiSystemStartup` executes.

4.4.3 Internal Kernel Data Structures (Real Structs & Fields)

At entry to `KiSystemStartup`, the following loader-constructed structures are already valid:

- `_LOADER_PARAMETER_BLOCK`
  - `KernelStack`
  - `KernelStackSize`
  - `MemoryDescriptorListHead`
- `_KPROCESSOR_STATE`
  - `ContextFrame`
  - `SpecialRegisters`
- `_KPCR`
  - `CurrentThread`
  - `Prcb`

These objects are mapped into kernel virtual space **before the first kernel instruction executes**.

4.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

Transfer from Loader

1. `winload.efi` resolves the kernel entry RVA.
2. The loader sets:
    - `RSP = LoaderBlock->KernelStack`
    - `CR3 = Kernel page tables`
3. A near jump transfers execution to `nt!KiSystemStartup`.

First Instructions Inside `KiSystemStartup`

Architecturally, the first instructions perform:

1. Interrupt masking (`CLI`)
2. Stack frame normalization
3. KPCR base installation into `GS`
4. Invocation of `HalInitializeProcessor`

Transition into C Initialization

Once the assembly prologue completes:

- Control passes into `KiInitializeKernel`
- Then into `ExpInitializeExecutive`

From this point onward, the kernel operates under full C control.

4.4.5 Secure Kernel / VBS / HVCI Interaction

When VBS is enabled:

- `KiSystemStartup` executes as a **VTL0 guest entry point**.
- The Secure Kernel (VTL1) already controls:
  - MSR writes (via MSR bitmap)
  - IOMMU configuration
  - Kernel code page integrity
- Any attempt to modify `KiSystemStartup` code is blocked by HVCI.

Thus, the first kernel instructions already execute inside a **hypervisor-enforced trust boundary**.

4.4.6 Performance Implications

`KiSystemStartup` directly determines:

- Bootstrap processor initialization latency
- AP startup synchronization cost
- Initial timer source selection
- Hybrid P-core / E-core classification

Any misconfiguration here propagates permanently into:

- Scheduler tick precision
- Cross-core wake-up efficiency
- Early NUMA locality decisions

4.4.7 Real Practical Example (External Assembly Only)

Educational Reconstruction of a Kernel Startup Stub

This example models the **structural behavior** of `KiSystemStartup`. It is not a functional Windows kernel implementation.

ki_startup_stub.asm:

```
PUBLIC KiSystemStartupStub
.code

KiSystemStartupStub PROC
    cli                         ; Interrupts disabled at entry

    mov     rsp, rcx            ; RCX = KernelStack from Loader
    and     rsp, 0FFFFFFFFFFFFFF0h ; 16-byte alignment
    sub     rsp, 20h            ; 32-byte shadow space

    call    HalInitializeProcessorStub
    call    KiInitializeKernelStub

    hlt                         ; Must never return
KiSystemStartupStub ENDP

END
```

This mirrors:

- Stack establishment
- Shadow-space enforcement
- HAL first-call ordering

4.4.8 Kernel Debugging & Inspection

Break at Kernel Entry:

```
bp nt!KiSystemStartup
g
```

Inspect Stack at Entry:

```
r rsp
dq rsp
```

Inspect KPCR Installation:

```
rdmsr c0000101
```

4.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack targets:

- Entry-point patching
- Loader stack pointer manipulation
- Fake KPCR base injection

Windows 11 defenses:

- Secure Boot signature validation
- HVCI kernel code protection
- Hypervisor-enforced MSR controls

Modern exploitation at this stage requires **firmware or hypervisor compromise**.

4.4.10 Professional Kernel Engineering Notes

- No pageable code can execute here.
- No interrupt handlers are valid yet.
- No scheduler objects exist.
- A single invalid stack pointer causes silent triple fault.
- Kernel debugging requires:
  - Boot-time KD enabled
  - Network or serial transport

Part III

Kernel Address Space, Paging & Page Tables

Chapter 5

Virtual Address Space in Windows 11

5.1 Complete Memory Layout

5.1.1 Precise Windows 11–Specific Definition

The complete virtual memory layout in Windows 11 is a **canonical 64-bit, four-level paged address space** governed by the x86-64 long-mode paging architecture and enforced by the Windows Memory Manager under strict Secure Boot, VBS, and HVCI constraints. Windows 11 uses:

- 48-bit canonical virtual addressing
- 4-level paging: `PML4  PDPT  PD  PT`
- Split user/kernel virtual address ranges

All memory accesses in user mode and kernel mode are resolved exclusively through this virtual address layout. No physical addressing is allowed after long mode activation.

5.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The virtual address space layout defines:

- Isolation between user processes
- Isolation between user mode and kernel mode
- Placement of kernel code, data, stacks, and pools
- Secure Kernel (VTL1) hidden memory regions
- Device memory mappings via HAL

Every kernel subsystemscheduler, memory manager, I/O manager, object managerrelies on this fixed architectural partitioning to enforce security and stability.

5.1.3 Internal Kernel Data Structures (Real Structs & Fields)

The layout is governed and tracked internally using:

- `_EPROCESS`
  - `UserDirectoryTableBase`
  - `VadRoot`
- `_KPROCESS`
  - `DirectoryTableBase`
- `_MMVAD`
  - `StartingVpn`

  - `EndingVpn`
  - `VadFlags`

- `_MI_SYSTEM_INFORMATION`
  - `MaximumSystemCacheSize`
  - `PagedPoolMaximumPages`

These structures define which virtual ranges belong to user allocations, mapped images, stacks, heaps, system cache, and kernel pools.

5.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Address Translation Pipeline

For every memory access:

1. CPU uses `CR3` to locate the active `PML4`.
2. PML4 entry indexes into the `PDPT`.
3. PDPT entry indexes into the `PD`.
4. PD entry indexes into the `PT`.
5. PT entry resolves the final physical frame.

Kernel and User Mapping Rules

- User mode uses a per-process `CR3`.
- Kernel space is mapped identically in all processes.
- Secure Kernel memory is unmapped from VTL0 entirely.

Fault Handling Path

1. Page fault triggers vector #PF.
2. Control enters `nt!KiPageFault`.
3. Memory Manager resolves VAD and PTE state.
4. Page is either populated, denied, or the process is terminated.

5.1.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- VTL1 Secure Kernel owns a hidden address space.
- VTL0 kernel cannot map Secure Kernel pages.
- HVCI enforces kernel code as read-only and executable only.
- Hypervisor validates all second-level page table updates.

This prevents:

- Kernel code patching
- Page table poisoning
- DMA-based kernel memory attacks

5.1.6 Performance Implications

Virtual layout directly affects:

- TLB hit rates
- Page walk latency
- NUMA locality
- Kernel pool fragmentation

Inefficient layout results in:

- DPC latency spikes
- Scheduler delays
- Cache line thrashing in kernel pools

5.1.7 Real Practical Example (External Assembly Only)

Reading the Current Page Table Base (CR3)

read_cr3.asm:

```
PUBLIC ReadCr3
.code

ReadCr3 PROC
    mov     rax, cr3
    ret
ReadCr3 ENDP

END
```

This routine retrieves the active page table base used to resolve all virtual memory accesses.

5.1.8 Kernel Debugging & Inspection

Dump Current Page Table Base:

```
r cr3
```

Inspect Virtual Address Translation:

```
!pte fffff80000000000
```

Dump Process VAD Tree:

```
!vad
```

5.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack techniques:

- Kernel PTE overwrite
- VAD tree corruption
- Pool header overflow leading to arbitrary mapping

Windows 11 protections:

- HVCI executable-only kernel code
- Hypervisor-controlled second-level translation
- Secure Kernel isolation of credential memory

Successful exploitation now requires bypassing both the kernel and the hypervisor.

5.1.10 Professional Kernel Engineering Notes

- Kernel virtual addresses are identical across all processes.
- User virtual addresses are always per-process.
- Secure Kernel memory is permanently invisible to the normal kernel.
- Page table corruption results in immediate bug check.
- Virtual layout assumptions must never be hardcoded in drivers.

5.2 Separation Between User and Kernel Space

5.2.1 Precise Windows 11–Specific Definition

The separation between user space and kernel space in Windows 11 is a **hardware-enforced, page-tablemediated privilege isolation model** based on x86-64 long-mode paging, privilege rings, and supervisor/user access bits. User space operates at **CPL=3**, while kernel space operates at **CPL=0**. The CPU enforces access rules using:

- Page Table Entry **U/S** (User/Supervisor) bit
- **SMEP** (Supervisor Mode Execution Prevention)
- **SMAP** (Supervisor Mode Access Prevention)
- **NX/XD** (No-Execute) bit

Windows 11 extends this model with **hypervisor-enforced VTL separation**, where the Secure Kernel (VTL1) is fully isolated from the normal kernel (VTL0).

5.2.2 Exact Architectural Role Inside the Windows 11 Kernel

This separation guarantees:

- User-mode code cannot directly access kernel memory.
- Kernel-mode code cannot accidentally execute user-mode code.
- Kernel-mode reads from user pages require explicit validation.
- Secure Kernel memory cannot be accessed by the normal kernel at all.

The scheduler, memory manager, I/O manager, and object manager all rely on this separation to:

- Enforce process isolation
- Protect kernel execution integrity
- Prevent escalation via direct memory access

5.2.3 Internal Kernel Data Structures (Real Structs & Fields)

The separation is enforced and tracked through:

- `_EPROCESS`
  - `UserDirectoryTableBase`
  - `Flags2` (process mitigation flags)
- `_KPROCESS`
  - `DirectoryTableBase`

- `_MMVAD`
  - `StartingVpn`
  - `EndingVpn`
  - `VadFlags.Protection`
- `_MMPTE`
  - `u.Hard.Valid`
  - `u.Hard.Owner` (User/Supervisor)
  - `u.Hard.NoExecute`

These structures determine exactly which virtual pages are accessible from user mode and which are restricted to the kernel.

5.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

User-Mode Memory Access

1. User instruction issues a virtual memory load/store.
2. CPU walks the page tables using `CR3`.
3. If `PTE.U/S = 0` and CPL=3, a fault occurs.
4. Control transfers to `nt!KiPageFault`.
5. The Memory Manager validates the faulting address via the VAD tree.

Kernel-Mode Access to User Memory

1. Kernel code requests user buffer access.
2. Probing is performed via `ProbeForRead/ProbeForWrite`.
3. Access is performed under structured exception handling.
4. Page fault is serviced or converted into an access violation.

SMEP/SMAP Enforcement

- SMEP blocks kernel execution of user pages.
- SMAP blocks kernel data reads from user pages unless AC flag is set.

5.2.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- VTL1 memory is mapped in a separate set of second-level page tables.
- VTL0 kernel cannot map or access Secure Kernel pages.
- HVCI enforces:
  - Kernel code as read-only + executable
  - Driver images as immutable after verification

This creates a **three-layer isolation model**:

- User space (CPL=3)
- Normal kernel (CPL=0, VTL0)
- Secure Kernel (CPL=0, VTL1)

5.2.6 Performance Implications

This separation directly impacts:

- Context switch cost (user kernel transitions)
- TLB flushing behavior across address space changes
- System call entry/exit overhead
- SMEP/SMAP validation latency

Excessive user-kernel transitions increase:

- Instruction pipeline flushes
- Cache pollution
- TLB miss frequency

5.2.7 Real Practical Example (External Assembly Only)

Detecting Current Privilege Level via CS Register

read_cpl.asm:

```
PUBLIC ReadCpl
.code

ReadCpl PROC
    mov     ax, cs
    and     eax, 3          ; Extract CPL from CS[1:0]
    ret
```

```
ReadCpl ENDP

END
```

Interpretation:

- Return value = 0 Kernel Mode
- Return value = 3 User Mode

5.2.8 Kernel Debugging & Inspection

Inspect Current Privilege and Process Context:

```
r cs
!process 0 1
!thread
```

Inspect a Virtual Address Translation:

```
!pte <virtual_address>
```

Inspect User VAD Tree:

```
!vad
```

5.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack techniques:

- User-to-kernel pointer confusion
- Kernel stack overwrite via unchecked user buffers
- SMEP bypass via ROP into kernel-mapped user memory

- Page table permission flipping

Windows 11 defensive barriers:

- SMEP + SMAP always enabled on supported CPUs
- HVCI enforces immutable kernel code
- VTL1 isolates credential and security memory

Modern kernel exploitation now requires a **multi-stage chain crossing both the kernel and the hypervisor boundary**.

5.2.10 Professional Kernel Engineering Notes

- Kernel code must never trust user pointers without probing.
- User buffers must always be accessed under exception guards.
- Hardcoding kernel virtual addresses is invalid in Windows 11.
- Secure Kernel memory can never be directly inspected from VTL0.
- Incorrect SMEP/SMAP handling results in immediate bug checks.

5.3 The Impact of KASLR on Assembly

5.3.1 Precise Windows 11–Specific Definition

Kernel Address Space Layout Randomization (KASLR) in Windows 11 is a **mandatory, hypervisor-aware, boot-time virtual base randomization mechanism** applied to:

- `ntoskrnl.exe`

- `HAL.dll`
- Boot-time drivers
- Kernel pools

KASLR randomizes the **virtual base address** of all kernel images on every boot. The randomization is applied by `winload.efi` **before** control is transferred to `KiSystemStartup`. No fixed kernel base address exists in Windows 11.

5.3.2 Exact Architectural Role Inside the Windows 11 Kernel

KASLR fundamentally changes how the kernel executes by:

- Eliminating static kernel virtual addresses.
- Forcing all control transfers to be **RIP-relative**.
- Preventing absolute addressing in all kernel-mode Assembly.
- Breaking static exploit primitives based on fixed offsets.

The scheduler, memory manager, and interrupt dispatch all execute from **randomized virtual locations**, while maintaining stable **relative layout internally**.

5.3.3 Internal Kernel Data Structures (Real Structs & Fields)

KASLR state is recorded and consumed through:

- `_KLDR_DATA_TABLE_ENTRY`
  - `DllBase` (randomized)
  - `SizeOfImage`

  - `EntryPoint`

- `_LOADER_PARAMETER_BLOCK`

  - `LoadOrderListHead`
  - `KernelStack`

- `_MI_SYSTEM_INFORMATION`

  - `KernelVaStart`
  - `KernelVaEnd`

All kernel components resolve addresses dynamically using these tables.

5.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Boot-Time Randomization

1. `winload.efi` selects a randomized kernel virtual base.
2. PE relocations are applied to:
   - `ntoskrnl.exe`
   - `HAL.dll`
3. `_KLDR_DATA_TABLE_ENTRY::DllBase` is updated.
4. Kernel page tables are created with randomized mappings.

Runtime Symbol Resolution

- Absolute jumps are forbidden.
- All kernel code uses:
  - **RIP-relative addressing**
  - **Indirect call tables**

Fault Handling Under KASLR

- A stale absolute address causes immediate `PAGE_FAULT_IN_NONPAGED_AREA`.
- No fallback address resolution exists.

5.3.5 Secure Kernel / VBS / HVCI Interaction

Under VBS + HVCI:

- Kernel code pages are mapped:
  - Read-only
  - Executable-only
- Relocation data is discarded after verification.
- Hypervisor validates all executable mappings.

Even if KASLR is bypassed, **runtime patching of kernel code is blocked by HVCI**.

5.3.6 Performance Implications

KASLR introduces:

- Zero runtime performance penalty after boot.
- No impact on:
  - Cache locality
  - TLB hit rate
  - Instruction pipeline

The only cost is:

- Boot-time relocation processing.

5.3.7 Real Practical Example (External Assembly Only)

Incorrect Absolute Kernel Addressing (Will Fail)

bad_kaslr.asm:

```
PUBLIC BadCall
.code

BadCall PROC
    mov     rax, 0FFFFF80000001000h ; INVALID: static kernel address
    call    rax
    ret
BadCall ENDP

END
```

Result: Immediate bug check due to invalid executable mapping.

Correct RIP-Relative Kernel Call (KASLR-Safe)

good_kaslr.asm:

```
EXTERN TargetFunction:PROC
PUBLIC SafeCall
.code

SafeCall PROC
    lea     rax, TargetFunction[rip]
    call    rax
    ret
SafeCall ENDP

END
```

This is the **only valid form of kernel control transfer under KASLR**.

5.3.8 Kernel Debugging & Inspection

Reveal Kernel Base Address:

```
lm m nt
```

Inspect KASLR Relocation:

```
!lmi nt
```

Verify Executable Permissions:

```
!pte nt!KiSystemStartup
```

5.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Pre-Windows 10 attack primitives:

- Static kernel gadget jumping
- Hardcoded exploit offsets

Windows 11 enforcement:

- Mandatory KASLR at every boot.
- HVCI executable-only enforcement.
- Hypervisor validation of all kernel mappings.

Modern kernel exploitation requires:

- KASLR base leak
- SMEP/SMAP bypass
- HVCI bypass

5.3.10 Professional Kernel Engineering Notes

- Absolute addressing is forbidden in Windows 11 kernel Assembly.
- All kernel symbols must be resolved dynamically.
- RIP-relative addressing is mandatory.
- Statically linked kernel offsets are non-functional.
- Any driver violating KASLR rules will crash the system instantly.

Chapter 6

x86-64 Page Tables as Used by Windows 11

6.1 PML4 / PDPT / PD / PT

6.1.1 Precise Windows 11–Specific Definition

Windows 11 on x86-64 uses the **four-level long-mode paging hierarchy** defined by AMD64 and Intel 64 architectures:

- **PML4** Page Map Level 4
- **PDPT** Page Directory Pointer Table
- **PD** Page Directory
- **PT** Page Table

Each virtual address is translated using a **48-bit canonical address** decomposed into:

- Bits 47:39 PML4 index

- Bits 38:30 PDPT index
- Bits 29:21 PD index
- Bits 20:12 PT index
- Bits 11:0 Page offset

Windows 11 enforces this hierarchy for **all** memory accesses in:

- User Mode (CPL=3)
- Kernel Mode (CPL=0, VTL0)
- Secure Kernel (CPL=0, VTL1 via second-level translation)

No alternate paging formats are supported in Windows 11.

6.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The x86-64 page table hierarchy is the **primary hardware mechanism** that the Windows 11 Memory Manager uses to enforce:

- Process isolation
- User / kernel separation
- Kernel code integrity
- Secure Kernel (VTL1) isolation
- DMA and IOMMU confinement

All kernel subsystems depend on it:

- Scheduler stack and thread object mapping
- I/O Manager MDL-backed buffer mapping
- Object Manager object directory mappings
- Executive system cache and section objects

The page tables are therefore the **hard enforcement layer** beneath all Windows 11 security mechanisms.

6.1.3 Internal Kernel Data Structures (Real Structs & Fields)

Windows 11 tracks and manipulates page tables using:

- `_KPROCESS`
  - `DirectoryTableBase` (CR3 value)
- `_EPROCESS`
  - `UserDirectoryTableBase`
- `_MMPTE`
  - `u.Hard.Valid`
  - `u.Hard.Write`
  - `u.Hard.Owner` (User/Supervisor)
  - `u.Hard.NoExecute`
- `_MI_SYSTEM_INFORMATION`
  - `KernelVaStart`

- `KernelVaEnd`

Each valid virtual page used by Windows 11 corresponds to a fully populated `_MMPTE` chain across the PML4 PDPT PD PT hierarchy.

6.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Hardware Translation Pipeline

For every virtual memory reference:

1. CPU reads `CR3` PML4 base.
2. Index into PML4 using VA[47:39].
3. Index into PDPT using VA[38:30].
4. Index into PD using VA[29:21].
5. Index into PT using VA[20:12].
6. Combine PFN with VA[11:0] final physical address.

Windows-Specific Mapping Rules

- Kernel space is **globally mapped** in all processes.
- User space is **per-process** and isolated via unique CR3.
- Secure Kernel uses **second-level translation** enforced by Hyper-V.

Page Fault Resolution Path

1. Hardware raises `#PF`.
2. Control enters `nt!KiPageFault`.
3. Memory Manager checks:
    - VAD permissions
    - PTE state
    - Backing store (pagefile or image)
4. Page is mapped, denied, or the process is terminated.

6.1.5 Secure Kernel / VBS / HVCI Interaction

When VBS and HVCI are enabled:

- Kernel page tables are **shadowed** by the hypervisor.
- VTL0 mappings are validated against VTL1 policy.
- All executable kernel pages must be:
    - Signed
    - Immutable
    - NX-enforced for data

This blocks:

- Page table permission flipping
- Code cave injection
- DMA-based kernel memory modification

6.1.6 Performance Implications

The four-level hierarchy directly impacts:

- Page walk latency on TLB miss
- TLB pressure under heavy context switching
- NUMA remote memory access
- Kernel pool access locality

Large page usage (2 MB via PD, 1 GB via PDPT) reduces:

- TLB miss frequency
- Page walk overhead

but increases internal fragmentation.

6.1.7 Real Practical Example (External Assembly Only)

Reading the Active PML4 Base (CR3)

read_cr3.asm:

```
PUBLIC ReadCr3
.code

ReadCr3 PROC
    mov     rax, cr3
    ret
ReadCr3 ENDP

END
```

Extracting Page Table Indices from a Virtual Address

va_decode.asm:

```
PUBLIC DecodeVa
.code

; RCX = Virtual Address
; Returns:
;  RAX = PML4 Index
;  RBX = PDPT Index
;  RDX = PD Index
;  RSI = PT Index
DecodeVa PROC
    mov     rax, rcx
    shr     rax, 39
    and     rax, 1FFh

    mov     rbx, rcx
    shr     rbx, 30
    and     rbx, 1FFh

    mov     rdx, rcx
    shr     rdx, 21
    and     rdx, 1FFh

    mov     rsi, rcx
    shr     rsi, 12
    and     rsi, 1FFh
    ret
```

```
DecodeVa ENDP

END
```

This mirrors the exact hardware decomposition used by Windows 11.

6.1.8 Kernel Debugging & Inspection

Inspect Active Page Tables:

```
r cr3
!pte <virtual_address>
```

Dump Kernel PML4 Region:

```
!pte fffff80000000000
```

Inspect Process Address Space:

```
!process 0 1
!vad
```

6.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack primitives:

- Kernel PTE overwrite
- Fake large-page remapping
- VAD corruption leading to RWX mappings

Windows 11 hardening:

- HVCI prevents executable PTE modification
- Hypervisor validates second-level translation
- Secure Kernel isolates credential memory

Modern page-table attacks require a **hypervisor escape**.

6.1.10 Professional Kernel Engineering Notes

- CR3 is always the authoritative PML4 base.
- Kernel mappings are identical across all processes.
- Never assume fixed kernel virtual addresses.
- Page tables must never be modified directly by drivers.
- Large-page mappings impact both performance and exploitability.

6.2 Flags from the Perspective of Security and Performance

6.2.1 Precise Windows 11–Specific Definition

In Windows 11 on x86-64, every paging structure entry (**PML4E, PDPTE, PDE, PTE**) is a 64-bit value composed of:

- **Physical Frame Number (PFN)**
- **Control and protection flags**

These flags are the **direct hardware enforcement layer** for:

- Memory access permissions
- Execution permission
- Caching behavior
- Privilege separation
- Hypervisor validation under VBS

Windows 11 strictly programs these flags through the Memory Manager and the Hyper-V hypervisor when VBS/HVCI is enabled.

6.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Paging flags serve as the **final authority** that determines:

- Whether a page is executable or not
- Whether user-mode may access kernel memory
- Whether memory is cached, write-combined, or uncached
- Whether writes are permitted
- Whether the page is global across address spaces

All security technologies in Windows 11including:

- HVCI
- Kernel-mode CFG
- Secure Kernel (VTL1)

ultimately resolve to **controlled manipulation and verification of page-table flags**.

6.2.3 Internal Kernel Data Structures (Real Structs & Fields)

Windows 11 represents paging entries using:

- `_MMPTE`
- `_MMPTE_HARDWARE`

Key hardware fields:

- `Valid`
- `Write`
- `Owner`
- `WriteThrough`
- `CacheDisable`
- `Accessed`
- `Dirty`
- `LargePage`
- `Global`
- `NoExecute`

These bits are mapped directly onto the hardware PTE format used by the CPU.

6.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Hardware Permission Check Sequence

For every memory reference:

1. CPU resolves the PTE through page-walk.
2. The **User/Supervisor** bit is compared to CPL.
3. The **Write** bit is checked for store operations.
4. The **NX** bit is checked on instruction fetch.
5. Cache type is determined by **PCD/PWT**.

If any violation occurs, a hardware exception is raised:

- `#PF` Page Fault

Windows 11 Policy Enforcement

Windows 11 programs flags based on:

- VAD permissions
- Section object attributes
- Kernel pool type
- Secure kernel policy

All executable kernel pages are forced into:

- **NX cleared**
- **Write disabled**

and verified by the hypervisor when HVCI is active.

6.2.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Page tables in VTL0 are **shadowed**
- Execute permissions are validated by VTL1
- Writable + executable mappings are **blocked in hardware**

This eliminates:

- Kernel shellcode injection
- Self-modifying kernel code
- Direct PTE-based permission flipping

6.2.6 Performance Implications

Global Bit (`G`)

- Prevents TLB flush on context switches
- Used for kernel mappings
- Increases TLB efficiency

NX Bit

- Introduces an additional permission check
- Negligible performance impact
- Massive security gain

Cache Disable (PCD) / Write Through (PWT)

- Used for MMIO regions
- Prevents speculative caching
- Significantly slower than cached RAM

Large Page Flag

- Reduces TLB pressure
- Improves kernel hot-path performance
- Increases internal fragmentation risk

6.2.7 Real Practical Example (External Assembly Only)

Reading a Raw PTE via Physical Mapping

read_pte.asm:

```
PUBLIC ReadPteEntry
.code

; RCX = Virtual address
; RDX = Base of PTE self-map
; Returns: RAX = Raw PTE value
ReadPteEntry PROC
    mov     rax, rcx
    shr     rax, 9
    and     rax, 7FFFFFFF8h
    add     rax, rdx
```

```
    mov     rax, [rax]
    ret
ReadPteEntry ENDP

END
```

Decoding Security-Critical Flags

pte_decode.asm:

```
PUBLIC DecodePteFlags
.code

; RCX = Raw PTE
; Returns:
;  RAX = Valid
;  RBX = Writable
;  RDX = User
;  RSI = NX
DecodePteFlags PROC

    mov     rax, rcx
    and     rax, 1

    mov     rbx, rcx
    shr     rbx, 1
    and     rbx, 1

    mov     rdx, rcx
    shr     rdx, 2
```

```
    and     rdx, 1

    mov     rsi, rcx
    shr     rsi, 63
    and     rsi, 1

    ret
DecodePteFlags ENDP

END
```

6.2.8 Kernel Debugging & Inspection

Inspect PTE Flags:

```
!pte fffff80001234000
```

Decode Permission Bits:

```
r cr3
dt nt!_MMPTE
```

Check NX Enforcement:

```
!process 0 1
!vad
```

6.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack usage:

- RWX kernel pages via PTE overwrite

- Clearing NX on data pages
- Forcing User/Supervisor bit inversion

Windows 11 hardening:

- Hypervisor-enforced PTE permissions
- Secure Kernel validation of executable pages
- IOMMU validation of DMA memory

Modern exploitation requires:

- A hypervisor escape
- Or a Secure Kernel vulnerability

6.2.10 Professional Kernel Engineering Notes

- Never assume writability implies executability.
- Global bit misuse leaks TLB state across processes.
- Cache-disabled pages severely impact performance.
- Large pages improve speed but complicate security auditing.
- All modern kernel exploits eventually target paging flags.

6.3 Manual Address Translation Using WinDbg

6.3.1 Precise Windows 11–Specific Definition

Manual address translation in Windows 11 is the **explicit reconstruction of the hardware page-walk algorithm** performed by the CPU for a given virtual address, using:

- The current **CR3** value
- The four-level paging hierarchy:
  - PML4
  - PDPT
  - PDT
  - PT

This process is performed manually inside **WinDbg/KD** to verify:

- Exact physical mapping
- Permission flags
- NX enforcement
- User/Supervisor boundaries
- VBS/HVCI shadow mappings

Manual translation is the **ground truth technique** used in kernel debugging, exploit research, DMA forensics, and Secure Kernel validation.

6.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Windows 11 relies on:

- Hardware page walking in the MMU
- Software page management by the Memory Manager
- Hypervisor shadow translation under VBS

Manual translation allows engineers to directly validate:

- Whether the **Memory Manager programmed the hardware correctly**
- Whether **Hyper-V modified the effective permissions**
- Whether a page fault originated from:
  - Missing entry
  - NX violation
  - Supervisor access violation
  - Write-protection

This is the only method that fully bypasses all higher-level abstractions.

6.3.3 Internal Kernel Data Structures (Real Structs & Fields)

Windows 11 uses the following core internal structures during translation:

- `_MMPTE`
- `_MMPTE_HARDWARE`

- `_EPROCESS.DirectoryTableBase`
- `_KPROCESS.DirectoryTableBase`

Key hardware-mapped fields of `_MMPTE_HARDWARE`:

- `Valid`
- `Write`
- `Owner`
- `WriteThrough`
- `CacheDisable`
- `Accessed`
- `Dirty`
- `LargePage`
- `Global`
- `NoExecute`

These fields map **bit-for-bit to the actual hardware PTE format**.

6.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Hardware Page Walk Algorithm

For a canonical 48-bit virtual address:

1. Read **CR3** physical base of PML4

2. Extract PML4 index (bits 4739)
3. Read PML4E
4. Extract PDPT index (bits 3830)
5. Read PDPTE
6. Extract PDT index (bits 2921)
7. Read PDE
8. Extract PT index (bits 2012)
9. Read PTE
10. Combine PTE PFN with page offset (bits 110)

Windows 11 Hypervisor Interception

With VBS enabled:

- CR3 used by the CPU may be **shadowed**
- Effective permission may differ from raw PTE
- NX and RWX enforcement is validated in VTL1

6.3.5 Secure Kernel / VBS / HVCI Interaction

Under HVCI:

- Writable + Executable mappings are blocked
- Kernel code PTEs are verified against signed images

- Page permission changes are validated in Secure Kernel

Manual translation detects:

- Hidden hypervisor-enforced NX pages
- Shadow permissions not visible to VTL0
- Rogue DMA memory mappings

6.3.6 Performance Implications

- Each manual page walk requires **4 physical memory reads**
- TLB normally hides this cost
- Page table corruption causes **catastrophic performance collapse**
- Large pages reduce translation overhead significantly

6.3.7 Real Practical Example (WinDbg Manual Walk)

Obtain the Active CR3

```
r cr3
```

Output gives the physical base of the PML4 table.

Select a Virtual Address

Example kernel virtual address:

```
fffff80001234567
```

Extract indices:

- PML4 index: bits 47–39
- PDPT index: bits 38–30
- PDT index: bits 29–21
- PT index: bits 20–12

Read the PML4 Entry

```
!dq <PML4_base + PML4_index * 8> L1
```

Walk PDPT, PDT, and PT

```
!dq <PDPT_base + PDPT_index * 8> L1
!dq <PDT_base  + PDT_index  * 8> L1
!dq <PT_base   + PT_index   * 8> L1
```

Reconstruct the Final Physical Address

```
PhysicalAddress = (PTE_PFN << 12) | Offset
```

This result must match hardware translation.

6.3.8 Kernel Debugging & Inspection Commands

Automatic vs Manual Comparison:

```
!pte fffff80001234567
```

Inspect Raw PTE:

```
dt nt!_MMPTE <address>
```

Check Current Process Address Space:

```
!process 0 1
r cr3
```

Validate VAD Permissions:

```
!vad
```

6.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack techniques:

- PTE overwrites to enable RWX kernel pages
- Clearing NX on shellcode buffers
- User/Supervisor bit inversion
- Fake PDE injection via DMA

Windows 11 defenses:

- Hypervisor-verified execute permissions
- Secure Kernel enforcement of PTE writes
- IOMMU-based DMA page validation

Modern successful exploitation now requires:

- Hypervisor escape
- Or Secure Kernel compromise

6.3.10 Professional Kernel Engineering Notes

- Manual translation is the only way to **prove real permissions**.
- `!pte` can lie under VBS shadowing.
- CR3 changes across process switches invalidate previous walks.
- Large pages change the walk depth and offset calculation.
- All kernel memory corruption debugging eventually ends at a PTE.

Part IV

Kernel Memory Manager (C & Assembly View)

Chapter 7

Memory Pools & Allocation

7.1 `ExAllocatePool2`

7.1.1 Windows 11Specific Definition (Engineering-Level)

`ExAllocatePool2` is the primary pool allocation routine introduced for modern Windows 10+ and used by Windows 11 kernel-mode components and drivers to allocate memory from kernel pools using a unified flag-based interface.

```
#include <wdm.h>

PVOID
ExAllocatePool2(
    _In_ ULONG      Flags,        // POOL_FLAG_*
    _In_ SIZE_T     NumberOfBytes,
    _In_ ULONG      Tag           // 4-byte pool tag
    );
```

Key properties:

- **Environment**: Kernel mode only, callable by ntoskrnl.exe, HAL.dll, and kernel-mode drivers.
- **IRQL constraints**:
  - Nonpaged allocations (`POOL_FLAG_NON_PAGED`, etc.): up to `DISPATCH_LEVEL`.
  - Paged allocations (`POOL_FLAG_PAGED`): up to `APC_LEVEL`.
- **Pool selection**: Encoded into `Flags` using `POOL_FLAG_*` bits instead of separate `POOL_TYPE`.
- **Failure semantics**: Returns `NULL` on failure by default. With `POOL_FLAG_RAISE_ON_FAILURE`, a bug check is raised instead of returning `NULL`.
- **Tagging**: The `Tag` argument is mandatory. It is used by pool diagnostics, leak detection, and security mechanisms such as pool cookies.

Typical flag categories:

- **Pool type**:
  - `POOL_FLAG_PAGED`
  - `POOL_FLAG_NON_PAGED`
  - `POOL_FLAG_NON_PAGED_EXECUTE` (for executable kernel memory)
- **Allocation behavior and attributes** (subset):
  - `POOL_FLAG_CACHE_ALIGNED`
  - `POOL_FLAG_UNINITIALIZED` (skip zeroing)
  - `POOL_FLAG_RAISE_ON_FAILURE`

  - `POOL_FLAG_SESSION` (session space)
  - `POOL_FLAG_SPECIAL_POOL`, `POOL_FLAG_VERIFIER_SPECIAL_POOL` (when configured)

7.1.2 Architectural Role Inside the Windows 11 Kernel

From the kernel architecture perspective, `ExAllocatePool2` is the canonical entry point into the *general-purpose kernel heap*. Its role:

- Provides a uniform API for:
  - ntoskrnl.exe core subsystems (Memory Manager, Object Manager, I/O Manager).
  - Device drivers (KMDF / WDM).
  - Support components (e.g., file systems, filter drivers).
- Abstracts:
  - Underlying page allocation from the Memory Manager (`Mm`).
  - Per-pool structures (lookaside lists, pool descriptors).
  - Security instrumentation (pool tagging, cookies, special pool, verifier).
- Integrates with:
  - Kernel address space layout (Section 1 of this Part).
  - KASLR (addresses are randomized, but allocation API remains stable).
  - Performance-related mechanisms such as per-processor pools and NUMA-aware allocation.

For an assembly or C engineer, `ExAllocatePool2` is the *lowest practical abstraction* to obtain kernel memory while still being compatible with all Windows 11 security and verification features.

7.1.3 Internal Kernel Data Structures (Real Structures & Fields)

The implementation of `ExAllocatePool2` is internal to ntoskrnl.exe and heavily version-dependent. It relies on several core structures that can be inspected via symbols in WinDbg but are not intended to be used by drivers directly.
A simplified view of a pool header (for conceptual understanding) looks like:

```
nt!_POOL_HEADER
   +0x000 PreviousSize     : UShort
   +0x002 PoolIndex        : UChar
   +0x003 BlockSize        : UChar
   +0x004 PoolType         : ULong
   +0x008 PoolTag          : ULong
   +0x00c PoolCookie       : ULong
   // Additional version-specific fields...
```

Important points:

- **Pool header** precedes the pointer returned to the caller. Corruption of this header (buffer overflows, use-after-free) is a classic exploitation vector.
- **Pool tag** mirrors the `Tag` argument supplied to `ExAllocatePool2`.
- **Pool cookie** is a per-boot randomized value used by the kernel to detect tampering with pool headers (mitigation against classic pool exploitation techniques).
- **Pool descriptors** and internal lists:
  - Non-paged and paged pools are managed by internal descriptors (e.g., `_POOL_DESCRIPTOR`) that track free lists, usage, and fragmentation.
  - There are per-node / per-processor structures for performance and scalability on NUMA systems.

As a rule:

- Kernel-mode drivers **must not** rely on the layout of these structures.
- When analysis is required (reverse engineering, debugging, forensics), these structures should be examined via public symbols and WinDbg (`dt`, `!pool`, etc.) on the exact Windows 11 build under inspection.

7.1.4 Execution Flow (Step-by-Step, C & Assembly View)

Conceptual execution of `ExAllocatePool2` within ntoskrnl.exe can be broken down into the following steps.

High-Level C-Style Flow

```
PVOID
ExAllocatePool2(
    _In_ ULONG      Flags,
    _In_ SIZE_T     NumberOfBytes,
    _In_ ULONG      Tag
    )
{
    // 1. Basic validation
    if (NumberOfBytes == 0) {
        // Either return NULL or raise, depending on Flags and build
        ↪  configuration.
    }

    // 2. IRQL checks based on Flags (paged vs nonpaged)
    KIRQL irql = KeGetCurrentIrql();
    if (Flags & POOL_FLAG_PAGED) {
        if (irql > APC_LEVEL) {
            // bugcheck or fail fast
```

```
        }
    } else {
        if (irql > DISPATCH_LEVEL) {
            // bugcheck or fail fast
        }
    }

    // 3. Determine effective pool and allocation attributes
    POOL_FLAGS effectiveFlags = (POOL_FLAGS)Flags;

    // 4. Fast-path allocation from existing free lists/SLISTs
    PVOID buffer = ExAllocatePoolInternal(effectiveFlags, NumberOfBytes,
    ↪  Tag);
    if (buffer == NULL) {
        if (Flags & POOL_FLAG_RAISE_ON_FAILURE) {
            // raise bugcheck or structured exception, depending on build
        }
        return NULL;
    }

    // 5. Initialization if required
    if (!(Flags & POOL_FLAG_UNINITIALIZED)) {
        RtlZeroMemory(buffer, NumberOfBytes);
    }

    return buffer;
}
```

This is *conceptual* pseudocode; the real implementation is more complex, NUMA-aware, and tightly integrated with the Memory Manager and pool verifier.

Calling Sequence (Assembly Perspective)

A typical kernel-mode call-site compiled by MSVC will use the Windows x64 ABI:

- `RCX` = `Flags`
- `RDX` = `NumberOfBytes`
- `R8` = `Tag`
- 32-byte shadow space reserved on the stack.
- Stack aligned to 16 bytes at call site.

A minimal external assembly helper (MASM-style, placed in a separate `.asm` file) that calls `ExAllocatePool2`:

```
; File: PoolHelper.asm
; Assembled with ml64 and linked into a kernel driver project
; Windows x64 ABI, no inline assembly used.

EXTERN  ExAllocatePool2:PROC

.code

; PVOID AllocateNonPagedPool(SIZE_T NumberOfBytes, ULONG Tag);
AllocateNonPagedPool PROC
    ; RCX = NumberOfBytes, RDX = Tag (as passed from C stub)
    ; We need to rearrange to:
    ;   RCX = Flags
    ;   RDX = NumberOfBytes
    ;   R8  = Tag

    sub     rsp, 32                 ; shadow space, keep 16-byte alignment
```

```
    mov     r8,  rdx                ; R8 = Tag
    mov     rdx, rcx                ; RDX = NumberOfBytes
    mov     ecx, POOL_FLAG_NON_PAGED

    call    ExAllocatePool2

    add     rsp, 32
    ret
AllocateNonPagedPool ENDP

END
```

A C stub in the same driver might declare:

```
PVOID
AllocateNonPagedPool(
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG  Tag
    );
```

MSVC for x64 does not support inline assembly in kernel drivers; all such low-level call-sequence control must be placed in external `.asm` files, as shown above.

7.1.5 Secure Kernel / VBS / HVCI Interaction

`ExAllocatePool2` itself allocates from the standard kernel virtual address space (VTL0). Its interaction with modern security mechanisms is indirect but important:

- **Virtualization-Based Security (VBS)**:
  - When VBS is enabled, some kernel memory may be mapped or mediated through the Hyper-V hypervisor and the Secure Kernel (VTL1).

  - `ExAllocatePool2` hides these details; from the perspective of a driver, it still returns a VTL0-mapped kernel virtual address.

- **HVCI / Code Integrity**:

  - Executable nonpaged pool (`POOL_FLAG_NON_PAGED_EXECUTE`) is highly constrained.
  - Drivers must not allocate executable memory and write code into it; such patterns are incompatible with HVCI and may be blocked or cause bug checks.

- **Secure pools**:

  - Secure pools are created via `ExCreatePool` and consumed via `ExAllocatePool3` + extended parameters.
  - `ExAllocatePool2` is generally used for non-secure (standard) pool allocations, but freed buffers can be handled universally via `ExFreePool2`.

From a low-level engineers point of view, the critical rule is: *do not attempt to bypass these abstraction layers*. All VBS/HVCI-related restrictions are enforced in the allocator, not at the call site.

7.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Pool allocations have direct performance impact across several subsystems:

- **Cache locality**:

  - Frequent small allocations scattered across the address space can increase cache and TLB pressure.

  - For hot-path data structures, use `POOL_FLAG_CACHE_ALIGNED` when appropriate, and prefer object reuse (lookaside lists, caches) rather than continuous allocate/free patterns.

- **NUMA-awareness**:

  - Internally, the kernel tries to allocate memory from the NUMA node local to the current processor.
  - Excessive cross-node usage or migrating threads across NUMA nodes can degrade performance due to remote memory access.

- **IRQL and scheduler interaction**:

  - Allocations at `DISPATCH_LEVEL` are constrained to nonpaged pool and cannot incur blocking I/O.
  - Large allocations at high IRQL may cause contention on global pool locks or increase fragmentation.

- **Fragmentation**:

  - Random-sized allocations with long lifetimes fragment the pool and may increase memory overhead.
  - For heavily used subsystems, using fixed-size slabs or lookaside lists (e.g., `NPAGED_LOOKASIDE_LIST`, `ExInitializeLookasideListEx`) can significantly reduce fragmentation and allocation cost.

7.1.7 Real Practical Example (Kernel C Driver Code)

Below is a minimal WDM-style driver using `ExAllocatePool2` and `ExFreePool2` correctly, including tagging and IRQL-safe usage.

```
#include <ntddk.h>

#define POOL_TAG_DEMO 'moDP' // 'PDom' in little-endian

DRIVER_UNLOAD DemoDriverUnload;

VOID
DemoDriverUnload(
    _In_ PDRIVER_OBJECT DriverObject
    )
{
    UNREFERENCED_PARAMETER(DriverObject);
    DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL,
               "DemoDriver: Unload\n");
}

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT  DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = DemoDriverUnload;

    SIZE_T      size  = 4096;
    ULONG       flags = POOL_FLAG_NON_PAGED; // Valid at DISPATCH_LEVEL and
    ↪ below
    PVOID       buffer;

    buffer = ExAllocatePool2(flags, size, POOL_TAG_DEMO);
    if (buffer == NULL) {
```

```
        DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_ERROR_LEVEL,
                   "DemoDriver: ExAllocatePool2 failed\n");
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    // Initialize the buffer explicitly if POOL_FLAG_UNINITIALIZED is used.
    RtlZeroMemory(buffer, size);

    // Use the buffer for some work at PASSIVE_LEVEL or DISPATCH_LEVEL
    // ...

    // Free with ExFreePool2 (non-secure allocation: no extended
    ↪  parameters)
    ExFreePool2(buffer,
                POOL_TAG_DEMO,
                NULL,    // ExtendedParameters
                0);      // ExtendedParametersCount

    return STATUS_SUCCESS;
}
```

Notes:

- The `POOL_TAG_DEMO` constant is a four-character literal; it appears byte-swapped in WinDbg (little-endian representation).
- Since this example uses non-secure pool, `ExtendedParameters` is `NULL` and `ExtendedParametersCount` is zero.
- All calls are at `PASSIVE_LEVEL`, so both paged and nonpaged pool would be legal; nonpaged is chosen here to demonstrate the most general IRQL-safe choice.

7.1.8 Kernel Debugging & Inspection (WinDbg / KD Commands)

To analyze `ExAllocatePool2`-based allocations in a live or crash-dump debugging session:

- **Inspect overall pool usage**:

```
!vm
!memusage
!poolused 2
```

- **Search for allocations by tag**:

```
!poolused 2 POOL_TAG_DEMO
!pool -n moDP
```

  - `!pool -n` searches by tag.
  - Remember that tags are byte-swapped in display.

- **Inspect a specific allocation header** (given an address returned by `ExAllocatePool2`):

```
!pool <address>
dt nt!_POOL_HEADER (<address> - 0x10)
```

- **Verify pool integrity and special pool**:

```
!verifier 3
!verifier 80
```

- **Track allocations/free from a specific driver**:

```
!poolused 2 <first-char-of-tag>*
lm m <yourdriver>    ; verify module base and timestamp
```

For deeper analysis, you can disassemble the call site in your driver:

```
uf <yourdriver>!DriverEntry
```

and the internals of `nt!ExAllocatePool2` (for reverse engineering / research purposes only):

```
uf nt!ExAllocatePool2
```

7.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Pool allocations have historically been a primary exploitation surface:

- **Buffer overflows**:
  - Writing past the end of a buffer returned by `ExAllocatePool2` can corrupt adjacent pool headers or objects.
  - Modern Windows versions employ pool cookies, safe unlinking, and other mitigations, but logic errors remain exploitable.
- **Use-after-free**:
  - Freeing a buffer via `ExFreePool2` and later dereferencing stale pointers can lead to arbitrary code execution or corruption of kernel structures.
- **Type confusion / tag misuse**:
  - Re-using the same pool tag for different allocation types hampers forensics and can hide dangerous patterns.
  - Some mitigations check tag consistency; freeing with a mismatched tag may bugcheck.
- **Executable pool misuse**:

  - Allocating with `POOL_FLAG_NON_PAGED_EXECUTE` and writing code into the buffer is incompatible with modern code integrity and exposes a strong exploit primitive.
  - Such use should be avoided unless you are implementing very low-level components under strict review.

- **Boundary with Secure Kernel / VBS**:
  - Standard `ExAllocatePool2` does not directly manipulate VTL1 memory, but compromised kernel code can still attack hypercall interfaces and secure pools if other protections are bypassed.

7.1.10 Professional Kernel Engineering Notes (Pitfalls & Rules)

Practical rules for using `ExAllocatePool2` correctly in Windows 11 kernel-mode code:

- **Always use a unique, meaningful tag**:
  - Choose tags per subsystem or per major data type.
  - Maintain an internal tag registry for your driver or product.
- **Respect IRQL and pool type**:
  - Never allocate paged memory at `DISPATCH_LEVEL` or above.
  - Avoid large allocations at high IRQL; refactor to allocate at `PASSIVE_LEVEL` and reuse buffers.
- **Initialize memory explicitly**:
  - Unless you have a very specific reason, *do not* use `POOL_FLAG_UNINITIALIZED`.

  - Zero or initialize fields explicitly to avoid uninitialized data leaks.

- **Pair allocation and free correctly**:

  - Prefer `ExFreePool2` with the same tag and, if applicable, appropriate extended parameters for secure pools.
  - Ensure every allocation path has a corresponding free path (including failure branches and early returns).

- **Avoid executable pool allocations**:

  - If you think you need `POOL_FLAG_NON_PAGED_EXECUTE`, re-evaluate your design under the Windows 11 HVCI model.
  - JIT-like behavior in the kernel is strongly discouraged.

- **Consider object-specific allocators for hot paths**:

  - For frequently allocated fixed-size objects, use lookaside lists or custom slab allocators built on top of `ExAllocatePool2`.
  - This reduces fragmentation and improves cache behavior.

- **Instrument with Driver Verifier and Special Pool**:

  - Use `POOL_FLAG_SPECIAL_POOL` / `POOL_FLAG_VERIFIER_SPECIAL_POOL` where appropriate to catch overruns and use-after-free in development.
  - Enable Driver Verifier for your driver with pool tracking to detect leaks and misuse early.

- **Never depend on deterministic addresses**:

  - KASLR and internal allocation strategies mean pool addresses are not stable across boots or systems.
  - All low-level analysis should be driven by symbols and WinDbg, not hard-coded addresses or assumptions.

This section establishes `ExAllocatePool2` as the central primitive for kernel heap allocations in Windows 11 from a C and assembly engineers point of view. All subsequent memory manager topics in this Part assume a precise understanding of these semantics, constraints, and security properties.

7.2 The Difference Between Paged Memory and NonPaged Memory

7.2.1 Windows 11Specific Definition (Engineering-Level)

In Windows 11, **paged memory** and **nonpaged memory** represent two fundamentally different classes of kernel pool allocations with strict execution, IRQL, and residency guarantees.

- **Paged memory**:
  - Kernel virtual memory that **may be paged out** to disk.
  - Backed by the system paging file.
  - Accessible only at IRQL $\leq$ `APC_LEVEL`.
- **Nonpaged memory**:
  - Kernel virtual memory that is **always resident in physical RAM**.
  - Never subject to paging I/O.

  - Accessible at IRQL $\leq$ `DISPATCH_LEVEL`.

In Windows 11, both types are allocated via `ExAllocatePool2` using explicit flag selection:

- `POOL_FLAG_PAGED`
- `POOL_FLAG_NON_PAGED`
- `POOL_FLAG_NON_PAGED_EXECUTE`

7.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Paged and nonpaged memory participate in different execution domains of the kernel:

- **Paged memory**:
  - Used by:
    - Object Manager metadata
    - Registry caches
    - File system metadata
    - Driver code and data executing only at `PASSIVE_LEVEL`
  - Fully integrated with the Memory Manager paging subsystem.
  - Subject to demand paging, trimming, and working set pressure.
- **Nonpaged memory**:
  - Used by:
    - Interrupt handlers (ISR/DPC)
    - Scheduler data structures

        * I/O request paths
        * Spinlock-protected objects
    - Must remain resident to guarantee interrupt-time correctness.
    - Forms part of the always-mapped kernel working set.

From an architectural standpoint:

- Paged memory trades **latency predictability** for **capacity**.
- Nonpaged memory trades **capacity** for **deterministic execution**.

7.2.3 Internal Kernel Data Structures (Real Structures & Fields)

Paged and nonpaged pools are managed using different internal descriptors inside `ntoskrnl.exe`.
Key structures visible through symbols include:

```
nt!_POOL_DESCRIPTOR
nt!_POOL_HEADER
nt!_MI_SYSTEM_PTE_TYPE
nt!_MMPFN
```

Important architectural distinctions:

- **Paged pool**:
    - Backed by pageable system PTEs.
    - PFN entries transition between resident and paged-out states.
    - Tracked by working set management logic.
- **Nonpaged pool**:

- Backed by permanently resident system PTEs.
- PFN entries are locked and never transitioned to paging file.
- Charged against the systemwide nonpaged pool limit.

Every allocation is preceded by a `_POOL_HEADER`, regardless of paged or nonpaged status. Pool cookies and tags apply to both.

7.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Paged Memory Allocation Flow

1. Caller executes at `PASSIVE_LEVEL` or `APC_LEVEL`.
2. `ExAllocatePool2(POOL_FLAG_PAGED, Size, Tag)` is invoked.
3. Kernel verifies IRQL.
4. Allocation is satisfied from paged pool free lists or via new system PTEs.
5. Pages may initially exist in a demand-zero state.
6. On first access, page fault may occur and be serviced by the Memory Manager.

Nonpaged Memory Allocation Flow

1. Caller may execute at `PASSIVE_LEVEL` through `DISPATCH_LEVEL`.
2. `ExAllocatePool2(POOL_FLAG_NON_PAGED, Size, Tag)` is invoked.
3. Kernel verifies IRQL and dispatch constraints.
4. Physical pages are reserved and permanently mapped.
5. Returned buffer is immediately usable without any page fault risk.

Assembly-Level Call Discipline

Both paged and nonpaged allocations use the same ABI:

- `RCX` = Flags
- `RDX` = Size
- `R8` = Tag

External assembly usage example (nonpaged):

```
EXTERN  ExAllocatePool2:PROC

AllocateNonPaged PROC
    sub     rsp, 32
    mov     r8,  'tseT'              ; Tag
    mov     rdx, rcx                 ; Size
    mov     ecx, POOL_FLAG_NON_PAGED
    call    ExAllocatePool2
    add     rsp, 32
    ret
AllocateNonPaged ENDP
```

Paged allocations follow the same calling pattern with `POOL_FLAG_PAGED`.

7.2.5 Secure Kernel / VBS / HVCI Interaction

- **Paged memory**:
  - Fully visible to VTL0.
  - Can be mapped, paged, and validated by hypervisor-backed page tables.
  - Page faults may involve Secure Kernel mediation under VBS.

- **Nonpaged memory**:
  - Permanently mapped into kernel virtual address space.
  - Used by hypervisor-facing kernel code paths.
  - Must obey HVCI restrictions on executable permissions.
- **Executable nonpaged memory**:
  - Strongly restricted under HVCI.
  - Writable + executable nonpaged memory is systematically blocked.

7.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- **Paged memory**:
  - Subject to:
    - Page faults
    - Working set trimming
    - Disk I/O latency
  - Lower physical memory pressure.
  - Higher worst-case latency.
- **Nonpaged memory**:
  - Zero page-fault risk.
  - Higher TLB and cache residency.
  - Consumes scarce nonpaged pool quota.
  - Excessive use degrades overall system performance and increases bug check risk.

- **NUMA considerations**:
  - Both paged and nonpaged allocations are NUMA-aware.
  - Nonpaged remote-node memory access has permanent latency cost.

7.2.7 Real Practical Example (C / Kernel Mode)

```
#define TAG_PAGED  'gaPT'
#define TAG_NPAGED 'gaPN'

PVOID pagedBuf;
PVOID nonpagedBuf;

pagedBuf = ExAllocatePool2(
    POOL_FLAG_PAGED,
    4096,
    TAG_PAGED
);

nonpagedBuf = ExAllocatePool2(
    POOL_FLAG_NON_PAGED,
    4096,
    TAG_NPAGED
);
```

Execution rules:

- `pagedBuf` must never be touched above `APC_LEVEL`.
- `nonpagedBuf` is safe up to `DISPATCH_LEVEL`.

7.2.8 Kernel Debugging & Inspection (WinDbg / KD)

```
!vm
!memusage
!poolused 2
!pool -n gaPT
!pool -n gaPN
```

Examine specific allocation type:

```
!pool <address>
dt nt!_POOL_HEADER (<address> - 0x10)
```

IRQL verification:

```
!irql
```

7.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

- **Paged memory vulnerabilities**:
  - Time-of-check/time-of-use after paging
  - Page-in race conditions
  - Use-after-free after trimming
- **Nonpaged memory vulnerabilities**:
  - Deterministic exploitation primitives
  - Use-after-free persistence
  - Pool header corruption leading to privilege escalation
- **Security boundary rule**:
  - Trust boundary between pageable data and interrupt-time execution is absolute.

7.2.10 Professional Kernel Engineering Notes (Real-World Rules)

- Never touch paged memory at `DISPATCH_LEVEL` or above.
- Prefer paged pool for:
  - Configuration data
  - Large buffers
  - Infrequently accessed structures
- Reserve nonpaged pool strictly for:
  - ISR/DPC paths
  - Spinlock-protected objects
  - Scheduler and I/O hot paths
- Never overuse nonpaged pool to "fix" IRQL mistakes.
- Always document IRQL assumptions near every allocation.
- Always pair pool type with execution context.

7.3 Common Driver Memory Allocation Mistakes

7.3.1 Windows 11Specific Definition (Engineering-Level)

In Windows 11, **driver memory allocation mistakes** refer to violations of strict kernel execution, IRQL, paging, synchronization, and security rules when using kernel pool allocators such as `ExAllocatePool2`. Unlike user-mode memory errors, these mistakes operate directly within the trusted kernel execution domain and therefore lead to:

- Immediate system crashes (bug checks)
- Silent memory corruption
- Irrecoverable security boundary violations
- Deterministic local privilege escalation

These mistakes are magnified under Windows 11 due to:

- HVCI-enforced W^X memory rules
- Secure Kernel separation (VTL0/VTL1)
- Stricter pool protections and tagging enforcement

7.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Kernel pool allocations participate directly in:

- Dispatcher objects
- I/O Manager request paths
- Scheduler queues
- Object Manager namespaces
- Memory Manager page tracking

Incorrect allocation behavior compromises:

- IRQL correctness
- Page residency guarantees

- Pool header integrity
- Hypervisor enforced page attributes

From an architectural standpoint, a pool misuse is not a local driver errorit is a cross-subsystem kernel integrity violation.

7.3.3 Internal Kernel Data Structures (Real Structures & Fields)

The following real kernel structures are directly impacted by allocation mistakes:

```
nt!_POOL_HEADER
nt!_POOL_DESCRIPTOR
nt!_MMPFN
nt!_MI_SYSTEM_PTE_TYPE
nt!_EX_PUSH_LOCK
```

Corruption of `_POOL_HEADER` fields such as:

- `BlockSize`
- `PoolType`
- `PoolTag`

directly destabilizes pool coalescing logic and can redirect kernel execution.

7.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Correct Allocation Path

1. Driver executes at valid IRQL.
2. `ExAllocatePool2` validates flags.

3. Pool descriptor resolves appropriate free list.

4. `_POOL_HEADER` is initialized.

5. Buffer is returned to caller.

Faulty Allocation Path (Typical Failure Case)

1. Driver executes at `DISPATCH_LEVEL`.

2. Requests `POOL_FLAG_PAGED`.

3. Kernel detects pageable access at elevated IRQL.

4. Immediate bug check: `IRQL_NOT_LESS_OR_EQUAL`.

Assembly Call Misuse Example

```
; INVALID: paged allocation at DISPATCH_LEVEL

EXTERN  ExAllocatePool2:PROC

FaultyAlloc PROC
    sub     rsp, 32
    mov     r8,  'DAFB'          ; Bad tag
    mov     rdx, 4096
    mov     ecx, POOL_FLAG_PAGED
    call    ExAllocatePool2      ; Bug check risk
    add     rsp, 32
    ret
FaultyAlloc ENDP
```

7.3.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces the following security constraints:

- Writable + executable nonpaged memory is blocked under HVCI.
- Pool headers are validated under hypervisor-backed shadow structures.
- Invalid executable mappings trigger `SECURE_KERNEL_ERROR`.
- Nonpaged execution is confined to verified image sections.

Common violations include:

- Executing dynamically allocated nonpaged memory
- Modifying pool headers to escalate permissions
- Reusing freed executable pool buffers

7.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Misuse of kernel memory allocations causes:

- Excessive nonpaged pool pressure
- Cross-node NUMA latency amplification
- TLB churn due to fragmented pool mappings
- Scheduler stalls on spinlock-protected pool lists

Overuse of nonpaged pool directly reduces available kernel working set and degrades I/O throughput deterministically.

7.3.7 Real Practical Mistake Examples (C / Kernel Mode)

Mistake 1: Paged Memory at DISPATCH_LEVEL

```
PVOID buf = ExAllocatePool2(
    POOL_FLAG_PAGED,
    1024,
    'BAD1'
);

KeAcquireSpinLock(&Lock, &oldIrql);
RtlZeroMemory(buf, 1024);    // INVALID at DISPATCH_LEVEL
KeReleaseSpinLock(&Lock, oldIrql);
```

Mistake 2: Double Free

```
PVOID buf = ExAllocatePool2(
    POOL_FLAG_NON_PAGED,
    512,
    'BAD2'
);

ExFreePool(buf);
ExFreePool(buf);   // Pool header corruption
```

Mistake 3: Executing from Non-Executable Pool

```
PVOID code = ExAllocatePool2(
    POOL_FLAG_NON_PAGED,
    4096,
    'BAD3'
);

((void(*)())code)();   // HVCI violation
```

7.3.8 Kernel Debugging & Inspection (WinDbg / KD)

Detect pool misuse:

```
!analyze -v
!poolfind BAD1
!poolfind BAD2
!poolfind BAD3
```

Verify pool headers:

```
!pool <address>
dt nt!_POOL_HEADER (<address> - 0x10)
```

Detect IRQL violations:

```
!irql
kp
```

Detect nonpaged pool exhaustion:

```
!vm 1
!memusage
```

7.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common vulnerability classes caused by allocation mistakes:

- Pool header overwrite exploitation
- Use-after-free on nonpaged pool
- Double free leading to list poisoning
- Executable pool reuse under disabled HVCI

Security boundary violations occur when:

- User data is stored in nonpaged kernel buffers without validation
- Pageable memory is accessed in interrupt context
- Pool headers are attacker-controlled

These flaws are primary primitives in modern Windows kernel exploits.

7.3.10 Professional Kernel Engineering Notes (Real-World Rules)

- Never allocate paged pool above `APC_LEVEL`.
- Never execute memory returned by `ExAllocatePool2`.
- Always pair allocation flags with strict IRQL documentation.
- Always zero sensitive nonpaged memory before free.
- Never reuse freed pool pointers.
- Use distinct pool tags per subsystem.
- Treat every pool allocation as a security boundary.

Chapter 8

VAD, Protection & Memory Security

8.1 Inside _MMVAD

8.1.1 Precise Windows 11Specific Definition (Engineering-Level)

The _MMVAD (Memory Manager Virtual Address Descriptor) is the core kernel data structure used by the Windows 11 Memory Manager to describe every contiguous region of virtual address space belonging to a process. Each _MMVAD instance represents:

- A uniquely defined virtual address interval
- Its access protection attributes
- Backing store type (private, mapped file, image)
- Commitment state
- Security-relevant execution attributes

Windows 11 relies on the _MMVAD tree as the **authoritative access-control database** for process virtual memory. Page tables alone are not trusted without VAD validation.

8.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Architecturally, `_MMVAD` resides at the boundary between:

- User-mode virtual memory requests
- Kernel-mode memory enforcement
- Secure Kernel memory validation

Every operation involving:

- `NtAllocateVirtualMemory`
- `NtMapViewOfSection`
- `NtProtectVirtualMemory`
- Image loading

must traverse the **VAD AVL tree** attached to the owning `_EPROCESS` object. The VAD system is therefore:

- A scheduling-critical structure
- A paging-critical structure
- A security-critical structure

8.1.3 Internal Kernel Data Structures (Real Structures & Fields)

The Windows 11 VAD system is implemented using these core structures:

```
nt!_MMVAD
nt!_MMVAD_SHORT
nt!_MMVAD_LONG
nt!_MM_AVL_TABLE
nt!_EPROCESS
```

Key fields inside `_MMVAD` include:

```
VadNode         -> AVL tree linkage
StartingVpn    -> Virtual page start
EndingVpn      -> Virtual page end
Flags          -> Protection, commit, private, image
Flags2         -> Long/Short format, secure flags
Subsection     -> Backing file mapping (if any)
```

Each `_EPROCESS` contains:

```
VadRoot        -> Root of the VAD AVL tree
```

8.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Virtual Allocation Path

1. User-mode issues `NtAllocateVirtualMemory`.
2. Transition via `syscall`.
3. Kernel resolves target `_EPROCESS`.
4. `MiAllocateVad` constructs a new `_MMVAD`.

5. VAD is inserted into the process AVL tree.

6. Page tables are updated lazily on access (demand paging).

Assembly-Level System Call Transition

```
; User-mode syscall entry (Windows x64 ABI)

mov     r10, rcx
mov     eax, NtAllocateVirtualMemory
syscall
```

The kernel dispatcher then transitions into:

```
nt!NtAllocateVirtualMemory
nt!MiAllocateVad
nt!MiInsertVadCharges
```

8.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 Secure Kernel enforcement:

- Every executable VAD is validated against HVCI rules.
- Writable + executable VAD ranges are blocked.
- VAD protection flags must align with Secure Kernel policy.
- Image-backed VADs are verified via code integrity before insertion.

Violations trigger:

- `SECURE_KERNEL_ERROR`
- `ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY`

VADs therefore form the first line of defense against:

- JIT spraying
- Shellcode injection
- User-mode ROP staging

8.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

The VAD AVL tree directly affects:

- Page fault resolution latency
- TLB miss recovery cost
- Working set trimming precision
- NUMA node allocation locality

Heavy VAD fragmentation causes:

- AVL rebalancing overhead
- Cache-line thrashing on VAD lookups
- Increased scheduler pressure during memory faults

Large image-backed VADs improve:

- TLB efficiency
- Sequential access locality
- Page-fault batching

8.1.7 Real Practical Example (Kernel Observation)

User-Mode Allocation Creating a VAD

```
PVOID Base = NULL;
SIZE_T Size = 0x20000;

NtAllocateVirtualMemory(
    NtCurrentProcess(),
    &Base,
    0,
    &Size,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);
```

This causes:

- Creation of an executable VAD
- Secure Kernel validation
- W^X enforcement

Kernel Inspection of VAD

```
dt nt!_MMVAD <vad_address>
```

8.1.8 Kernel Debugging & Inspection (WinDbg / KD)

Dump process VAD tree:

```
!process 0 1
!vad
```

Inspect a single VAD:

```
dt nt!_MMVAD <address>
```

Validate protection:

```
!address <virtual_address>
```

Detect VAD inconsistencies:

```
!analyze -v
```

8.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common exploitation primitives involving VADs:

- VAD permission flipping (RW → RX)
- VAD unlink attacks
- VAD subtree poisoning
- Fake image-backed VAD injection

Security boundary violations occur when:

- User-mode memory is remapped as executable
- Kernel trusts user-supplied subsection metadata
- HVCI enforcement is bypassed via malformed VAD flags

VAD manipulation remains the dominant vector for modern kernel-mode memory exploits.

8.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never assume page table permissions reflect execution permissionsVADs override.
- Never trust executable memory without Secure Kernel validation.
- Never modify VAD structures outside `Mi*` routines.
- All image mappings must originate from verified sections.
- VAD corruption always results in delayed system collapse, not immediate failure.
- VAD invariants are enforced continuously by Memory Manager worker threads.

8.2 DEP / NX / Copy-on-Write

8.2.1 Precise Windows 11Specific Definition (Engineering-Level)

Data Execution Prevention (DEP), **No-Execute (NX)**, and **Copy-on-Write (CoW)** are three tightly integrated memory protection mechanisms enforced by the Windows 11 kernel Memory Manager at both the **VAD layer** and the **page-table layer**.

- **DEP** is the policy enforcement layer.
- **NX** is the hardware execution prevention bit implemented in x86-64 page tables.
- **Copy-on-Write** is the lazy physical duplication mechanism used for private writable mappings.

Windows 11 enforces these mechanisms simultaneously through coordinated validation of:

- `_MMVAD` protection flags

- Page table NX bit
- Secure Kernel executable memory rules

8.2.2 Exact Architectural Role Inside the Windows 11 Kernel

These mechanisms operate across three architectural enforcement domains:

- **VAD Layer**: Declares memory intent and access policy.
- **Paging Hardware**: Enforces execution and write permissions.
- **Secure Kernel (VTL1)**: Validates executable memory eligibility.

DEP/NX/CoW are involved in:

- Image section mapping
- Private heap allocations
- Stack growth
- Shared memory views
- Process cloning (fork-style behavior inside subsystem initialization)

8.2.3 Internal Kernel Data Structures (Real Structures & Fields)

Protection enforcement spans these structures:

```
nt!_MMVAD
nt!_MMVAD_FLAGS
nt!_MMPTE
nt!_MMPTE_HARDWARE
nt!_EPROCESS
nt!_SECTION_OBJECT
```

Critical fields:

```
_MM_VAD_FLAGS.Protection
_MM_VAD_FLAGS.NoChange
_MMPTE_HARDWARE.NX
_MMPTE_HARDWARE.Write
```

Copy-on-Write is tracked implicitly through:

```
Prototype PTEs
Private Demand-Zero PTEs
```

8.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

DEP/NX Enforcement Path

1. Process issues executable memory request.
2. `NtAllocateVirtualMemory` transitions to kernel.
3. VAD is created with Protection field.
4. Secure Kernel validates executable policy.
5. NX bit is programmed into PTE.
6. First instruction fetch triggers hardware validation.

Copy-on-Write Fault Path

1. Two processes map same physical page as private.
2. Write instruction triggers page fault.
3. Memory Manager allocates new physical page.

4. Original data is copied.
5. New PTE is updated as writable.

Hardware Enforcement (Assembly-Level View)

```
mov     rax, [rip]     ; instruction fetch
; CPU checks:
; - PTE.Present
; - PTE.User/Supervisor
; - PTE.NX
; NX violation triggers:
; #PF (Execute Access Violation)
```

8.2.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces DEP through:

- HVCI validation before any executable page is committed
- Blocking W+X memory mappings
- Forcing image-backed execution for kernel-mode code

Secure Kernel prevents:

- User-mode writable executable memory
- Runtime shellcode staging
- JIT-based exploit chains

Copy-on-Write is also validated under VBS to prevent:

- Cross-process code page corruption
- Shared memory privilege escalation

8.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- NX-enabled pages reduce speculative execution attack surfaces.
- CoW increases page-fault overhead during write-heavy workloads.
- TLB shootdowns occur during CoW page split.
- NUMA locality is preserved only after CoW duplication.

DEP/NX has near-zero steady-state overhead but increases:

- Fault handling latency for misconfigured memory.
- JIT runtime execution cost.

8.2.7 Real Practical Example (C / Assembly)

User-Mode NX-Protected Allocation

```
PVOID Base = NULL;
SIZE_T Size = 0x1000;

NtAllocateVirtualMemory(
    NtCurrentProcess(),
    &Base,
    0,
    &Size,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE
);
```

Execution from this region triggers:

- NX fault
- `STATUS_ACCESS_VIOLATION`

Copy-on-Write Trigger Example

```
char *Shared = MapViewOfFile(...);
Shared[0] = 0x41;   // Triggers CoW fault
```

Kernel Observation of NX PTE

```
!pte <virtual_address>
```

8.2.8 Kernel Debugging & Inspection (WinDbg / KD)

Inspect VAD protection:

```
!vad
dt nt!_MMVAD <address>
```

Inspect NX bit:

```
!pte <address>
```

Monitor CoW faults:

```
!vm 4
!memusage
```

Detect DEP violations:

```
!analyze -v
```

8.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common exploit targets:

- Forcing PAGE_EXECUTE_READWRITE VADs
- Removing NX via PTE corruption
- CoW race manipulation during page split
- Prototype PTE poisoning

Security boundaries:

- User-mode cannot override NX
- Kernel executable memory must be image-backed
- CoW pages must remain private after split

8.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never request PAGE_EXECUTE_READWRITE in modern Windows.
- Always assume CoW faults under heavy write load.
- NX is enforced by both CPU and Secure Kernel.
- Copy-on-Write is invisible at API level but dominant in fault behavior.
- DEP bypass always implies VAD or PTE corruption.
- Secure Kernel enforces execute policy before PTE programming.

8.3 How These Protections Are Practically Bypassed

8.3.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **DEP**, **NX**, and **Copy-on-Write (CoW)** form a layered execution-prevention and memory-integrity enforcement system spanning:

- The **VAD layer** (logical policy)
- The **PTE layer** (hardware enforcement)
- The **Secure Kernel (VTL1)** layer (hypervisor-validated execution rules)

A **practical bypass** in Windows 11 is therefore *never a single flip of a bit*, but a **multi-layer violation** involving corruption or desynchronization between:

- `_MMVAD` protection metadata
- Hardware PTE permissions (especially NX)
- Secure Kernel execution validation state

8.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Bypass activity targets the following architectural choke points:

- VAD creation and protection updates
- Prototype PTE resolution
- Demand-zero and private page faults
- Secure Kernel executable memory validation callbacks

All three protection mechanisms must be compromised for reliable execution:

$$\text{VAD} \rightarrow \text{PTE} \rightarrow \text{Secure Kernel}$$

8.3.3 Internal Kernel Data Structures (Real Structures & Fields)

The following kernel structures are consistently present in real-world bypass chains:

```
nt!_MMVAD
nt!_MMVAD_SHORT
nt!_MMVAD_FLAGS
nt!_MMPTE
nt!_MMPTE_HARDWARE
nt!_MI_VAD_EVENT_BLOCK
nt!_EPROCESS
```

Critical security-relevant fields:

```
_MM_VAD_FLAGS.Protection
_MM_VAD_FLAGS.NoChange
_MM_VAD_FLAGS.PrivateMemory
_MMPTE_HARDWARE.NX
_MMPTE_HARDWARE.Write
_MMPTE_HARDWARE.Owner
```

8.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Legitimate DEP/NX Path

1. Process requests virtual memory.
2. Memory Manager creates a VAD.
3. VAD flags determine executable intent.
4. Secure Kernel validates execution eligibility.
5. NX bit is set or cleared in final PTE.
6. CPU enforces execution at instruction fetch.

Generalized Bypass Flow (Abstract)

1. A kernel write primitive modifies VAD protection.
2. PTE is rebuilt with executable permission.
3. Secure Kernel execution validation is desynchronized.
4. Instruction fetch succeeds from formerly non-executable memory.

Hardware Enforcement (Assembly Perspective)

```
; Instruction fetch sequence
mov     rax, [rip]

; CPU checks:
; - Present
; - Privilege level
; - NX bit

; NX violation triggers:
; #PF (Instruction Fetch Fault)
```

Execution occurs only if **all three layers agree**.

8.3.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 eliminates entire historical bypass classes by enforcing:

- **Executable memory must be image-backed**
- **Writable + executable (W+X) mappings are forbidden**
- **Kernel code must pass VTL1 signature validation**

Even if:

- VAD flags are corrupted
- PTE NX bit is cleared

the Secure Kernel will still block execution if the page is not backed by a validated image section.

8.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Bypass activity introduces measurable performance anomalies:

- Repeated TLB invalidations due to forced PTE rebuilds
- Unexpected CoW page splits under write-triggered execution
- Cache line invalidation storms during prototype PTE poisoning
- Elevated page fault rate under masked execution probing

These side effects are frequently used by kernel security telemetry.

8.3.7 Real Practical Example (C / Assembly Defensive Analysis)

Detecting NX Violations (User-Mode)

```
__try {
    ((void(*)())Buffer)();
} __except(EXCEPTION_EXECUTE_HANDLER) {
    // STATUS_ACCESS_VIOLATION under NX enforcement
}
```

Detecting CoW Split Behavior

```
volatile char *p = SharedPage;
p[0] = 0x41;    // Forces CoW page duplication
```

External Assembly Observation of NX (No Inline ASM)

```
; External object file used only for observation
mov rax, [rcx]      ; Triggers NX fault if rcx maps NX page
ret
```

8.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD)

Inspect VAD and protection:

```
!vad
dt nt!_MMVAD <vad_address>
```

Inspect NX and write permissions:

```
!pte <virtual_address>
```

Track copy-on-write splits:

```
!vm 4
!memusage
```

Detect execution violations:

```
!analyze -v
```

8.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack vectors (now largely mitigated in Windows 11):

- VAD protection overwrites
- PTE NX clearing
- Prototype PTE poisoning
- Shared memory execution pivots

Hard boundaries in Windows 11:

- Secure Kernel prevents unsigned kernel execution
- HVCI blocks runtime kernel code mutation
- User-mode cannot influence final PTE execute state
- W+X kernel mappings are denied

Any modern bypass must therefore violate **VTL0 + VTL1 simultaneously**.

8.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- DEP bypass always implies corruption across **at least two layers**.
- NX alone is insufficient as a security control in Windows 11.
- Copy-on-Write is a frequent forensic indicator of execution staging.
- Secure Kernel execution validation cannot be bypassed from VTL0 without hypervisor compromise.

- Legitimate kernel drivers must assume DEP, NX, and CoW are always active.
- Any observed W+X mapping in kernel space is a critical security failure.

Part V

Processes, Threads & Context Switching

Chapter 9

Process Model for Low-Level Engineers

9.1 _EPROCESS Analysis

9.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, `_EPROCESS` is the definitive kernel-resident executive process object representing a live process instance. It is allocated from nonpaged pool during process creation and persists for the full lifetime of the process until final object dereference. `_EPROCESS` is not a scheduling structure. It is a **security, address-space, resource ownership, and lifetime authority object** that:

- Owns the user-mode virtual address space
- Owns the handle table
- Owns the security token reference
- Anchors all threads (`_ETHREAD`)

- Anchors the VAD tree
- Anchors page tables and working sets

In Windows 11, _EPROCESS is tightly coupled with VBS and Secure Kernel validation through indirect trust chains over its address space and image mappings.

9.1.2 Exact Architectural Role Inside the Windows 11 Kernel

_EPROCESS is the architectural nucleus of:

- Virtual memory management (VAD tree, page directory ownership)
- Security context (token, protection level, mitigation flags)
- Object lifetime control (process object reference counting)
- Kernel scheduling containment (thread ownership, CPU sets)
- Kernel debugging context binding

Subsystem dependency graph:

$$\text{Scheduler} \rightarrow \text{\_ETHREAD} \rightarrow \text{\_EPROCESS} \rightarrow \text{Memory Manager / Object Manager / Security}$$

9.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Windows 11 uses a large _EPROCESS structure with version-dependent layout. The following fields are architecturally stable and universally present:

```
nt!_EPROCESS
  +0x000 Pcb                    : _KPROCESS
  +0x028 ProcessLock            : _EX_PUSH_LOCK
```

```
+0x030 UniqueProcessId       : Ptr64 Void
+0x038 ActiveProcessLinks    : _LIST_ENTRY
+0x080 Token                 : _EX_FAST_REF
+0x0c0 VadRoot               : Ptr64 Void
+0x3c0 ImageFileName         : [15] UCHAR
+0x420 ProtectedProcess      : UCHAR
+0x440 ObjectTable           : Ptr64 _HANDLE_TABLE
+0x570 SectionObject         : Ptr64 Void
+0x580 WorkingSetWatch       : Ptr64 Void
+0x590 Vm                    : _MMSUPPORT_FULL
```

Critical embedded substructures:

- `_KPROCESS` scheduling and CPU affinity
- `_MMSUPPORT_FULL` working set and page aging
- VAD tree root (`VadRoot`)
- Handle table (`ObjectTable`)

9.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Process Creation Path

1. `NtCreateUserProcess` issued from user mode
2. `PspAllocateProcess` allocates `_EPROCESS`
3. VAD root initialized
4. Token attached
5. Handle table created

6. Image section mapped
7. Process inserted into active list

Context Switch Resolution Path

1. Scheduler selects an `_ETHREAD`
2. `_KTHREAD.ApcState.Process` resolves owning `_EPROCESS`
3. CR3 loaded from `_KPROCESS.DirectoryTableBase`
4. Address space instantly switches

CR3 Load at Instruction Level

```
mov     rax, [KPROCESS.DirectoryTableBase]
mov     cr3, rax
```

This instruction alone enforces the entire process address space switch.

9.1.5 Secure Kernel / VBS / HVCI Interaction

In Windows 11 with VBS enabled:

- The image sections referenced by `_EPROCESS.SectionObject` are validated by Secure Kernel.
- Token integrity levels are enforced via VTL1-protected policy.
- `ProtectedProcess` and `ProtectedProcessLight` restrict kernel tampering.

Even kernel-mode drivers cannot arbitrarily remap executable pages into protected `_EPROCESS` contexts when HVCI is active.

9.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

_EPROCESS-related performance costs include:

- Full TLB flush on each CR3 switch
- NUMA node locality enforced via _KPROCESS
- Working set trimming governed per _EPROCESS
- CPU set constraints influence migration latency

Process-heavy workloads suffer more from address-space switching than thread-heavy workloads.

9.1.7 REAL Practical Example (C / Assembly)

Accessing _EPROCESS in a Kernel Driver

```
PEPROCESS proc = PsGetCurrentProcess();

HANDLE pid = PsGetProcessId(proc);

PUCHAR name = PsGetProcessImageFileName(proc);
```

Walking the Active Process List (Kernel)

```
PLIST_ENTRY head = &PsActiveProcessHead;
PLIST_ENTRY entry = head->Flink;

while (entry != head) {
    PEPROCESS p = CONTAINING_RECORD(entry, EPROCESS, ActiveProcessLinks);
    entry = entry->Flink;
}
```

External Assembly Observation Stub (No Inline ASM)

```
global ReadCr3
ReadCr3:
    mov     rax, cr3
    ret
```

9.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump the current process:

```
!process 0 1
```

Inspect a specific _EPROCESS:

```
dt nt!_EPROCESS <address>
```

Display address space:

```
!vad
```

Dump working set:

```
!vm
```

Inspect token:

```
!token <TokenAddress>
```

9.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically targeted _EPROCESS attack surfaces:

- Token stealing via `Token` field overwrite

- VAD injection for shellcode staging
- Handle table substitution

Windows 11 hardened boundaries:

- Token integrity enforced via Secure Kernel
- VAD execution controlled by NX + HVCI
- Kernel pointer poisoning mitigations
- Randomized `_EPROCESS` layout per build

Direct `_EPROCESS` corruption is now a high-detection attack.

9.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never cache `_EPROCESS` pointers across asynchronous driver contexts.
- Always reference-count process objects when storing pointers.
- Never assume fixed offsets inside `_EPROCESS`.
- Do not infer security state from user-mode PEB; trust only `_EPROCESS`.
- CR3 manipulation outside the scheduler is forbidden.
- Protected processes are not debuggable with conventional kernel tools.

9.2 Tokens

9.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, a **security token** is a kernel-managed executive object that represents the complete security context of a process or thread. The token defines:

- User identity (SID)
- Group memberships
- Privileges
- Integrity level
- AppContainer and sandbox state
- Capability SIDs

Each process owns a `_TOKEN` referenced by the `_EPROCESS.Token` fast-reference field. Threads may temporarily override it using impersonation tokens stored in `_ETHREAD`.

9.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Tokens enforce all access control decisions inside the kernel:

- Object Manager access checks
- File system permissions
- Registry security
- IPC authorization

- ALPC security enforcement
- Protected Process Light (PPL) isolation

Every security boundary in Windows 11 ultimately resolves to a token comparison executed inside:

`SeAccessCheck`

The token is therefore the **final authority** over privilege, not the user-mode identity.

9.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Windows 11 uses the executive structure `_TOKEN`. The following fields are architecturally stable:

```
nt!_TOKEN
  +0x000 TokenSource         : _TOKEN_SOURCE
  +0x010 TokenId             : _LUID
  +0x018 AuthenticationId    : _LUID
  +0x020 ParentTokenId       : _LUID
  +0x030 ExpirationTime      : _LARGE_INTEGER
  +0x040 UserAndGroups       : Ptr64 _SID_AND_ATTRIBUTES
  +0x048 RestrictedSids      : Ptr64 _SID_AND_ATTRIBUTES
  +0x058 Privileges          : Ptr64 _LUID_AND_ATTRIBUTES
  +0x070 IntegrityLevelSid   : Ptr64 Void
  +0x090 TokenType           : _TOKEN_TYPE
  +0x0a0 ImpersonationLevel  : _SECURITY_IMPERSONATION_LEVEL
  +0x0b0 MandatoryPolicy     : ULONG
  +0x0c0 Capabilities        : Ptr64 Void
```

Fast reference in `_EPROCESS`:

```
nt!_EPROCESS
  +0x080 Token : _EX_FAST_REF
```

9.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Token Resolution Path During Access Check

1. Handle request enters Object Manager
2. `ObReferenceObjectByHandle` resolves object
3. `SeAccessCheck` invoked
4. Token retrieved from `_EPROCESS.Token`
5. SID and privilege arrays evaluated
6. Access granted or denied

Token Extraction at Instruction Level

```
mov     rax, [rcx + EPROCESS.Token]   ; EX_FAST_REF
and     rax, 0FFFFFFFFFFFFFFF0h       ; remove ref count bits
```

9.2.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Token privilege elevation paths are validated in VTL1
- PPL tokens are cryptographically isolated
- Kernel token manipulation requires Secure Kernel compliance
- LSA tokens are protected by Credential Guard

HVCI prevents unsigned kernel code from tampering with token memory.

9.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Token-related overhead appears in:

- ALPC message validation
- File system I/O authorization
- Registry access filtering

Each access check involves multiple SID comparisons and privilege scans, stressing:

- L1 data cache
- Branch prediction
- NUMA-local security descriptor access

9.2.7 REAL Practical Example (C / C++ / Assembly)

Kernel Access to Current Process Token

```
PEPROCESS proc = PsGetCurrentProcess();
PACCESS_TOKEN token = PsReferencePrimaryToken(proc);

/* Use token */

PsDereferencePrimaryToken(token);
```

Detecting SYSTEM Token in Kernel

```
if (SeTokenIsAdmin(token)) {
    // Elevated context
}
```

External Assembly Token Read Stub

```
global GetProcessToken
GetProcessToken:
    mov     rax, [rcx + EPROCESS.Token]
    and     rax, 0FFFFFFFFFFFFFFF0h
    ret
```

9.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump current process token:

```
!process -1 1
```

Dump token:

```
!token <TokenAddress>
```

Inspect privileges:

```
dt nt!_TOKEN <TokenAddress>
```

Check integrity level:

```
!sid <IntegritySid>
```

9.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Classical attack targets:

- Token privilege bit flipping
- SYSTEM token stealing
- Impersonation token abuse

Windows 11 mitigations:

- PPL token isolation
- Secure Kernel validation of high-privilege tokens
- Kernel pointer encryption
- Restricted token propagation rules

Modern token attacks require combined memory corruption and Secure Kernel bypass.

9.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never cache token pointers without reference tracking.
- Never modify token privileges directly.
- Always use `PsReferencePrimaryToken`.
- Token offsets are build-dependent.
- PPL tokens cannot be impersonated.
- Impersonation level must be validated before use.

9.3 Handles

9.3.1 Precise Windows 11Specific Definition (Engineering-Level)

A **handle** in Windows 11 is a user-mode reference to a kernel-mode executive object managed by the **Object Manager**. A handle is *not* a pointer. It is an index into a **per-process handle table** that maps to an **object header** in kernel memory.
Handles are used to reference:

- Processes
- Threads
- Files
- Sections
- Tokens
- Events, mutexes, semaphores
- ALPC ports

The handle itself has meaning **only inside the owning process context**. The kernel resolves it through the process `_HANDLE_TABLE`.

9.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Handles form the **primary indirection layer** between:

- User-mode access requests
- Kernel-mode object lifetime control

Their architectural roles include:

- Enforcing access masks on kernel objects
- Supporting reference counting and object lifetime
- Isolating user-mode from direct kernel pointers
- Enabling secure object sharing across processes

All user-mode system calls that operate on kernel objects (`NtReadFile`, `NtOpenProcess`, `NtDuplicateObject`, etc.) begin with:

`Handle → Handle Table → Object Header → Executive Object`

9.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Handle Table in _EPROCESS

```
nt!_EPROCESS
  +0x570 ObjectTable : Ptr64 _HANDLE_TABLE
```

Handle Table Core Structure

```
nt!_HANDLE_TABLE
  +0x000 TableCode          : Uint8
  +0x008 QuotaProcess       : Ptr64 _EPROCESS
  +0x010 HandleTableList    : _LIST_ENTRY
  +0x020 HandleCount        : Uint4B
  +0x028 Flags              : Uint4B
```

Handle Table Entry

```
nt!_HANDLE_TABLE_ENTRY
  +0x000 Object             : Ptr64 _EX_FAST_REF
  +0x008 GrantedAccessBits  : Uint4B
  +0x00c Attributes         : Uint4B
```

Object Header (Referenced by Handle Entry)

```
nt!_OBJECT_HEADER
  +0x000 PointerCount  : Int8B
  +0x008 HandleCount   : Int8B
  +0x010 Type          : Ptr64 _OBJECT_TYPE
```

```
+0x018 NameInfoOffset
+0x019 HandleInfoOffset
```

9.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Handle Resolution During a System Call

Example: `NtReadFile`

1. User passes a handle value
2. Kernel enters via `syscall`
3. `ObReferenceObjectByHandle` executes
4. Handle table is selected from `_EPROCESS.ObjectTable`
5. Handle index resolves to `_HANDLE_TABLE_ENTRY`
6. Access mask validated
7. Object pointer extracted from `_EX_FAST_REF`
8. Object reference count incremented

Instruction-Level Resolution Core

```
mov     rdx, [rcx + EPROCESS.ObjectTable]
shr     r8,  2                      ; handle index
lea     rdx, [rdx + r8*HANDLE_ENTRY_SIZE]
mov     rax, [rdx]                  ; EX_FAST_REF object pointer
and     rax, 0FFFFFFFFFFFFFFF0h
```

9.3.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Handle access to Secure Kernel objects is prohibited
- Token handles are validated by VTL1 policies
- Protected Process Light (PPL) handles are filtered
- Cross-VTL object duplication is blocked

HVCI enforces that no unsigned kernel driver can:

- Forge handle entries
- Modify `_HANDLE_TABLE` memory
- Patch Object Manager routines

9.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Handle lookups directly impact:

- L1/L2 cache hit rate (handle table walks)
- Branch prediction (access mask checks)
- NUMA-locality (remote process object tables)
- Lock contention on shared handle tables

High-frequency handle usage appears heavily in:

- File I/O

- ALPC messaging
- Registry operations
- Thread synchronization

9.3.7 REAL Practical Example (C / C++ / Assembly)

Kernel Handle to Object Resolution

```
PVOID Object;
NTSTATUS status = ObReferenceObjectByHandle(
    Handle,
    PROCESS_QUERY_INFORMATION,
    *PsProcessType,
    KernelMode,
    &Object,
    NULL
);
```

Kernel Handle Closure

```
ZwClose(Handle);
```

External Assembly Handle Resolver Stub

```
global ResolveHandleObject
ResolveHandleObject:
    mov     rdx, [rcx + EPROCESS.ObjectTable]
    shr     r8,  2
    lea     rdx, [rdx + r8*16]
    mov     rax, [rdx]
    and     rax, 0FFFFFFFFFFFFFFF0h
    ret
```

9.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

List process handles:

```
!process -1 1
```

Dump handle table:

```
!handle 0 3
```

Dump specific handle:

```
!handle <HandleValue> 1
```

Dump handle entry:

```
dt nt!_HANDLE_TABLE_ENTRY <Address>
```

9.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical handle-related attack categories:

- Use-after-free on handle entries
- Handle reuse after object destruction
- Incorrect access mask enforcement
- Cross-process handle duplication abuse

Windows 11 mitigations:

- Encoded object pointers
- Strong handle reference validation
- PPL and VBS isolation
- SMAP/SMEP enforcement

Direct handle manipulation from kernel drivers is now treated as a critical integrity violation.

9.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never cache handle values across process contexts.
- Always validate access masks explicitly.
- Never assume handle value stability.
- Do not dereference handle table entries directly in drivers.
- Always use `ObReferenceObjectByHandle`.
- Never duplicate SYSTEM handles into user processes.
- Always close kernel handles explicitly.

9.4 The Difference Between a Process and a Job Object

9.4.1 Precise Windows 11Specific Definition (Engineering-Level)

A **process** in Windows 11 is an executive object represented by `_EPROCESS` that encapsulates:

- Virtual address space
- Handle table
- Security token
- Thread list
- Working set

A **Job Object** is a kernel executive object represented by `_EJOB` that provides **group-level resource control and policy enforcement across one or more processes**.
A process is an *execution container*. A job object is a *control container*.

9.4.2 Exact Architectural Role Inside the Windows 11 Kernel

Process Role:

- Fundamental scheduling unit owner
- Virtual memory ownership
- Security boundary endpoint

Job Object Role:

- Enforces CPU, memory, I/O, and process-creation constraints
- Implements process lifetime governance
- Serves as the core containment primitive for AppContainers, Containers, and Sandbox subsystems

Windows 11 uses Job Objects heavily for:

- Windows Defender sandboxing
- AppContainer isolation
- Modern browser process isolation
- UWP process groups (even though UWP is not part of this book, the kernel mechanism is)

9.4.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_EPROCESS Core Fields

```
nt!_EPROCESS
  +0x570 ObjectTable        : Ptr64 _HANDLE_TABLE
  +0x4b8 Token              : _EX_FAST_REF
  +0x440 VadRoot            : _RTL_AVL_TREE
  +0x5a8 JobLinks           : _LIST_ENTRY
  +0x5b8 Job                : Ptr64 _EJOB
```

_EJOB Core Fields

```
nt!_EJOB
  +0x000 EventListHead      : _LIST_ENTRY
  +0x030 ProcessListHead    : _LIST_ENTRY
  +0x090 LimitFlags         : Uint4B
  +0x094 ActiveProcessLimit: Uint4B
  +0x0b0 TotalUserTime      : Uint8B
  +0x0b8 TotalKernelTime    : Uint8B
  +0x1d8 JobToken           : _EX_FAST_REF
```

Relationship

```
EPROCESS --> EJOB (Many-to-One Mapping)
```

9.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

Process Creation Without Job Binding

1. `NtCreateUserProcess`
2. `PspAllocateProcess`

3. `PspInsertProcess`
4. Scheduler inserts threads

Process Creation With Job Binding

1. `NtCreateUserProcess`
2. `ObReferenceObject(JobHandle)`
3. `PspBindProcessToJob`
4. `PspApplyJobLimits`
5. `PspInsertProcess`

Instruction-Level Job Attachment (Kernel Path)

```
mov     rax, [rcx + EPROCESS.Job]
test    rax, rax
jnz     ProcessAlreadyInJob
mov     [rcx + EPROCESS.Job], rdx
```

9.4.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Job limits affecting token privilege restrictions are validated in VTL1
- Protected processes cannot be assigned to untrusted job objects
- HVCI blocks kernel drivers from modifying `_EJOB` limit flags

Job Objects are used as:

- Containment boundaries for secure process groups
- Policy enforcement layers between user-mode and the Secure Kernel

9.4.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Process:

- Owns TLB context
- Owns page directory base (CR3)
- Direct scheduling cost

Job Object:

- No direct TLB ownership
- Adds accounting overhead
- Adds enforcement checks on:
  - Process creation
  - Thread creation
  - I/O operations

High job limit density increases:

- Branch mispredictions
- Lock contention on `_EJOB.ProcessListHead`

9.4.7 REAL Practical Example (C / C++ / Assembly)

Kernel: Query Job of a Process

```
PEJOB Job = PsGetProcessJob(PsGetCurrentProcess());
```

Kernel: Check If Process Is in a Job

```
if (PsIsProcessInJob(Process)) {
    // Process is governed by job limits
}
```

External Assembly: Reading Job Pointer

```
global GetProcessJob
GetProcessJob:
    mov     rax, [rcx + EPROCESS.Job]
    ret
```

9.4.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump process job association:

```
dt nt!_EPROCESS <ProcessAddress> Job
```

Dump job object:

```
dt nt!_EJOB <JobAddress>
```

List processes inside a job:

```
!job 0x<JobAddress>
```

9.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

Process Attacks:

- Token theft
- Handle table abuse
- VAD manipulation

Job Object Attacks:

- Job escape via handle duplication
- Incorrect limit inheritance
- Race conditions during job binding

Windows 11 mitigations include:

- Encoded job object pointers
- Protected Process Light restrictions
- Secure Kernel validation of job-policy-sensitive processes

9.4.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- A process can exist without a job; a job cannot exist without processes.
- A process can belong to only one job at a time.
- Job limits are inherited by child processes by default.
- Never assume job limits are advisory; many are enforced synchronously in the scheduler and memory manager.

- Do not modify `_EJOB` fields directly from drivers.
- Always treat job objects as security boundaries.

Chapter 10

Threads, Stacks & Context Switching in Assembly

10.1 _KTHREAD and _ETHREAD

10.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, a **thread** is the minimal schedulable execution entity. It is represented by two tightly coupled kernel objects:

- **`_KTHREAD`** Pure scheduler and execution state object
- **`_ETHREAD`** Executive thread object that embeds `_KTHREAD` and extends it with security, I/O, APC, and object-manager integration

`_KTHREAD` is owned by the scheduler. `_ETHREAD` is owned by the Executive, I/O manager, and security subsystem.

10.1.2 Exact Architectural Role Inside the Windows 11 Kernel

_KTHREAD provides:

- CPU context storage
- Kernel stack management
- Scheduling state
- Wait blocks
- Affinity, priority, and quantum tracking

_ETHREAD provides:

- Link to owning process (`_EPROCESS`)
- Security impersonation
- APC queues
- I/O request tracking
- Thread object reference counting

The scheduler operates **exclusively on `_KTHREAD`**. The Object Manager, Security Subsystem, and I/O Manager operate on **`_ETHREAD`**.

10.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTHREAD (Core Fields)

```
nt!_KTHREAD
  +0x000 Header             : _DISPATCHER_HEADER
```

```
  +0x030 SListFaultAddress : Ptr64
  +0x038 QuantumTarget     : Uint8B
  +0x040 InitialStack      : Ptr64
  +0x048 StackLimit        : Ptr64
  +0x050 KernelStack       : Ptr64
  +0x058 ThreadLock        : Uint8B
  +0x074 State             : UChar
  +0x0b0 Affinity          : Uint8B
  +0x100 WaitBlockList     : _LIST_ENTRY
```

_ETHREAD (Core Fields)

```
nt!_ETHREAD
  +0x000 Tcb               : _KTHREAD
  +0x430 CreateTime        : _LARGE_INTEGER
  +0x4b8 ExitTime          : _LARGE_INTEGER
  +0x580 Cid               : _CLIENT_ID
  +0x5b0 TerminationPort   : Ptr64
  +0x5f8 ThreadsProcess    : Ptr64 _EPROCESS
  +0x600 ThreadListEntry   : _LIST_ENTRY
```

Structural Relationship

```
ETHREAD
  KTHREAD (embedded at offset 0x0)
```

10.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Thread Creation Path

1. `NtCreateThreadEx`
2. `PspAllocateThread`

3. KeInitializeThread
4. KeReadyThread
5. KiInsertReadyQueue

First Context Switch Into a Thread

1. Scheduler selects _KTHREAD
2. KiSwapThread invoked
3. Old thread context saved into _KTHREAD
4. New thread kernel stack loaded
5. SwapContext restores registers

Instruction-Level Context Switch Core (Conceptual)

```
; rcx = OldKthread, rdx = NewKthread

mov     [rcx + KTHREAD.KernelStack], rsp
mov     rsp, [rdx + KTHREAD.KernelStack]

fxsave64 [rcx + KTHREAD.FpuState]
fxrstor64 [rdx + KTHREAD.FpuState]

ret
```

10.1.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Thread creation is validated against Secure Kernel trust policies
- User-mode thread start RIP is validated under VTL1
- HVCI enforces integrity of:
  - Kernel stack pointers
  - Context switch code
  - Saved register frames

Protected Processes enforce additional restrictions:

- Only signed code may execute on threads
- Thread impersonation is restricted at token level

10.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Thread switching cost components:

- Register save/restore
- Kernel stack cache pollution
- Branch predictor invalidation
- Core migration across NUMA nodes

Key costs:

- L1/L2 cache invalidation
- TLB flush on address space change

- Scheduler lock contention

Hybrid CPU scheduling:

- `_KTHREAD.Affinity` directs P-core vs E-core placement
- Latency-sensitive threads are biased toward P-cores

10.1.7 REAL Practical Example (C / C++ / Assembly)

Kernel: Obtain Current ETHREAD

```
PETHREAD Thread = PsGetCurrentThread();
```

Kernel: Access Embedded KTHREAD

```
PKTHREAD Kthread = &Thread->Tcb;
```

External Assembly: Read Kernel Stack Pointer

```
global GetKernelStack
GetKernelStack:
    mov     rax, [rcx + KTHREAD.KernelStack]
    ret
```

10.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump current thread:

```
!thread
```

Dump ETHREAD structure:

```
dt nt!_ETHREAD <ThreadAddress>
```

Dump embedded KTHREAD:

```
dt nt!_KTHREAD <ThreadAddress>
```

Display thread stack:

```
!stk
```

10.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Kernel stack corruption
- APC queue manipulation
- Thread token impersonation
- Stack pivoting via ROP

Windows 11 mitigations:

- Kernel stack randomization
- Shadow stacks (CET)
- HVCI-protected context switch paths
- VTL1 validation of control-flow transfers

10.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- `_KTHREAD` is not reference-counted; `_ETHREAD` is.
- Never store persistent pointers to `_KTHREAD` across context switches.
- Kernel stack overflows corrupt adjacent scheduler data.
- APC delivery is tied directly to `_KTHREAD.WaitBlockList`.
- Context switching is one of the most heavily protected execution paths under HVCI.
- Any corruption of `_KTHREAD.KernelStack` results in immediate bugcheck.

10.2 Thread Stack Layout

10.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, the **thread stack** is a per-thread, kernel-controlled memory region used to store:

- Function call frames
- Interrupt and exception frames
- Saved volatile CPU state
- APC delivery context
- Context switch save areas

Each thread owns:

- One **user-mode stack** (Ring 3)
- One **kernel-mode stack** (Ring 0)

The kernel stack is pointed to by:

```
_KTHREAD.KernelStack
_KTHREAD.InitialStack
_KTHREAD.StackLimit
```

10.2.2 Exact Architectural Role Inside the Windows 11 Kernel

The kernel stack is the **execution anchor** for:

- System call entry (`syscall`)
- Interrupt handling
- DPC/APC execution
- Scheduler preemption
- Exception dispatch

Every transition from Ring 3 to Ring 0 immediately switches to the **current thread kernel stack**. No kernel execution is allowed on the user stack.

10.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Stack Pointers in _KTHREAD

```
nt!_KTHREAD
  +0x040 InitialStack    : Ptr64
  +0x048 StackLimit      : Ptr64
  +0x050 KernelStack     : Ptr64
```

Stack Location Summary

```
InitialStack  --> Top of kernel stack (highest VA)
StackLimit    --> Bottom of kernel stack (guard page)
KernelStack   --> Current RSP while executing in Ring 0
```

10.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

System Call Entry Stack Transition

1. User executes `syscall`
2. CPU loads kernel RSP from MSR `IA32_KERNEL_GS_BASE`
3. Kernel stack pointer loaded from current `_KTHREAD`
4. Execution continues at `KiSystemCall64`

Interrupt Entry Stack Transition

1. CPU validates IDT gate
2. Interrupt stack loaded (IST or normal kernel stack)
3. Trap frame pushed onto kernel stack
4. Dispatcher invoked

Context Switch Stack Exchange (Conceptual)

```
; rcx = OldKthread, rdx = NewKthread

mov     [rcx + KTHREAD.KernelStack], rsp
mov     rsp, [rdx + KTHREAD.KernelStack]
ret
```

10.2.5 Secure Kernel / VBS / HVCI Interaction

With VBS enabled:

- Kernel stacks are integrity-protected under VTL1
- Shadow stacks (CET) mirror return addresses
- HVCI enforces stack execution policies
- Stack switching logic is hypervisor-validated

Protected threads:

- Use hardened kernel stacks
- Disallow stack pivoting

10.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Stack performance impact sources:

- Cache line pollution during deep call chains
- TLB pressure during stack growth
- NUMA migration increases stack memory latency

Windows 11 optimization strategies:

- Thread stack locality on the owning NUMA node
- Dedicated interrupt stacks (IST)
- Stack probing during thread creation

10.2.7 REAL Practical Example (C / C++ / Assembly)

Kernel C: Read Kernel Stack of Current Thread

```
PKTHREAD Kt = (PKTHREAD)PsGetCurrentThread();
PVOID Stack = Kt->KernelStack;
```

External Assembly: Return Kernel Stack Pointer

```
global GetCurrentKernelStack
GetCurrentKernelStack:
    mov     rax, [rcx + 0x50]   ; KTHREAD.KernelStack
    ret
```

Stack Growth Direction (x86-64)

```
High VA  -> InitialStack
            |
            |  <-- Stack grows downward (RSP--)
            |
Low VA   -> StackLimit (Guard Page)
```

10.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump current thread stack:

```
!thread
```

Dump kernel stack:

```
!kstack
```

Display stack bounds:

```
dt nt!_KTHREAD <Thread> InitialStack StackLimit KernelStack
```

Show active call frames:

```
k
```

10.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary kernel stack attack vectors:

- Stack buffer overflows
- ROP-based stack pivoting
- Corruption of saved return RIP
- APC stack injection

Windows 11 mitigations:

- Guard pages at StackLimit
- CET shadow stacks
- HVCI-protected return flow
- NX enforced on stack pages

10.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Kernel stack overflows cause immediate bugcheck.
- Never assume a fixed stack size across Windows builds.
- APC delivery always occurs on the kernel stack.

- Interrupts may switch to IST instead of normal kernel stacks.
- Stack corruption compromises the scheduler and leads to fatal system instability.
- Shadow stacks under CET make traditional ROP unusable in modern Windows 11.

10.3 Context Switching Instruction by Instruction

10.3.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, a **context switch** is the low-level kernel operation that suspends execution of one thread and restores execution of another by:

- Saving the complete volatile CPU execution state of the outgoing thread
- Restoring the complete CPU state of the incoming thread
- Switching kernel stack pointers
- Updating scheduler ownership fields

The authoritative implementation resides in:

```
ntoskrnl.exe!KiContextSwitch
ntoskrnl.exe!SwapContext
```

This mechanism executes entirely in Ring 0 and is verified under HVCI when enabled.

10.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Context switching is the **core execution primitive** of:

- Preemptive multitasking

- Hybrid CPU core scheduling (P-Cores / E-Cores)
- APC/DPC delivery
- IRQL-based preemption control

Every scheduling quantum expiration, wait completion, APC delivery, or priority inversion correction flows through the context switch path.

10.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTHREAD Core Fields Used in Context Switching

```
nt!_KTHREAD
  +0x028 InitialStack
  +0x030 StackLimit
  +0x038 KernelStack
  +0x048 State
  +0x050 Priority
  +0x058 ContextSwitches
  +0x2D0 TrapFrame
  +0x2D8 ApcState
```

_KPRCB Scheduler Control Block

```
nt!_KPRCB
  +0x000 CurrentThread
  +0x008 NextThread
  +0x010 IdleThread
```

_KTRAP_FRAME (Saved CPU Context)

```
nt!_KTRAP_FRAME
  +0x000 P1Home
```

```
+0x008 Rax
+0x010 Rcx
+0x018 Rdx
+0x020 R8
+0x028 R9
+0x030 R10
+0x038 R11
+0x040 GsBase
+0x048 Rip
+0x050 SegCs
+0x058 EFlags
+0x060 Rsp
```

10.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

High-Level Context Switch Flow

1. Scheduler selects a new runnable thread
2. Current thread state is saved into its _KTRAP_FRAME
3. Kernel stack pointer is saved into _KTHREAD.KernelStack
4. New thread kernel stack is loaded
5. New thread trap frame is restored
6. Execution resumes at restored RIP

Instruction-Level Save Phase (Outgoing Thread)

```
; RCX = OldKthread
; Save kernel stack
mov     [rcx + 0x38], rsp
```

```
; Save volatile registers into trap frame
mov     [rcx + 0x2D0 + 0x08], rax
mov     [rcx + 0x2D0 + 0x10], rcx
mov     [rcx + 0x2D0 + 0x18], rdx
mov     [rcx + 0x2D0 + 0x40], gs
pushfq
pop     [rcx + 0x2D0 + 0x58]
```

Instruction-Level Restore Phase (Incoming Thread)

```
; RCX = NewKthread
; Load kernel stack
mov     rsp, [rcx + 0x38]

; Restore registers
mov     rax, [rcx + 0x2D0 + 0x08]
mov     rcx, [rcx + 0x2D0 + 0x10]
mov     rdx, [rcx + 0x2D0 + 0x18]
push    [rcx + 0x2D0 + 0x58]
popfq
```

Instruction-Level Control Transfer

```
jmp     [rcx + 0x2D0 + 0x48]   ; Jump to restored RIP
```

This final jump resumes execution of the incoming thread at its exact suspended instruction.

10.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- All context switch code executes inside VTL0 under hypervisor monitoring

- HVCI verifies integrity of:
  - Trap frame memory
  - Stack memory
  - Context switch code paths
- CET shadow stacks mirror return addresses across switches
- Stack pointers are validated against shadow stack metadata

Illegitimate stack pivots immediately trigger secure kernel violations.

10.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Each context switch incurs:

- L1/L2 cache displacement
- TLB churn
- Branch predictor invalidation

Windows 11 mitigations:

- Core parking awareness for hybrid CPUs
- NUMA-local thread resumption
- Quantum-length tuning by scheduler

Excessive context switching directly reduces IPC (Instructions Per Cycle).

10.3.7 REAL Practical Example (C / C++ / Assembly)

Kernel C: Force Context Yield

```
KeYieldProcessor();
```

External Assembly: Manual Register Snapshot

```
global SaveVolatileContext
SaveVolatileContext:
    mov     [rcx + 0x00], rax
    mov     [rcx + 0x08], rcx
    mov     [rcx + 0x10], rdx
    mov     [rcx + 0x18], r8
    mov     [rcx + 0x20], r9
    mov     [rcx + 0x28], r10
    mov     [rcx + 0x30], r11
    ret
```

10.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display current and next threads:

```
!prcb
```

Dump trap frame:

```
dt nt!_KTRAP_FRAME <address>
```

Show context switch count:

```
dt nt!_KTHREAD <thread> ContextSwitches
```

Live context trace:

```
!thread
```

10.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Trap frame corruption
- Stack pivot injection
- Register spoofing
- Return RIP modification

Windows 11 defenses:

- CET shadow stacks
- NX enforcement
- HVCI context verification
- Kernel CFG (Control Flow Guard)

Any invalid restoration results in immediate bugcheck or secure kernel termination.

10.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Every context switch touches hundreds of CPU microarchitectural structures.
- Stack misalignment during switching causes immediate kernel crashes.
- IRQL violations during switching lead to deadlocks.
- APC delivery is deferred during active context switch windows.

- Hybrid CPU scheduling increases the cost of cold-cache thread migration.
- Modern Windows 11 context switches are hypervisor-validated when VBS is active.

Part VI

Scheduler Internals & Modern CPU Awareness

Chapter 11

Windows 11 Scheduler Architecture

11.1 P-Cores / E-Cores

11.1.1 Precise Windows 11Specific Definition (Engineering-Level)

Windows 11 implements a **heterogeneous-aware scheduler** designed for modern hybrid CPUs that combine:

- **Performance Cores (P-Cores)**: High-frequency, wide-issue, out-of-order cores optimized for latency-sensitive workloads.
- **Efficiency Cores (E-Cores)**: Power-optimized, throughput-oriented cores optimized for background and parallel tasks.

The Windows 11 scheduler integrates this topology through **hardware feedback from Intel Thread Director and AMD CPPC2 telemetry**, exposed to the kernel through HAL and ACPI processor objects.

11.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The hybrid scheduler logic is distributed across:

- `ntoskrnl.exe` Core scheduling logic
- `hal.dll` CPU topology abstraction
- **ACPI PPTT / SRAT tables** Hardware topology discovery
- **Scheduler classes** Foreground, background, and real-time

The scheduler dynamically decides:

- Which core class (P or E) executes a thread
- When a thread must migrate between core classes
- How power, thermals, and latency constraints influence dispatch

11.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KPRCB (Per-Processor Control Block)

```
nt!_KPRCB
  +0x000 CurrentThread
  +0x008 NextThread
  +0x020 IdleThread
  +0x2C0 SchedulingGroup
  +0x2D8 CoreId
  +0x2E0 EfficiencyClass
```

_KTHREAD Scheduling Fields

```
nt!_KTHREAD
  +0x048 State
  +0x050 Priority
  +0x054 BasePriority
  +0x058 ContextSwitches
  +0x074 IdealProcessor
  +0x2B0 SchedulingGroup
```

_KSCHEDULING_GROUP

```
nt!_KSCHEDULING_GROUP
  +0x000 RunningSummary
  +0x040 Weight
  +0x048 MaxConcurrency
```

11.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Thread Dispatch Decision Flow

1. Thread becomes Ready (`KTHREAD.State = Ready`)
2. Scheduler evaluates:
    - Priority class
    - Core efficiency class
    - Cache locality
    - NUMA locality
3. If thread is latency-sensitive:
    - Prefer dispatch on P-Core

4. If thread is background or power-efficient:
    - Prefer dispatch on E-Core
5. `KiSelectReadyThread` selects target PRCB
6. Context switch is initiated

Assembly-Level Dispatch Control (Conceptual, External)

```
; RCX = Target PRCB
; RDX = Selected KTHREAD

mov     [rcx + 0x08], rdx     ; PRCB.NextThread = KTHREAD
call    KiContextSwitch       ; Enter low-level context switch
```

11.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- Scheduler decisions are monitored by the hypervisor
- Thread dispatch transitions are verified under HVCI
- CET shadow stacks track return addresses across core migrations
- Kernel stack pointers are validated during migration between cores

Any stack corruption during P-Core to E-Core migration triggers immediate secure kernel termination.

11.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- **P-Core Scheduling**
  - Higher IPC
  - Larger private caches
  - Better branch prediction
- **E-Core Scheduling**
  - Lower power consumption
  - Shared resources
  - Optimized for background parallelism

Windows 11 actively minimizes:

- Cross-core cache invalidation
- TLB shootdowns during migrations
- NUMA remote execution penalties

11.1.7 REAL Practical Example (C / C++ / Assembly)

Kernel C: Hinting Ideal Processor

```
KeSetSystemAffinityThreadEx(DesiredAffinityMask);
```

External Assembly: Reading Processor Number

```
global GetCurrentProcessor
GetCurrentProcessor:
    mov     eax, 1
    cpuid
    shr     ebx, 24
    mov     eax, ebx
    ret
```

This allows verifying whether a thread is executing on a P-Core or E-Core.

11.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display per-core scheduler state:

```
!prcb
```

Display current thread scheduling details:

```
!thread
```

Inspect processor topology:

```
!cpuinfo
```

Check ideal processor and affinity:

```
dt nt!_KTHREAD <addr> IdealProcessor
```

11.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Thread affinity manipulation
- Scheduler group weight corruption
- Stack migration attacks during core switching

Windows 11 defenses:

- HVCI scheduler validation
- CET shadow stacks
- Kernel CFG
- Write-protected PRCB structures

11.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Incorrect affinity enforcement destroys scheduler load balancing.
- Excessive cross-core migration causes severe cache thrashing.
- IRQL violations during hybrid dispatch lead to immediate system deadlock.
- P-Core starvation results in observable UI latency.
- E-Core saturation leads to background task collapse.
- Hypervisor-aware scheduling is mandatory under Windows 11 security model.

11.2 NUMA

11.2.1 Precise Windows 11Specific Definition (Engineering-Level)

Non-Uniform Memory Access (NUMA) in Windows 11 is a **hardware-topologyaware memory and scheduling model** where:

- Each processor group is associated with a **local memory node**
- Memory access latency depends on whether the memory is **local or remote**
- Scheduling decisions are optimized for **memory locality**

Windows 11 implements NUMA as a **first-class kernel scheduling and memory management construct**, not as an optional optimization layer.

11.2.2 Exact Architectural Role Inside the Windows 11 Kernel

NUMA influences the following kernel subsystems:

- Scheduler thread placement
- Physical page allocation
- Cache locality enforcement
- Interrupt steering
- I/O DMA routing

Primary NUMA-aware kernel components:

- `ntoskrnl.exe` Core scheduling and memory policies

- `hal.dll` Node-to-processor mapping
- ACPI SRAT / SLIT tables Hardware topology description

11.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KNODE (NUMA Node Descriptor)

```
nt!_KNODE
  +0x000 NodeNumber
  +0x008 ProcessorMask
  +0x020 PfnRanges
  +0x048 Distance
  +0x080 FreeCount
```

_KPRCB (NUMA Binding Fields)

```
nt!_KPRCB
  +0x2E0 NodeNumber
  +0x2E8 MemoryNode
  +0x2F0 ParentNode
```

_MMNODE

```
nt!_MMNODE
  +0x000 NodeNumber
  +0x020 PfnDatabase
  +0x040 FreePageCount
  +0x080 TotalPages
```

11.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

NUMA-Aware Thread Scheduling Flow

1. Thread becomes Ready
2. Scheduler queries IdealNode from `KTHREAD`
3. Scheduler queries `KPRCB.NodeNumber`
4. If IdealNode matches PRCB Node:
    - Dispatch locally
5. Else:
    - Evaluate migration cost
    - Evaluate cache and TLB penalty
    - Possibly migrate thread

Physical Page Allocation Flow

1. Allocation request arrives
2. Preferred NUMA node is selected
3. `MiAllocateFromNode` is invoked
4. PFN database of target node is updated

Assembly-Level Node Identification (External)

```
global GetNumaNode
GetNumaNode:
    mov     eax, 0Bh
    xor     ecx, ecx
```

```
    cpuid
    mov     eax, edx
    ret
```

11.2.5 Secure Kernel / VBS / HVCI Interaction

Under VBS:

- NUMA node mappings are validated by the hypervisor
- Cross-node page remapping is audited by Secure Kernel
- DMA protection enforces node isolation
- Remote memory mapping attempts are verified through EPT/NPT

Any illegal PFN remap between nodes results in immediate bugcheck.

11.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- **Local Node Access**
  - Lowest latency
  - Maximal cache hit rate
  - Minimal QPI/InfinityFabric traffic
- **Remote Node Access**
  - Increased memory latency
  - Elevated TLB miss rate
  - Interconnect congestion

Windows 11 scheduler aggressively avoids:

- Cross-node thread migration
- Remote memory allocations for hot threads
- Interrupt dispatch across NUMA boundaries

11.2.7 REAL Practical Example (C / C++ / Assembly)

Kernel C: NUMA-Aware Allocation

```
PVOID Buffer = MmAllocatePagesForMdlEx(
    LowAddress,
    HighAddress,
    SkipBytes,
    Size,
    MmNonCached,
    MM_ALLOCATE_FROM_LOCAL_NODE
);
```

External Assembly: Reading NUMA-Aware Processor ID

```
global GetProcessorId
GetProcessorId:
    mov     eax, 1
    cpuid
    shr     ebx, 24
    mov     eax, ebx
    ret
```

11.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display NUMA nodes:

```
!numa
```

Display per-node memory usage:

```
!memusage 1
```

Inspect a NUMA node:

```
dt nt!_KNODE <addr>
```

Inspect PFN node bindings:

```
!pfn
```

11.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary attack surfaces:

- Cross-node page remapping
- PFN database corruption
- Remote DMA abuse

Windows 11 defenses:

- VBS-enforced memory isolation
- IOMMU node-bound DMA protection
- Hypervisor-controlled PFN access
- Secure Kernel page ownership validation

11.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Remote NUMA allocations destroy deterministic latency.
- Incorrect NUMA affinity leads to catastrophic cache thrashing.
- Mixing high-IRQL code with remote memory access is forbidden.
- NUMA-unaware drivers cause severe performance regressions.
- DMA buffers must always be allocated on the local node.
- Cross-node interrupt routing causes immediate scheduling instability.

11.3 Priority Boosting

11.3.1 Precise Windows 11Specific Definition (Engineering-Level)

Priority boosting in Windows 11 is a **dynamic, kernel-enforced scheduling adjustment mechanism** that temporarily elevates a threads base dynamic priority above its original base priority in order to:

- Reduce I/O completion latency
- Prevent starvation
- Improve interactive responsiveness
- Enforce forward progress under contention

Priority boosting is applied exclusively at **dispatch-time** and is always **time-decayed**. The boost never permanently alters the threads base priority stored in `_KTHREAD.BasePriority`.

11.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Priority boosting is a **core responsibility of the scheduler fast-path** located primarily within:

- `ntoskrnl.exe` Scheduler dispatch and quantum expiration logic
- `KiDeferredReadyThread`
- `KiBoostThreadIo`

It directly impacts:

- Ready queue placement
- Dispatcher database lock behavior
- Quantum recalculation
- Run queue ordering

Priority boosting is applied when:

- APC completion occurs
- I/O request completes
- GUI input is delivered to a thread
- Synchronization wait is released

11.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTHREAD Fields

```
nt!_KTHREAD
  +0x01A BasePriority
  +0x01B Priority
  +0x01C Quantum
  +0x01D WaitReason
  +0x1F0 WaitBlockList
  +0x2A0 PreviousMode
```

_KPRCB Dispatch State

```
nt!_KPRCB
  +0x020 CurrentThread
  +0x028 NextThread
  +0x2E8 MemoryNode
  +0x360 ReadySummary
```

11.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

I/O Completion Boost Flow

1. Thread enters wait state (e.g., `KeWaitForSingleObject`)
2. I/O completes
3. `KiUnwaitThread` invoked
4. Scheduler evaluates wait reason
5. `KiBoostThreadIo` executed
6. `KTHREAD.Priority` increased above `BasePriority`

7. Thread is inserted into boosted ready queue

Boost Decay Flow

1. Thread dispatched at boosted priority
2. Quantum expiration occurs
3. Priority decays by one level per dispatch
4. Eventually returns to BasePriority

External Assembly: Runtime Thread Priority Observation

```
global GetCurrentThreadPriority
extern NtCurrentTeb

GetCurrentThreadPriority:
    sub     rsp, 40h
    call    NtCurrentTeb
    mov     rax, [rax + 30h]
    mov     al,  [rax + 1Bh]
    add     rsp, 40h
    ret
```

11.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 VBS:

- Secure Kernel threads never receive user-mode boosts
- Hypervisor-enforced scheduling prevents manipulation of boost fields
- Boost logic is validated through control-flow integrity checks

- Scheduler metadata is protected by HVCI

Any kernel attempt to artificially inflate priority without proper dispatcher context is blocked.

11.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Boosting reduces input latency for interactive threads
- High-frequency boosting increases:
  - Context switching
  - Cache invalidation
  - TLB pressure
- Excessive boost usage leads to:
  - Starvation of background threads
  - Dispatch imbalance across P-Cores

Windows 11 limits boost magnitude and decay rate to maintain global fairness.

11.3.7 REAL Practical Example (C / C++ / Assembly)

Kernel C: Explicit Temporary Boost via APC Completion

```
VOID IoCompletionRoutine(
    PVOID Context,
    PIO_STATUS_BLOCK IoStatus,
    ULONG Reserved
)
{
    UNREFERENCED_PARAMETER(Context);
```

```
    UNREFERENCED_PARAMETER(IoStatus);
    UNREFERENCED_PARAMETER(Reserved);
}
```

I/O completion automatically triggers scheduler boost.

External Assembly: Observing Quantum Reduction

```
global ReadThreadQuantum
extern NtCurrentTeb

ReadThreadQuantum:
    sub     rsp, 40h
    call    NtCurrentTeb
    mov     rax, [rax + 30h]
    mov     al, [rax + 1Ch]
    add     rsp, 40h
    ret
```

11.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect a running thread:

```
!thread
```

Dump KTHREAD:

```
dt nt!_KTHREAD <addr>
```

View ready queues:

```
!ready
```

Inspect dispatcher state:

```
!prcb
```

11.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Attack targets:

- Manual priority inflation via corrupted KTHREAD
- Ready queue manipulation
- APC-based starvation attacks

Windows 11 defenses:

- HVCI enforcement on scheduler routines
- Secure Kernel validation of dispatcher transitions
- Non-writable dispatcher queues from user mode
- PatchGuard protection on KTHREAD fields

11.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never assume boosted priority persists across dispatch cycles.
- Boosting is always subordinate to IRQL.
- Boost abuse leads to cache thrashing on hybrid cores.
- Priority boosting cannot override real-time scheduling classes.
- Kernel drivers must never attempt manual priority injection.
- Incorrect assumptions about boost duration cause deadlocks.

11.4 Impact on C Performance and Assembly Performance

11.4.1 Precise Windows 11Specific Definition (Engineering-Level)

The performance impact of the Windows 11 scheduler on C and x86-64 assembly code is defined as the **direct influence of thread dispatch latency, core selection (P-Core vs E-Core), quantum management, priority boosting, NUMA locality, and cache topology awareness** on:

- Instruction throughput
- Memory access latency
- Branch prediction stability
- Pipeline utilization

This impact is entirely controlled by the kernel scheduler inside `ntoskrnl.exe` and the Hyper-Venforced execution environment.

11.4.2 Exact Architectural Role Inside the Windows 11 Kernel

The scheduler directly governs C and assembly performance via:

- **Thread placement policy** across P-Cores and E-Cores
- **Dynamic priority boosting**
- **NUMA-aware dispatch**
- **Time-slice (quantum) expiration**
- **IRQL-aware preemption rules**

Critical internal execution points include:

- `KiDispatchInterrupt`
- `KiSelectReadyThread`
- `KiSwapThread`
- `KiDeferredReadyThread`

All C and assembly code execute strictly inside these scheduling constraints.

11.4.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTHREAD

```
nt!_KTHREAD
  +0x01A BasePriority
  +0x01B Priority
  +0x01C Quantum
  +0x01D WaitReason
  +0x070 Affinity
  +0x078 IdealProcessor
  +0x184 ApcStateIndex
```

_KPRCB

```
nt!_KPRCB
  +0x020 CurrentThread
  +0x028 NextThread
  +0x0B0 ReadyListHead
  +0x2E8 MemoryNode
```

These structures directly determine how C and assembly instructions are scheduled for execution.

11.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

C Execution Dispatch Flow

1. C function enters execution in Ring 3
2. Thread is bound to a logical processor
3. Scheduler selects core based on:
    - Priority
    - NUMA node
    - Core type (P/E)
4. Timer interrupt triggers `KiDispatchInterrupt`
5. Context switch performed by `KiSwapThread`
6. Next C thread resumes execution

Assembly Execution Dispatch Flow

1. Assembly routine executes with strict time quantum
2. Any IRQL raise blocks preemption
3. Quantum expiration forces scheduler entry
4. General-purpose registers saved to KTHREAD stack
5. New thread state restored at instruction level

External Assembly: Measuring Dispatch Delay

```
global MeasureDispatchLatency
extern QueryPerformanceCounter

MeasureDispatchLatency:
    sub     rsp, 40h
    lea     rcx, [rsp + 20h]
    call    QueryPerformanceCounter
    mov     rax, [rsp + 20h]
    add     rsp, 40h
    ret
```

11.4.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 security enforcement:

- Secure Kernel isolates scheduler metadata
- HVCI prevents modification of dispatch routines
- Hypervisor validates thread state transitions
- User-mode assembly cannot influence core selection

This guarantees deterministic enforcement of scheduling impact on all C and assembly workloads.

11.4.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

C Performance Implications:

- High-level compiler optimizations are constrained by:

  - Quantum expiration
  - Core migration
  - NUMA page locality

Assembly Performance Implications:

- Pipeline stalls during migration
- TLB flush on core switch
- Cache invalidation across NUMA nodes

Hybrid CPU scheduling causes:

- P-Core migration improves IPC
- E-Core migration increases latency

11.4.7 REAL Practical Example (C / C++ / Assembly)

C Example: Priority-Sensitive Busy Loop

```c
volatile int flag;

void BusyThread(void)
{
    while (!flag)
    {
        _mm_pause();
    }
}
```

This loops execution rate directly depends on:

- Quantum length
- Priority boosting
- Core placement

External Assembly: Core-Sensitive Spin Loop

```
global SpinWait
SpinWait:
    mov     ecx, 1000000
.loop:
    pause
    dec     ecx
    jnz     .loop
    ret
```

Execution speed varies significantly across P-Cores and E-Cores.

11.4.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect executing thread:

```
!thread
```

Inspect scheduler queues:

```
!ready
```

Inspect processor block:

```
!prcb
```

Inspect NUMA node:

```
!numa
```

11.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

Attack vectors affecting performance:

- Priority inversion
- Core starvation
- NUMA thrashing
- APC flooding

Windows 11 protections:

- Scheduler invariants protected by HVCI
- Secure Kernel validates thread transitions
- PatchGuard protects KTHREAD scheduling fields

11.4.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- High-performance assembly must assume forced preemption.
- C performance tuning is meaningless without scheduler awareness.
- Never depend on uninterrupted execution unless IRQL $>$ DISPATCH_LEVEL.
- E-Core execution severely impacts tight spinlocks.
- NUMA locality outweighs instruction-level optimization.
- Priority boosting distorts deterministic benchmarking.

Part VII

IRQL, Interrupts, APC & DPC

Chapter 12

Hardware Interrupts from the CPU to the Kernel

12.1 IDT

12.1.1 Precise Windows 11Specific Definition (Engineering-Level)

The **Interrupt Descriptor Table (IDT)** in Windows 11 is a per-processor, hardware-resident dispatch table that maps **interrupt vectors (0–255)** to kernel-mode entry points inside `ntoskrnl.exe`. It is loaded into the CPU using the `LIDT` instruction and operates strictly in **Ring 0**. Every hardware interrupt, exception, and software-generated interrupt (including `INT n`) is resolved through this structure.

Windows 11 uses a **64-bit IDT with interrupt gates only**. Trap gates are not used for external hardware interrupts.

12.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The IDT is the **first kernel entry point for all asynchronous CPU events**. It:

- Bridges CPU hardware with the Windows 11 kernel interrupt dispatcher
- Transfers execution from arbitrary privilege levels into controlled Ring 0 code
- Enforces IRQL transitions
- Separates:
  - Exceptions (vectors 0–31)
  - Hardware IRQs (mapped via IOAPIC)
  - Software interrupts

The entry points referenced by the IDT terminate in:

- `KiInterruptDispatch`
- `KiPageFault`
- `KiGeneralProtectionFault`
- `KiDivideErrorFault`

12.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Hardware IDT Entry Format (x86-64)

```
typedef struct _IDT_ENTRY64 {
    USHORT OffsetLow;
    USHORT Selector;
```

```
    UCHAR  IstIndex : 3;
    UCHAR  Reserved0 : 5;
    UCHAR  Type : 5;
    UCHAR  Dpl : 2;
    UCHAR  Present : 1;
    USHORT OffsetMiddle;
    ULONG  OffsetHigh;
    ULONG  Reserved1;
} IDT_ENTRY64;
```

Kernel IDT Pointer

```
nt!KiInterruptDescriptorTable
```

KPCR Linkage

```
nt!_KPCR
  +0x038 IdtBase
```

Each logical processor owns its own IDT.

12.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Hardware Interrupt Arrival

1. Device asserts IRQ line
2. IOAPIC maps IRQ to interrupt vector
3. CPU performs vector lookup in IDT
4. CPU switches to Ring 0 stack from TSS
5. CPU pushes:

- RIP
- CS
- RFLAGS
- RSP
- SS

6. Control transfers to kernel interrupt stub

Kernel Dispatch Path

1. IDT stub executes in assembly
2. IRQL raised to DISPATCH_LEVEL or higher
3. `KiInterruptDispatch` executed
4. Registered ISR invoked
5. DPC queued if required
6. `IRETQ` returns execution

External Assembly IDT Stub Example

```
global InterruptStubExample

InterruptStubExample:
    sub     rsp, 88h
    mov     [rsp+20h], rax
    mov     [rsp+28h], rcx
    mov     [rsp+30h], rdx
    call    KiInterruptDispatch
```

```
    mov     rax, [rsp+20h]
    mov     rcx, [rsp+28h]
    mov     rdx, [rsp+30h]
    add     rsp, 88h
    iretq
```

12.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 security architecture:

- IDT itself remains in standard kernel memory (VTL0)
- Secure Kernel validates interrupt return paths
- HVCI prevents unauthorized modification of ISR targets
- PatchGuard monitors IDT consistency indirectly via ISR integrity

Direct runtime modification of IDT entries is **detected and fatal**.

12.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Each interrupt causes:
  - Pipeline flush
  - Stack switch
  - TLB pressure
- Cross-NUMA interrupt routing increases latency
- High interrupt rates reduce effective instruction throughput
- E-Core interrupt handling is slower than P-Core handling

12.1.7 REAL Practical Example (C / C++ / Assembly)

Kernel ISR Registration (C)

```
NTSTATUS IoConnectInterruptEx(
    _Inout_ PIO_INTERRUPT_INTERRUPT_PARAMETERS Parameters
);
```

ISR Function (C)

```
BOOLEAN MyInterruptServiceRoutine(
    PKINTERRUPT Interrupt,
    PVOID ServiceContext
)
{
    UNREFERENCED_PARAMETER(Interrupt);
    UNREFERENCED_PARAMETER(ServiceContext);
    return TRUE;
}
```

External Assembly Interrupt Timestamping

```
global ReadTSC
ReadTSC:
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    ret
```

Used to measure interrupt entry latency.

12.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display IDT base:

```
r idtr
```

Dump IDT:

```
!idt
```

Inspect interrupt object:

```
!interrupts
```

Trace ISR execution:

```
!thread
!stacks 2
```

12.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack vectors:

- IDT hooking
- Fake ISR redirection
- Stack pivot during interrupt entry

Windows 11 protections:

- PatchGuard protects ISR code
- HVCI prevents unsigned ISR execution
- Secure Kernel blocks privilege escalation via interrupt abuse

12.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- IDT entries must never be modified at runtime.
- ISR code must execute at elevated IRQL and remain non-blocking.
- Stack corruption inside ISR is unrecoverable.
- All registers must be preserved correctly by the interrupt stub.
- Nested interrupts amplify TLB and cache pressure.
- ISR latency dominates real-time device performance.

12.2 ISR

12.2.1 Precise Windows 11Specific Definition (Engineering-Level)

An **Interrupt Service Routine (ISR)** in Windows 11 is a kernel-mode, hardware-bound execution handler that runs at **device IRQL (DIRQL)** immediately after an interrupt vector is dispatched through the IDT. The ISR executes in **Ring 0**, preempts all normal thread execution, and is responsible for performing the **minimum possible work** required to acknowledge the device and queue deferred processing.
In Windows 11, ISRs are strictly controlled by:

- The **I/O Manager**
- The **Interrupt Object Manager**
- The **HAL interrupt abstraction layer**

ISRs never run in VTL1 and never execute pageable code.

12.2.2 Exact Architectural Role Inside the Windows 11 Kernel

The ISR is the **first software-level execution stage after hardware interrupt dispatch**. Its responsibilities are:

- Acknowledge and mask the hardware interrupt
- Capture minimal device state
- Queue a **Deferred Procedure Call (DPC)**
- Exit immediately using **IRETQ**

The ISR operates above **DISPATCH_LEVEL** and temporarily suspends:

- All thread scheduling
- APC delivery
- Pageable memory access

12.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KINTERRUPT

```
typedef struct _KINTERRUPT {
    ULONG Vector;
    UCHAR Irql;
    UCHAR SynchronizeIrql;
    KAFFINITY Affinity;
    PKINTERRUPT_ROUTINE ServiceRoutine;
    PVOID ServiceContext;
    KSPIN_LOCK SpinLock;
    LIST_ENTRY InterruptListEntry;
} KINTERRUPT, *PKINTERRUPT;
```

Interrupt Object Location

```
nt!_KINTERRUPT
```

KPCR Linkage

```
nt!_KPCR
  +0x0D8 InterruptObject
```

12.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Hardware to ISR Entry

1. Device asserts IRQ
2. IOAPIC maps IRQ to interrupt vector
3. CPU reads IDT entry
4. Ring transition to kernel stack
5. Assembly stub executes
6. ISR function pointer is invoked

Kernel ISR Dispatch Path

1. `KiInterruptDispatch`
2. `HalpInterruptDispatch`
3. Registered ISR is executed
4. DPC optionally queued
5. `IRETQ` restores execution

External x64 ISR Stub (NASM)

```
global HardwareISRStub

HardwareISRStub:
    sub     rsp, 88h
    mov     [rsp+20h], rax
    mov     [rsp+28h], rcx
    mov     [rsp+30h], rdx
    call    MyInterruptServiceRoutine
    mov     rax, [rsp+20h]
    mov     rcx, [rsp+28h]
    mov     rdx, [rsp+30h]
    add     rsp, 88h
    iretq
```

12.2.5 Secure Kernel / VBS / HVCI Interaction

- ISRs execute only in **VTL0**
- HVCI enforces code integrity of ISR code pages
- Secure Kernel validates interrupt return flow
- DMA remapping is enforced before ISR execution

Any attempt to execute unsigned ISR code immediately triggers a **bug check**.

12.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- ISR execution flushes CPU pipeline
- Kernel stack switching invalidates cache locality

- Cross-NUMA interrupts incur memory latency penalties
- Excessive ISR duration causes system-wide scheduling stalls
- E-Core ISRs execute slower than P-Core ISRs

12.2.7 REAL Practical Example (C / C++ / Assembly)

ISR Function (C)

```
BOOLEAN MyInterruptServiceRoutine(
    PKINTERRUPT Interrupt,
    PVOID ServiceContext
)
{
    UNREFERENCED_PARAMETER(Interrupt);
    UNREFERENCED_PARAMETER(ServiceContext);
    return TRUE;
}
```

ISR Registration (C)

```
IO_INTERRUPT_MESSAGE_INFO Info;
Info.Version = CONNECT_MESSAGE_BASED;
Info.MessageCount = 1;

IoConnectInterruptEx(&Info);
```

External Assembly Timing Measurement

```
global ReadTimestampCounter

ReadTimestampCounter:
    rdtsc
```

```
    shl     rdx, 32
    or      rax, rdx
    ret
```

12.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

List active interrupts:

```
!interrupts
```

Inspect ISR object:

```
dt nt!_KINTERRUPT <address>
```

Check current IRQL:

```
!irql
```

Trace ISR execution:

```
!stacks 2
```

12.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attack vectors:

- ISR pointer overwrite
- Fake interrupt injection
- Stack pivot via malformed ISR

Windows 11 mitigations:

- PatchGuard ISR verification
- HVCI code-signing enforcement
- Secure Kernel interrupt return validation

12.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- ISRs must never block or wait.
- No memory allocation is allowed inside ISR.
- No pageable memory access is permitted.
- All registers must be preserved precisely.
- ISR execution time must be sub-microsecond.
- Any ISR stall directly reduces system-wide throughput.

12.3 MSI & MSI-X

12.3.1 Precise Windows 11Specific Definition (Engineering-Level)

Message Signaled Interrupts (MSI) and **MSI-X** are PCIe-based interrupt signaling mechanisms used in Windows 11 where devices generate interrupts by performing a **memory write transaction** instead of asserting a physical interrupt line. The write targets a specific **APIC-mapped physical address**, causing the Local APIC to raise an interrupt vector.
In Windows 11, MSI/MSI-X is the **default interrupt mechanism** for all modern PCIe devices and fully replaces:

- Legacy INTx line-based interrupts
- Edge-triggered shared IRQs

MSI is limited to **32 vectors**, while MSI-X scales up to **2048 independent vectors per device**.

12.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Within Windows 11, MSI/MSI-X forms the **core interrupt delivery fabric** between PCIe devices and the kernel interrupt subsystem:

- Eliminates IRQ sharing entirely
- Enables full CPU-core affinity control
- Allows direct mapping of device queues to cores
- Enables zero-contention interrupt delivery
- Enables interrupt steering under hybrid CPU schedulers

All modern subsystems depend on MSI:

- NVMe
- USB xHCI
- GPU command submission
- Network queues (RSS)

12.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_IO_INTERRUPT_MESSAGE_INFO

```
typedef struct _IO_INTERRUPT_MESSAGE_INFO {
    ULONG Version;
    ULONG MessageCount;
    ULONG Reserved[2];
    IO_INTERRUPT_MESSAGE_INFO_ENTRY MessageInfo[1];
} IO_INTERRUPT_MESSAGE_INFO, *PIO_INTERRUPT_MESSAGE_INFO;
```

_IO_INTERRUPT_MESSAGE_INFO_ENTRY

```
typedef struct _IO_INTERRUPT_MESSAGE_INFO_ENTRY {
    PHYSICAL_ADDRESS MessageAddress;
    ULONG MessageData;
    ULONG Vector;
    KIRQL Irql;
    KAFFINITY Affinity;
} IO_INTERRUPT_MESSAGE_INFO_ENTRY;
```

_KINTERRUPT (Per Vector)

```
nt!_KINTERRUPT
```

Each MSI-X vector owns a dedicated `_KINTERRUPT` object.

12.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

MSI Interrupt Generation

1. Device performs PCIe memory write:
2. Target = APIC MSI physical address
3. Payload = interrupt vector + trigger mode
4. Local APIC asserts vector internally
5. CPU dispatches IDT entry
6. Kernel ISR executes at DIRQL

Kernel Dispatch Path

1. `HalpMsiInterrupt`
2. `KiInterruptDispatch`
3. `KINTERRUPT.ServiceRoutine`
4. Optional DPC queued

External Assembly Vector Entry Stub

```
global MsiVectorStub

MsiVectorStub:
    sub     rsp, 88h
    mov     [rsp+20h], rax
    mov     [rsp+28h], rcx
    mov     [rsp+30h], rdx
    call    MsiIsrHandler
    mov     rax, [rsp+20h]
    mov     rcx, [rsp+28h]
    mov     rdx, [rsp+30h]
    add     rsp, 88h
    iretq
```

12.3.5 Secure Kernel / VBS / HVCI Interaction

- MSI configuration space is protected by IOMMU
- VBS enforces interrupt DMA remapping
- HVCI validates MSI ISR code signatures
- Secure Kernel prevents interrupt vector spoofing

- Interrupt message addresses are locked post-boot

Any illegal MSI address write is blocked at the IOMMU layer.

12.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Zero IRQ sharing eliminates cache thrashing
- MSI allows per-core device queue binding
- NUMA-local MSI routing minimizes interconnect latency
- MSI-X enables perfect RSS scalability on NICs
- E-Core MSI routing reduces energy efficiency but increases latency

MSI-X enables full **lock-free interrupt parallelism**.

12.3.7 REAL Practical Example (C / C++ / Assembly)

MSI Registration (KMDF / WDM)

```
IO_INTERRUPT_MESSAGE_INFO Info = {0};

Info.Version = CONNECT_MESSAGE_BASED;
Info.MessageCount = 8;

NTSTATUS status = IoConnectInterruptEx(&Info);
```

MSI Handler (C)

```
BOOLEAN MsiIsrHandler(
    PKINTERRUPT Interrupt,
    PVOID Context
```

```
)
{
    UNREFERENCED_PARAMETER(Interrupt);
    UNREFERENCED_PARAMETER(Context);
    return TRUE;
}
```

Timestamp for MSI Latency (Assembly)

```
global ReadTsc

ReadTsc:
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    ret
```

12.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

List MSI vectors:

```
!interrupts
```

Dump MSI interrupt:

```
dt nt!_IO_INTERRUPT_MESSAGE_INFO <addr>
```

Check IRQL:

```
!irql
```

Trace interrupt routing:

```
!pci 3
```

12.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical attacks:

- MSI vector spoofing
- DMA-based interrupt forgery
- Fault injection through malformed MSI messages

Windows 11 defenses:

- VT-d DMA isolation
- Interrupt remapping tables
- PatchGuard ISR integrity enforcement
- Secure Kernel vector verification

12.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- MSI must always be preferred over legacy IRQs.
- Each MSI-X vector must have its own ISR/DPC.
- Cross-core MSI routing destroys cache locality.
- Never bind multiple device queues to one MSI vector.
- MSI misconfiguration causes silent device hangs.
- MSI interrupt storms can stall the scheduler.

Chapter 13

IRQL Rules That Every C/Assembly Developer Must Know

13.1 When Memory Access Is Forbidden

13.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **memory access is strictly constrained by the current IRQL (Interrupt Request Level)**. Any attempt to access memory that may incur a **page fault** while executing at an IRQL **greater than `APC_LEVEL`** constitutes a fatal kernel programming violation and results in:

- `IRQL_NOT_LESS_OR_EQUAL`
- `PAGE_FAULT_IN_NONPAGED_AREA`

Windows 11 forbids any pageable memory access at or above `DISPATCH_LEVEL`. This restriction is enforced by the Memory Manager, Trap Handler, and PatchGuard integrity mechanisms.

13.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The IRQL-based memory access restriction protects:

- Interrupt dispatch determinism
- Scheduler forward progress
- APC/DPC execution ordering
- Secure Kernel execution integrity
- DMA and IOMMU synchronization

At elevated IRQL:

- The **pager is disabled**
- **VAD resolution is forbidden**
- **Page-in operations are blocked**
- **Kernel APC delivery is disabled**

This ensures that interrupt service routines and DPCs **never block on memory I/O**.

13.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTHREAD (IRQL Tracking)

```
typedef struct _KTHREAD {
    UCHAR CurrentIrql;
    UCHAR WaitIrql;
    UCHAR PreviousMode;
    UCHAR NpxState;
    ...
} KTHREAD, *PKTHREAD;
```

_MMPFN (Page Fault Dependency)

```
typedef struct _MMPFN {
    ULONG Active;
    ULONG ReferenceCount;
    PVOID PteAddress;
    ...
} MMPFN, *PMMPFN;
```

Trap Frame Page Fault Context

```
nt!_KTRAP_FRAME
```

The trap frame determines whether a fault is recoverable at the current IRQL.

13.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Page Fault at PASSIVE_LEVEL

1. Instruction accesses a paged address
2. CPU raises #PF
3. `KiPageFault` executes
4. Memory Manager resolves VAD
5. Page is paged-in from disk
6. Execution resumes safely

Page Fault at DISPATCH_LEVEL or Higher

1. Instruction accesses pageable memory

2. CPU raises #PF
3. `KiPageFault` detects elevated IRQL
4. Fault cannot be serviced
5. Kernel issues bugcheck immediately

External Assembly Faulting Access (Demonstration)

```
global FaultingRead

FaultingRead:
    mov     rax, [rcx]      ; rcx points to pageable address
    ret
```

If this executes at `DISPATCH_LEVEL`, the system will bugcheck.

13.1.5 Secure Kernel / VBS / HVCI Interaction

- Secure Kernel executes in **VTL1**
- Page faults from VTL0 cannot propagate into VTL1
- DMA remapping prevents fault-based privilege escalation
- HVCI blocks execution of fault-generating shellcode
- PatchGuard validates page fault origin paths

Any attempt to trigger controlled faults at elevated IRQL under VBS results in immediate kernel termination.

13.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Page faults cause full TLB invalidation
- Cache lines are evicted on page transitions
- NUMA locality is destroyed on cross-node paging
- Scheduler stalls if faults occur in high-frequency DPC paths
- ISR/DPC execution must be 100% fault-free

Windows 11 aggressively migrates hot kernel paths into **nonpaged memory** to preserve deterministic interrupt latency.

13.1.7 REAL Practical Example (C / C++ / Assembly)

Illegal Pattern (Will Crash)

```
VOID DpcRoutine(PKDPC Dpc, PVOID Context, PVOID A1, PVOID A2)
{
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(A1);
    UNREFERENCED_PARAMETER(A2);

    volatile CHAR value = *((CHAR*)Context); // Context points to paged
    ↪  memory
}
```

Correct Pattern (NonPaged Pool)

```
PVOID buffer = ExAllocatePool2(POOL_FLAG_NON_PAGED, 64, 'IRQL');

VOID DpcRoutine(PKDPC Dpc, PVOID Context, PVOID A1, PVOID A2)
```

```
{
    volatile CHAR value = *((CHAR*)Context);
}
```

IRQL Read (Assembly)

```
global ReadCurrentIrql

ReadCurrentIrql:
    mov     al, gs:[0x1A4]    ; KPCR.CurrentIrql (symbolic)
    ret
```

13.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check current IRQL:

```
!irql
```

Detect pageable code at elevated IRQL:

```
!chkimg nt
```

Trace last page fault:

```
!analyze -v
```

Inspect faulting address:

```
r cr2
```

13.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Controlled page faults were historically abused for kernel ROP
- Fault spraying was used for stack pivot attacks
- DMA-based fault injection enabled write-what-where primitives
- Windows 11 blocks all of these using:
  - VBS
  - HVCI
  - Kernel DMA Protection
  - PatchGuard trap validation

Modern Windows 11 effectively eliminates fault-driven kernel exploitation.

13.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never touch pageable memory at `DISPATCH_LEVEL` or above.
- Never acquire pageable mutexes inside DPCs or ISRs.
- All ISR and DPC code must be in **NONPAGED_TEXT**.
- Never call Zw/Nt services at elevated IRQL.
- Never probe user memory above `PASSIVE_LEVEL`.
- Always validate IRQL using `KeGetCurrentIrql()`.
- One illegal memory touch at high IRQL equals a guaranteed system crash.

13.2 When Interrupts Are Forbidden

13.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **interrupts are forbidden whenever the kernel explicitly raises IRQL above the hardware interrupt delivery threshold or explicitly masks interrupts at the CPU level**. This state is enforced using:

- `IRQL >= HIGH_LEVEL`
- CPU interrupt flag (`RFLAGS.IF = 0`)
- Certain spinlock acquisition paths
- Secure Kernel execution barriers

When interrupts are forbidden, **no external hardware interrupt, IPI, or deferred interrupt may be delivered to the current logical processor**. Any attempt to violate this constraint results in:

- Global system deadlock
- Watchdog timeout (`CLOCK_WATCHDOG_TIMEOUT`)
- Immediate system crash under VBS

13.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Interrupt suppression is required to guarantee:

- Atomic modification of scheduler state
- Deterministic spinlock protection

- CPU-local interrupt controller programming
- Context switch register safety
- Secure Kernel transition integrity

Windows 11 uses **two independent suppression layers**:

1. **Logical suppression via IRQL** (scheduler-managed)
2. **Physical suppression via CPU IF flag** (hardware-managed)

Both must be correct simultaneously to preserve kernel forward progress.

13.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KPRCB (Interrupt Control)

```
typedef struct _KPRCB {
    UCHAR CurrentIrql;
    UCHAR PendingSoftwareInterrupts;
    ULONG InterruptCount;
    ...
} KPRCB, *PKPRCB;
```

_KINTERRUPT

```
typedef struct _KINTERRUPT {
    ULONG Vector;
    KIRQL Irql;
    PVOID ServiceRoutine;
    ...
} KINTERRUPT, *PKINTERRUPT;
```

_KPCR (CPU Interrupt State)

```
typedef struct _KPCR {
    PKPRCB Prcb;
    ...
} KPCR, *PKPCR;
```

13.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Interrupt Masking via IRQL

1. Kernel raises IRQL using `KeRaiseIrql`
2. Scheduler blocks delivery of lower-priority interrupts
3. Software interrupts are deferred
4. DPC/APC queues are frozen

Interrupt Masking via CPU Flags

1. Kernel executes `cli` instruction
2. CPU clears `RFLAGS.IF`
3. All maskable hardware interrupts are blocked
4. Only NMI and MCE remain deliverable

External Assembly Interrupt Masking

```
global DisableInterrupts
DisableInterrupts:
    cli
    ret
```

```
global EnableInterrupts
EnableInterrupts:
    sti
    ret
```

These routines are only legal inside tightly controlled kernel paths.

13.2.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 VBS:

- Secure Kernel executes in **VTL1**
- Interrupt routing is virtualized by the hypervisor
- Guest kernels cannot permanently disable interrupts
- Any prolonged interrupt suppression triggers hypervisor intervention
- HVCI blocks execution of unauthorized interrupt handlers

This prevents:

- Interrupt-based rootkits
- CPU stall attacks
- NMI-based privilege escalation

13.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Interrupt suppression blocks:
  - Timer ticks
  - I/O completion delivery
  - DPC execution
  - Scheduler load balancing
- Cache coherency traffic may accumulate
- TLB shootdowns are deferred
- NUMA cross-node IPIs are blocked
- Prolonged suppression causes system-wide latency collapse

Windows 11 enforces **strict time budgets** for any region executing with interrupts disabled.

13.2.7 REAL Practical Example (C / C++ / Assembly)

Illegal Pattern (Fatal Deadlock)

```
KIRQL oldIrql;
KeRaiseIrql(HIGH_LEVEL, &oldIrql);

/* Illegal: Wait operation while interrupts are disabled */
KeWaitForSingleObject(Event, Executive, KernelMode, FALSE, NULL);
```

Correct Atomic Pattern

```
KIRQL oldIrql;
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);
```

```
/* Safe atomic work */
InterlockedIncrement(&GlobalCounter);

KeLowerIrql(oldIrql);
```

Direct CPU Control (Assembly)

```
global AtomicSection
AtomicSection:
    cli
    inc     qword [rcx]
    sti
    ret
```

This must never span external memory access or OS calls.

13.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check interrupt state:

```
r rflags
```

Inspect IRQL:

```
!irql
```

Detect interrupt stalls:

```
!cpuinfo
```

Display pending interrupts:

```
!interrupts
```

13.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical abuse vectors:

- Interrupt suppression for scheduler starvation
- Fake ISR redirection for stealth rootkits
- CLI-based infinite lock attacks
- NMI-based execution pivots

Windows 11 mitigations:

- Hypervisor-enforced interrupt routing
- HVCI code integrity enforcement
- PatchGuard ISR validation
- Watchdog CPU stall detection

Modern Windows 11 makes persistent interrupt suppression non-viable for attackers.

13.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never hold interrupts disabled across function calls.
- Never disable interrupts while accessing pageable memory.
- Never disable interrupts while holding dispatcher objects.
- ISR code must never call blocking primitives.

- DPC code must execute with interrupts enabled.
- Secure Kernel transitions require interrupts enabled.
- Any interrupt-disabled region must be:
  - Constant time
  - Non-faulting
  - Non-blocking
- Violating these rules causes non-recoverable system failure.

13.3 Causes of the Most Common Crashes (`IRQL_NOT_LESS_OR_EQUAL`)

13.3.1 Precise Windows 11Specific Definition (Engineering-Level)

`IRQL_NOT_LESS_OR_EQUAL` (BugCheck `0xA`) is a **fatal kernel-mode violation raised by ntoskrnl.exe when code executing at an elevated IRQL attempts an illegal memory access that requires page fault handling**.
In Windows 11, this specifically means:

- The current **IRQL** $\geq$ **DISPATCH_LEVEL**
- The faulting access targets:
  - Pageable memory
  - User-mode virtual addresses
  - Invalid or freed kernel addresses

- The CPU triggers a page fault (`#PF`) that the kernel is **architecturally forbidden** to service at that IRQL

Windows 11 immediately invokes **`KeBugCheckEx(0xA)`** to prevent silent kernel corruption.

13.3.2 Exact Architectural Role Inside the Windows 11 Kernel

This bugcheck is a **hard architectural safeguard** protecting:

- Scheduler state integrity
- DPC execution correctness
- Spinlock atomicity
- Interrupt routing correctness
- Secure Kernel isolation

At `DISPATCH_LEVEL` and above:

- Page faults **cannot be resolved**
- Memory manager **cannot block**
- I/O completion **cannot be waited on**
- The thread **cannot be rescheduled**

Allowing a pageable access here would **instantly corrupt kernel execution ordering**.
Therefore Windows 11 traps and terminates.

13.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KTRAP_FRAME (Fault Context)

```
typedef struct _KTRAP_FRAME {
    ULONG64 Rip;
    ULONG64 Rsp;
    ULONG64 Cr2;      // Faulting virtual address
    ULONG64 ErrorCode;
    ...
} KTRAP_FRAME, *PKTRAP_FRAME;
```

_KPRCB (Current IRQL)

```
typedef struct _KPRCB {
    UCHAR CurrentIrql;
    ...
} KPRCB, *PKPRCB;
```

BugCheck Parameters (0xA)

- Param 1: Faulting virtual address
- Param 2: IRQL at time of access
- Param 3: Access type (read/write/execute)
- Param 4: Instruction pointer (RIP)

13.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

1. Driver or kernel routine raises IRQL to `DISPATCH_LEVEL` or higher
2. Code dereferences a pageable or invalid address

3. CPU raises `#PF` (Page Fault)
4. Memory manager detects illegal IRQL state
5. Kernel calls `KeBugCheckEx(IRQL_NOT_LESS_OR_EQUAL)`
6. System transitions to crash dump

Faulting Assembly Pattern

```
; RCX contains a user-mode pointer
mov     rax, [rcx]        ; Illegal at DISPATCH_LEVEL -> #PF -> BugCheck
↪  0xA
```

13.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- Secure Kernel validates:
  - IRQL state
  - Page ownership (VTL0 vs VTL1)
  - Code origin (HVCI-enforced)
- Any high-IRQL fault in:
  - Driver code
  - ISR
  - DPC

  cannot be intercepted, emulated, or corrected
- Hypervisor forbids guest page-ins at elevated IRQL

This makes `0xA` **non-recoverable by design** under HVCI.

13.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Elevated IRQL disables:
  - Scheduler preemption
  - DPC dispatching
  - Timer interrupts
- Any fault:
  - Stalls the entire logical processor
  - Blocks cross-core TLB shootdowns
  - Freezes NUMA IPIs
- Allowing retries would cause:
  - Cache line corruption
  - Partial TLB invalidation
  - Global scheduler desynchronization

Immediate crash is **architecturally cheaper than recovery**.

13.3.7 REAL Practical Example (C / C++ / Assembly)

Illegal Driver Code (Guaranteed 0xA)

```
KIRQL oldIrql;
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);

/* User pointer dereference  fatal */
volatile ULONG value = *(volatile ULONG*)UserPointer;

KeLowerIrql(oldIrql);
```

Correct Pattern Using Locked Kernel Memory

```
PMDL mdl = IoAllocateMdl(UserBuffer, Size, FALSE, FALSE, NULL);
MmProbeAndLockPages(mdl, KernelMode, IoReadAccess);
PVOID kaddr = MmGetSystemAddressForMdlSafe(mdl, NormalPagePriority);

/* Safe at DISPATCH_LEVEL */
volatile ULONG value = *(volatile ULONG*)kaddr;

MmUnlockPages(mdl);
IoFreeMdl(mdl);
```

Assembly Fault Example

```
; DISPATCH_LEVEL active
mov     rax, [gs:188h]    ; KPCR
mov     rcx, [rax+30h]    ; Current thread
mov     rdx, [rcx]        ; Unvalidated pointer -> #PF -> 0xA
```

13.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Identify the bugcheck:

```
!analyze -v
```

Check IRQL:

```
!irql
```

Display faulting instruction:

```
r rip
u @rip-20
```

Inspect the faulting address:

```
r cr2
!pte @cr2
```

Verify driver:

```
lmv m drivername
```

13.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically exploited for:

- Arbitrary kernel read/write via high-IRQL dereference
- ISR pointer substitution
- Fake DPC memory execution

Windows 11 protections:

- HVCI enforced code identity
- PatchGuard IRQL validation
- SMAP/SMEP enforcement
- Hypervisor-controlled page ownership

Modern exploitation of `0xA` is **not viable without hardware-level compromise**.

13.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never touch:
  - User pointers
  - Pageable memory
  - File buffers
  - Network buffers

  at `DISPATCH_LEVEL` or above.
- All DPC and ISR code must execute from **non-pageable memory**.
- Never assume a pointer remains resident after IRQL is raised.
- Pre-map all memory before raising IRQL.
- Validate all MDLs before DPC execution.
- Any IRQL violation is treated as a **kernel integrity breach**.

Chapter 14

APC & DPC from Inside the Kernel

14.1 The Real Difference Between APC and DPC

14.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **APC (Asynchronous Procedure Call)** and **DPC (Deferred Procedure Call)** are two fundamentally different kernel dispatch mechanisms designed to execute deferred work under **strictly different IRQL, scheduling, and memory rules**.

- **DPC**: A **CPU-local, interrupt-driven, high-IRQL execution mechanism** used to complete work deferred from an ISR at `DISPATCH_LEVEL`.
- **APC**: A **thread-context, scheduler-injected execution mechanism** delivered at `APC_LEVEL` or `PASSIVE_LEVEL`.

They are **not interchangeable**, not equivalent, and serve **orthogonal roles** inside ntoskrnl.exe.

14.1.2 Exact Architectural Role Inside the Windows 11 Kernel

DPC Architectural Role

- ISR defers non-critical interrupt work
- Runs at `DISPATCH_LEVEL`
- Executes on the **same logical processor** that received the interrupt
- Used by:
  - Network stack
  - Storage stack
  - Timer subsystem
  - Scheduler clock interrupts

APC Architectural Role

- Injects work into a **specific thread context**
- Used for:
  - User-mode async I/O completion
  - Thread termination
  - Alertable waits
  - Memory manager paging completion
- Can execute at:
  - `APC_LEVEL` (Kernel APC)
  - `PASSIVE_LEVEL` (User APC)

14.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KDPC

```
typedef struct _KDPC {
    UCHAR Type;
    UCHAR Importance;
    USHORT Number;
    LIST_ENTRY DpcListEntry;
    PKDEFERRED_ROUTINE DeferredRoutine;
    PVOID DeferredContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    PKPRCB TargetProcessor;
} KDPC, *PKDPC;
```

_KAPC

```
typedef struct _KAPC {
    UCHAR Type;
    UCHAR SpareByte0;
    UCHAR Size;
    UCHAR SpareByte1;
    ULONG SpareLong0;
    PKTHREAD Thread;
    LIST_ENTRY ApcListEntry;
    PKKERNEL_ROUTINE KernelRoutine;
    PKRUNDOWN_ROUTINE RundownRoutine;
    PKNORMAL_ROUTINE NormalRoutine;
    PVOID NormalContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
} KAPC, *PKAPC;
```

_KTHREAD APC State

```
typedef struct _KAPC_STATE {
    LIST_ENTRY ApcListHead[2];
    struct _KPROCESS *Process;
    BOOLEAN KernelApcInProgress;
    BOOLEAN KernelApcPending;
    BOOLEAN UserApcPending;
} KAPC_STATE, *PKAPC_STATE;
```

14.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

DPC Flow

1. Hardware interrupt arrives
2. CPU vectors into ISR via IDT
3. ISR queues a DPC using `KeInsertQueueDpc`
4. Kernel clock tick triggers DPC drain
5. DPC executes at `DISPATCH_LEVEL`

APC Flow

1. Kernel queues APC using `KeInsertQueueApc`
2. Target thread enters alertable or kernel transition
3. Dispatcher delivers APC at safe IRQL
4. APC runs in **thread context**

Assembly-Level Difference

```
; DPC context: no thread stack, no blocking allowed
; APC context: thread stack active, blocking allowed (PASSIVE only)
```

14.1.5 Secure Kernel / VBS / HVCI Interaction

- DPCs execute entirely in **VTL0 kernel context**
- APC delivery is **validated against thread ownership**
- HVCI enforces:
  - Valid code origins for DPC/APC routines
  - No user-writable executable pages
- Secure Kernel blocks any attempt to:
  - Redirect APC routines
  - Hijack DPC tables

APC/DPC execution paths are **fully protected by hypervisor-enforced integrity**.

14.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

DPC Performance

- Executes with:
  - Scheduler disabled
  - Preemption blocked
  - Spinlocks active

- Excessive DPC time causes:
  - `DPC_WATCHDOG_VIOLATION`
  - I/O starvation
  - Network collapse

APC Performance

- Executes inside thread scheduling budget
- Can fault, block, allocate memory
- Poor APC design leads to:
  - Thread starvation
  - User-mode latency
  - I/O completion stalls

NUMA locality:

- DPC: fixed to CPU node
- APC: follows thread migration

14.1.7 REAL Practical Example (C / C++ / Assembly)

DPC Example (KMDF)

```
KDPC DpcObject;

VOID DpcRoutine(
    PKDPC Dpc,
```

```
    PVOID Context,
    PVOID Arg1,
    PVOID Arg2
) {
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(Context);
}

KeInitializeDpc(&DpcObject, DpcRoutine, NULL);
KeInsertQueueDpc(&DpcObject, NULL, NULL);
```

APC Example (Kernel APC)

```
KAPC Apc;

KeInitializeApc(
    &Apc,
    TargetThread,
    OriginalApcEnvironment,
    KernelApcRoutine,
    NULL,
    NULL,
    KernelMode,
    NULL
);

KeInsertQueueApc(&Apc, NULL, NULL, 0);
```

External x64 Assembly APC Stub

```
PUBLIC ApcAsmStub

ApcAsmStub PROC
    sub     rsp, 20h
```

```
    ; APC-safe execution
    add     rsp, 20h
    ret
ApcAsmStub ENDP
```

14.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect queued DPCs:

```
!dpcs
```

Inspect APC state of a thread:

```
!thread
```

Dump APC list:

```
dt nt!_KAPC_STATE
```

Check IRQL:

```
!irql
```

14.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Attacks

- DPC callback pointer overwrite
- APC injection into system threads
- Fake APC normal routines

Windows 11 Protections

- HVCI-enforced function pointer validation
- PatchGuard DPC/APC integrity checks
- SMEP/SMAP enforcement
- VTL isolation

Modern exploitation through APC/DPC is **effectively blocked** without hardware-level compromise.

14.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never allocate memory in DPC unless using non-pageable pools.
- Never block, wait, or access user memory in DPC.
- APCs may page fault only at `PASSIVE_LEVEL`.
- Misusing APC for ISR completion is **architectural abuse**.
- DPCs must always execute in bounded time.
- Treat APC as a **controlled thread injection mechanism**.
- Treat DPC as a **hard-real-time interrupt continuation**.

14.2 Executing DPC in Assembly

14.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, executing a **Deferred Procedure Call (DPC)** in assembly refers to the execution of a kernel-registered `PKDEFERRED_ROUTINE` function at `DISPATCH_LEVEL` using a routine whose implementation is written in **external x86-64 assembly**, fully compliant with the **Windows x64 ABI** and enforced kernel execution constraints.
A DPC assembly routine:

- Executes at `DISPATCH_LEVEL`
- Has **no valid thread context**
- Uses the **current CPU kernel stack**
- Cannot page fault
- Cannot block, sleep, or wait
- Cannot touch pageable memory

14.2.2 Exact Architectural Role Inside the Windows 11 Kernel

DPC assembly execution occurs in the following architectural context:

- Triggered by `KiRetireDpcList`
- Executed from:
  - Clock interrupt retirement path
  - ISR deferred completion path

- Bound to:
  - Local `KPRCB`
  - Local per-CPU DPC queue

The execution pipeline:

1. ISR queues a DPC
2. Scheduler tick arrives
3. `KiExecuteAllDpcs` is invoked
4. `DeferredRoutine` pointer is called directly
5. Assembly routine executes at `IRQL = DISPATCH_LEVEL`

14.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KDPC Structure

```
typedef struct _KDPC {
    UCHAR Type;
    UCHAR Importance;
    USHORT Number;
    LIST_ENTRY DpcListEntry;
    PKDEFERRED_ROUTINE DeferredRoutine;
    PVOID DeferredContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    PKPRCB TargetProcessor;
} KDPC, *PKDPC;
```

_KPRCB DPC Fields

```
typedef struct _KPRCB {
    KDPC_DATA DpcData[2];
    ULONG DpcQueueDepth;
    ULONG MaximumDpcQueueDepth;
    ULONG DpcRequestRate;
} KPRCB, *PKPRCB;
```

14.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Queueing the DPC (C-Level)

```
KDPC DpcObject;

VOID DpcRoutineAsm(
    PKDPC Dpc,
    PVOID Context,
    PVOID Arg1,
    PVOID Arg2
);

KeInitializeDpc(&DpcObject, DpcRoutineAsm, NULL);
KeSetTargetProcessorDpc(&DpcObject, 0);
KeInsertQueueDpc(&DpcObject, NULL, NULL);
```

Kernel Dispatcher Execution

1. `KiRetireDpcList`
2. `KiExecuteDpc`
3. `call qword ptr [KDPC->DeferredRoutine]`

Assembly Entry Context

- `RCX = PKDPC`
- `RDX = DeferredContext`
- `R8 = SystemArgument1`
- `R9 = SystemArgument2`
- Shadow space already allocated
- `RSP` is 16-byte aligned

14.2.5 Secure Kernel / VBS / HVCI Interaction

- DPC execution occurs strictly in:
  - **VTL0 Kernel Mode**
  - Non-virtualized execution lane
- HVCI enforces:
  - DPC entry must point to valid executable kernel memory
  - No writable executable DPC routines
- PatchGuard validates:
  - KDPC table integrity
  - DeferredRoutine pointer consistency

Any attempt to redirect an assembly DPC routine dynamically is blocked by:

- SMEP

- HVCI
- Kernel CFG

14.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Execution Cost

- No context switch
- No scheduler preemption
- Runs inside interrupt tail
- Direct impact on:
  - Interrupt latency
  - Network throughput
  - Storage completion latency

NUMA Behavior

- DPC runs on:
  - Local NUMA node CPU
  - Local memory hot cache
- Cross-node memory access causes:
  - TLB misses
  - QPI/IF fabric stalls

14.2.7 REAL Practical Example (C / C++ / Assembly)

External x64 Assembly DPC Routine

```
PUBLIC DpcAsmRoutine

.code

DpcAsmRoutine PROC
    sub     rsp, 28h                ; Shadow + alignment

    ; RCX = PKDPC
    ; RDX = Context
    ; R8  = Arg1
    ; R9  = Arg2

    mov     rax, rdx                ; Example: copy context
    mov     qword ptr [rax], 1      ; Write flag (non-pageable memory only)

    add     rsp, 28h
    ret
DpcAsmRoutine ENDP

END
```

Kernel C Registration

```
extern VOID DpcAsmRoutine(
    PKDPC Dpc,
    PVOID Context,
    PVOID Arg1,
    PVOID Arg2
);
```

```
KeInitializeDpc(&DpcObject, DpcAsmRoutine, SharedNonPagedFlag);
KeInsertQueueDpc(&DpcObject, NULL, NULL);
```

Visual Studio 2022 Build Rules

- Assembly compiled as:
  - `ml64.exe`
  - Not inline
  - Linked into KMDF driver
- Driver compiled with:
  - WDK + MSVC
  - `/kernel`
  - `/Qspectre`
  - `/guard:cf`

14.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect live DPCs:

```
!dpcs
```

Inspect KDPC object:

```
dt nt!_KDPC <address>
```

Inspect PRCB DPC state:

```
dt nt!_KPRCB @prcb
```

Check current IRQL during execution:

```
!irql
```

Set breakpoint on DPC execution:

```
bp DpcAsmRoutine
```

14.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Attack Vectors

- KDPC pointer overwrite
- DPC queue corruption
- ISR-to-DPC redirection

Windows 11 Mitigations

- HVCI executable validation
- PatchGuard KDPC table validation
- SMEP/SMAP enforcement
- Kernel CFG indirect call protection

Modern assembly DPC exploitation without a signed kernel driver is no longer viable.

14.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never touch pageable memory inside an assembly DPC.
- Never call:
  - `KeWaitForSingleObject`
  - `ExAllocatePoolWithTag(Paged)`
  - Any blocking primitive
- Never exceed a few microseconds of execution time.
- Never perform floating-point operations (FPU state not guaranteed).
- Always assume:
  - Preemption disabled
  - Interrupts enabled
  - Scheduler locked
- Assembly DPCs must be:
  - Deterministic
  - Lock-free
  - Non-faulting

14.3 Common Deadlocks

14.3.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, a **kernel deadlock** is a non-progress condition where execution halts because two or more execution contexts (threads, ISRs, APCs, or DPCs) are each waiting on synchronization resources held by the others, while no preemption, IRQL lowering, or dependency break is possible. At elevated IRQLs (`DISPATCH_LEVEL` and above), deadlocks become **architecturally unrecoverable** and result in system hangs or bug checks.
Deadlocks involving:

- Spin locks
- Executive resources
- ERESOURCE pushlocks
- Dispatcher objects

are particularly destructive when mixed across **Thread, APC, DPC, and ISR contexts**.

14.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Deadlocks arise from incorrect synchronization ordering across the following architectural layers:

- **Thread Context** (PASSIVE_LEVEL)
- **APC Context** (APC_LEVEL)
- **DPC Context** (DISPATCH_LEVEL)
- **ISR Context** (DIRQL)

Architecturally, Windows 11 enforces:

- Strict IRQL-based execution barriers
- Non-blocking execution above PASSIVE_LEVEL
- Per-CPU spinlock ownership via KPRCB

A violation creates a circular wait where:

1. A high-IRQL context owns a spinlock
2. A lower-IRQL context holds a blocking object
3. Mutual acquisition becomes impossible

14.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_KSPIN_LOCK

```
typedef ULONG_PTR KSPIN_LOCK;
```

_KTHREAD Synchronization Fields

```
typedef struct _KTHREAD {
    ULONG CombinedApcDisable;
    KSPIN_LOCK ThreadLock;
    UCHAR WaitIrql;
    UCHAR State;
} KTHREAD, *PKTHREAD;
```

_KPRCB Lock Ownership

```
typedef struct _KPRCB {
    PKSPIN_LOCK LockQueue[16];
    ULONG KernelTime;
    ULONG DpcQueueDepth;
} KPRCB, *PKPRCB;
```

14.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Fatal Thread-to-DPC Deadlock Sequence

1. Thread acquires spinlock at `PASSIVE_LEVEL`
2. Thread queues a DPC
3. Thread blocks waiting for an event
4. DPC fires at `DISPATCH_LEVEL`
5. DPC attempts to acquire the same spinlock
6. Deadlock occurs (thread blocked, DPC spinning)

Fatal APC-to-DPC Deadlock

1. APC acquires executive resource
2. APC is preempted by DPC
3. DPC attempts same resource
4. DPC cannot block infinite spin system hang

Assembly-Level IRQL Deadlock Condition

```
; Executing at DISPATCH_LEVEL
lock xchg [SpinLock], eax
cmp eax, 0
jne DeadlockSpin    ; Infinite loop if lower thread holds the lock
```

14.3.5 Secure Kernel / VBS / HVCI Interaction

- Deadlocks occur exclusively inside **VTL0**
- Secure Kernel (VTL1):
  - Isolated from spinlock ownership
  - Cannot be deadlocked by VTL0 drivers
- HVCI:
  - Prevents dynamic patching of deadlocked code paths
  - Enforces control-flow integrity even during hangs
- Watchdog timers:
  - Detect ISR/DPC stalls
  - Trigger `CLOCK_WATCHDOG_TIMEOUT` bugcheck

14.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Spin Deadlocks

- Cache line thrashing from locked memory operations
- Front-end pipeline starvation
- Total CPU core livelock

NUMA Impact

- Cross-node locked memory:
  - Interconnect saturation
  - Remote cache coherency storms

Scheduler Impact

- IRQL never lowers
- Ready threads never dispatch
- Watchdog resets entire machine

14.3.7 REAL Practical Example (C / C++ / Assembly)

Fatal Spinlock Deadlock (C Driver)

```
KSPIN_LOCK Lock;
KEVENT Event;

VOID DangerousDpc(
    PKDPC Dpc,
    PVOID Context,
    PVOID Arg1,
    PVOID Arg2
)
{
    KeAcquireSpinLockAtDpcLevel(&Lock);    // DEADLOCK
}
```

```
KeAcquireSpinLock(&Lock, &OldIrql);
KeInitializeEvent(&Event, NotificationEvent, FALSE);
```

```
KeInsertQueueDpc(&DpcObject, NULL, NULL);
KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, NULL);
// DEADLOCK: Thread waits, DPC spins
```

Assembly-Level Deadlock Loop

```
PUBLIC DpcAsmDeadlock

.code
DpcAsmDeadlock PROC
DeadlockLoop:
    mov eax, 1
    lock xchg [rcx], eax
    test eax, eax
    jne DeadlockLoop
    ret
DpcAsmDeadlock ENDP
END
```

Visual Studio 2022 Build Constraint

- External assembly only via `ml64.exe`
- Must be linked into KMDF driver
- No inline assembly supported on x64

14.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Detect spin deadlocks:

```
!locks
```

Inspect blocked threads:

```
!thread 0 1
```

Inspect owning KTHREAD:

```
dt nt!_KTHREAD <address>
```

Check IRQL on each CPU:

```
!irql
```

Inspect DPC queues:

```
!dpcs
```

Watchdog crash analysis:

```
!analyze -v
```

14.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Exploitation

- Forced deadlock to bypass PatchGuard timing
- Spinlock corruption to force write-what-where primitives

Windows 11 Protections

- Kernel CFG prevents lock hijacking
- HVCI enforces immutable executable memory
- Watchdog timers detect stalls
- Secure Kernel isolation prevents VTL0 deadlocks from escaping

Deadlocks are now denial-of-service only, not privilege escalation primitives.

14.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never acquire a spinlock at PASSIVE_LEVEL and wait on an object.
- Never acquire a dispatcher object above PASSIVE_LEVEL.
- Never acquire the same lock in:
  - Thread DPC
  - APC DPC
- Never nest spinlocks without strict global ordering.
- Never access pageable memory inside:
  - APC
  - DPC
  - ISR
- Always validate IRQL before:
  - Memory access
  - Lock acquisition
  - Resource ownership
- Most IRQL-related deadlocks end in:
  - `IRQL_NOT_LESS_OR_EQUAL`
  - `CLOCK_WATCHDOG_TIMEOUT`
  - `DPC_WATCHDOG_VIOLATION`

Part VIII

Object Manager & Handle Internals

Chapter 15

Windows Object Model for Engineers

15.1 Object Headers

15.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, an **object header** is a kernel-resident metadata structure that immediately precedes every executive object in memory. It is implemented as the `_OBJECT_HEADER` structure and provides unified services for:

- Reference counting
- Handle counting
- Security descriptor attachment
- Object type binding
- Quota tracking
- Name management

Every kernel object exposed through the Object Manager (`nt!Ob`) is addressed internally as:

```
Object = (PUCHAR)Header + sizeof(_OBJECT_HEADER)
```

This model is invariant across Windows 11 releases and is enforced by `ntoskrnl.exe`.

15.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The object header is the **core enforcement boundary** between:

- Object Manager (Ob)
- Security Reference Monitor (Se)
- Memory Manager (Mm)
- Handle Table Manager (Exp)
- Kernel Dispatcher (Ke)

Architecturally, it enables:

- Uniform object lifetime control
- Cross-subsystem object visibility
- Secure handle translation
- Kernel-mode and user-mode object isolation

All system calls that manipulate objects (e.g., `NtCreateFile`, `NtOpenProcess`, `NtDuplicateObject`) ultimately resolve and validate `_OBJECT_HEADER`.

15.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_OBJECT_HEADER (Windows 11 x64)

```
typedef struct _OBJECT_HEADER {
    LONGLONG PointerCount;
    union {
        LONGLONG HandleCount;
        PVOID NextToFree;
    };
    EX_PUSH_LOCK Lock;
    UCHAR TypeIndex;
    union {
        UCHAR TraceFlags;
        struct {
            UCHAR DbgRefTrace : 1;
            UCHAR DbgTracePermanent : 1;
            UCHAR Reserved : 6;
        };
    };
    UCHAR InfoMask;
    UCHAR Flags;
} OBJECT_HEADER, *POBJECT_HEADER;
```

_OBJECT_TYPE (Selected Fields)

```
typedef struct _OBJECT_TYPE {
    LIST_ENTRY TypeList;
    UNICODE_STRING Name;
    PVOID DefaultObject;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
```

```
    OBJECT_TYPE_INITIALIZER TypeInfo;
    EX_PUSH_LOCK TypeLock;
} OBJECT_TYPE, *POBJECT_TYPE;
```

Header Optional Extensions via InfoMask

- `OBJECT_HEADER_NAME_INFO`
- `OBJECT_HEADER_HANDLE_INFO`
- `OBJECT_HEADER_CREATOR_INFO`
- `OBJECT_HEADER_QUOTA_INFO`

These are conditionally appended *before* `_OBJECT_HEADER` based on `InfoMask` bits.

15.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Object Creation Path

1. User-mode calls `NtCreateXxx`
2. Transition via `syscall`
3. `ObCreateObjectEx` allocates object body
4. `ObpAllocateObject` prefixes `_OBJECT_HEADER`
5. Security descriptor attached
6. Handle table entry created
7. PointerCount and HandleCount initialized

Handle Resolution Path

1. Handle passed from user-mode
2. ExpHandleTable lookup
3. Object pointer resolved
4. Header validated
5. Access mask verified

Assembly-Level Pointer Arithmetic

```
; RCX = object body address
sub rcx, sizeof _OBJECT_HEADER
; RCX now points to OBJECT_HEADER
```

15.1.5 Secure Kernel / VBS / HVCI Interaction

- Object headers live exclusively in **VTL0**
- VTL1 Secure Kernel:
  - Cannot directly access VTL0 object headers
  - Operates only on isolated Secure Objects
- HVCI:
  - Prevents runtime patching of header validation paths
  - Enforces read-only executable mappings of Ob routines
- PatchGuard:

  - Monitors `_OBJECT_TYPE` and Ob dispatch tables
  - Detects header tampering

15.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Object headers are heavily cache-resident
- PointerCount and HandleCount are cache-line contention points
- High handle churn causes:
  - L3 cache pressure
  - Cross-core atomic invalidations
- NUMA:
  - Header allocation is node-local
  - Remote handle resolution increases latency

15.1.7 REAL Practical Example (C / C++ / Assembly)

C Driver: Resolving Object Header

```
POBJECT_HEADER Header;

Header = (POBJECT_HEADER)((PUCHAR)FileObject - sizeof(OBJECT_HEADER));
```

Inspecting Header Fields

```
DbgPrint("PointerCount: %lld\n", Header->PointerCount);
DbgPrint("HandleCount: %lld\n", Header->HandleCount);
DbgPrint("TypeIndex  : %u\n", Header->TypeIndex);
```

External Assembly (ml64)

```
PUBLIC GetObjectHeader

.code
GetObjectHeader PROC
    ; RCX = object body
    sub rcx, 30h        ; sizeof _OBJECT_HEADER on x64
    mov rax, rcx
    ret
GetObjectHeader ENDP
END
```

- Built using `ml64.exe`
- Linked into KMDF driver
- No inline assembly is permitted on x64 in Visual Studio 2022

15.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect an object header manually:

```
dt nt!_OBJECT_HEADER <ObjectAddress-30h>
```

Dump object type table:

```
!object \\
```

Inspect handle table:

```
!handle 0 1
```

Validate pointer and handle counts:

```
!obtrace
```

15.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Attacks

- TypeIndex confusion
- Header flag manipulation
- PointerCount overflows

Windows 11 Mitigations

- HVCI blocks header execution redirection
- CFG protects Ob dispatch
- PatchGuard detects corruption
- SMAP/SMEP enforce supervisor isolation

Direct header corruption now reliably results in bug checks rather than privilege escalation.

15.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never assume object layout stability without symbol validation.
- Never modify header fields outside Ob APIs.
- Do not cache object header pointers across:
  - Handle closure
  - Object dereference

- Never touch `InfoMask` manually.
- Always use:
  - `ObReferenceObject`
  - `ObDereferenceObject`
- Handle leaks always manifest as:
  - Increasing HandleCount
  - Non-zero PointerCount at driver unload
- Header corruption guarantees:
  - `KMODE_EXCEPTION_NOT_HANDLED`
  - `SYSTEM_SERVICE_EXCEPTION`
  - `ATTEMPTED_WRITE_TO_READONLY_MEMORY`

15.2 Object Types

15.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, an **object type** is a kernel-resident executive descriptor that defines the behavioral contract, security policy, access semantics, and lifecycle management rules for a specific class of kernel objects. Object types are represented by the `_OBJECT_TYPE` structure and are globally registered inside the Object Manager (`nt!Ob`) during kernel initialization. Every executive object instance is bound to exactly one `_OBJECT_TYPE`, which governs:

- Valid access masks
- Object-specific callbacks

- Pool allocation policy
- Security enforcement
- Close and delete behavior

Windows 11 enforces object type isolation as a fundamental kernel security boundary.

15.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Object types serve as the **formal interface contract** between:

- The Object Manager (`Ob`)
- The Security Reference Monitor (`Se`)
- The I/O Manager (`Io`)
- The Process Manager (`Ps`)
- The Configuration Manager (`Cm`)
- The Memory Manager (`Mm`)

Architecturally, object types enable:

- Uniform access validation
- Centralized lifetime control
- Dynamic dispatch of object-specific methods
- Strict separation between kernel object categories

All handle-based operations (`NtCreateXxx`, `NtOpenXxx`, `NtClose`) are validated against the owning `_OBJECT_TYPE`.

15.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_OBJECT_TYPE (Windows 11 x64)

```
typedef struct _OBJECT_TYPE {
    LIST_ENTRY TypeList;
    UNICODE_STRING Name;
    PVOID DefaultObject;
    ULONG Index;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
    OBJECT_TYPE_INITIALIZER TypeInfo;
    EX_PUSH_LOCK TypeLock;
    ULONG Key;
    LIST_ENTRY CallbackList;
} OBJECT_TYPE, *POBJECT_TYPE;
```

_OBJECT_TYPE_INITIALIZER

```
typedef struct _OBJECT_TYPE_INITIALIZER {
    USHORT Length;
    BOOLEAN UseDefaultObject;
    BOOLEAN CaseInsensitive;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    BOOLEAN SecurityRequired;
    BOOLEAN MaintainHandleCount;
    BOOLEAN MaintainTypeList;
    POOL_TYPE PoolType;
    ULONG DefaultPagedPoolCharge;
    ULONG DefaultNonPagedPoolCharge;
```

```
    OB_DUMP_METHOD DumpProcedure;
    OB_OPEN_METHOD OpenProcedure;
    OB_CLOSE_METHOD CloseProcedure;
    OB_DELETE_METHOD DeleteProcedure;
    OB_PARSE_METHOD ParseProcedure;
    OB_SECURITY_METHOD SecurityProcedure;
    OB_QUERYNAME_METHOD QueryNameProcedure;
    OB_OKAYTOCLOSE_METHOD OkayToCloseProcedure;
} OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;
```

Examples of Core Windows 11 Object Types

- `Process`
- `Thread`
- `File`
- `Key`
- `Event`
- `Semaphore`
- `Mutant`
- `Section`
- `Token`
- `Driver`
- `Device`

Each of these has a statically registered `_OBJECT_TYPE` inside `ntoskrnl.exe`.

15.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Object Type Registration at Boot

1. Kernel initializes Object Manager
2. `ObCreateObjectType` invoked for each core type
3. `OBJECT_TYPE` inserted into `ObTypeList`
4. Type index assigned
5. Global dispatch table updated

Object Creation Using a Type

1. Caller supplies type pointer to `ObCreateObjectEx`
2. Memory allocated using type's pool policy
3. `_OBJECT_HEADER` initialized
4. Type callbacks attached
5. Object inserted into global type list

Assembly-Level Type Resolution

```
; RCX = OBJECT_HEADER address
movzx eax, byte ptr [rcx + OBJECT_HEADER.TypeIndex]
; EAX now holds the object type index
```

Type index is later resolved into `OBJECT_TYPE*` using the global type table.

15.2.5 Secure Kernel / VBS / HVCI Interaction

- All object type metadata resides in VTL0
- Secure Kernel (VTL1):
  - Cannot directly register object types
  - Uses isolated Secure Kernel objects only
- HVCI:
  - Prevents dynamic patching of type callbacks
  - Enforces read-only dispatch of Ob methods
- PatchGuard:
  - Monitors `OBJECT_TYPE` integrity
  - Detects modification of callback tables

Object type corruption triggers immediate bug checks under Windows 11 security enforcement.

15.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Object type metadata is heavily shared across all cores
- High-frequency handle operations:
  - Cause cache-line contention on `TotalNumberOfHandles`
- NUMA:
  - Type structures are globally shared across nodes

  - Per-node object instances still obey type rules

- Scheduler:

  - Type callbacks execute under caller's scheduling context

Improper type design in custom drivers directly reduces system-wide scalability.

15.2.7 REAL Practical Example (C / C++ / Assembly)

KMDF Driver Referencing an Object Type

```
POBJECT_TYPE* PsProcessType;

PsProcessType = (POBJECT_TYPE*)MmGetSystemRoutineAddress(
    &RTL_CONSTANT_STRING(L"PsProcessType")
);
```

Validating an Object Against a Type

```
if (Header->TypeIndex != (*PsProcessType)->Index) {
    return STATUS_INVALID_PARAMETER;
}
```

External Assembly Type Index Extractor (ml64)

```
PUBLIC GetTypeIndex

.code
GetTypeIndex PROC
    ; RCX = OBJECT_HEADER
    movzx eax, byte ptr [rcx + 18h] ; TypeIndex offset
    ret
GetTypeIndex ENDP
```

```
END
```

Linked using `ml64.exe` and integrated into a KMDF driver build. Inline x64 assembly is not supported by Visual Studio 2022.

15.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump all registered object types:

```
!object \
```

Dump a specific type by name:

```
!object \File
```

Inspect the internal `OBJECT_TYPE` structure:

```
dt nt!_OBJECT_TYPE <address>
```

Inspect type callback table:

```
dt nt!_OBJECT_TYPE_INITIALIZER <address>
```

15.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Exploits

- TypeIndex confusion
- Callback hijacking
- Invalid access mask manipulation

Windows 11 Mitigations

- HVCI enforces executable permissions on callback dispatch
- CFG protects indirect calls
- PatchGuard verifies:
  - Callback pointers
  - Type lists
- SMAP/SMEP block cross-mode injection

Object type corruption reliably causes controlled bug checks instead of silent escalation.

15.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never cache `OBJECT_TYPE*` without re-validation.
- Never modify type callback pointers.
- Always respect:
  - ValidAccessMask
  - GenericMapping
- Custom object types in drivers must:
  - Use NonPagedPool
  - Define DeleteProcedure correctly
  - Maintain HandleCount if exported

- Invalid type usage causes:
  - `SYSTEM_SERVICE_EXCEPTION`
  - `KMODE_EXCEPTION_NOT_HANDLED`
  - `REFERENCE_BY_POINTER`
- Type misuse is one of the most common sources of kernel memory corruption.

15.3 Reference Counting

15.3.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **reference counting** is the fundamental kernel mechanism used by the Object Manager to track the lifetime of every executive object instance. Each object maintains two independent counters inside its `_OBJECT_HEADER`:

- **PointerCount**: Number of active kernel-mode references
- **HandleCount**: Number of active user-mode handles

An object is **eligible for destruction** only when:

$$\texttt{PointerCount} = 0 \quad \textbf{and} \quad \texttt{HandleCount} = 0$$

This dual-counter model is strictly enforced by Windows 11 to guarantee:

- Use-after-free prevention
- Cross-mode lifetime correctness
- Deterministic object destruction

15.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Reference counting is the **core synchronization contract** between:

- Object Manager (`Ob`)
- I/O Manager (`Io`)
- Process Manager (`Ps`)
- Security Subsystem (`Se`)
- Memory Manager (`Mm`)

Architecturally, it enforces:

- Safe sharing of kernel objects across:
  - Threads
  - Processes
  - Drivers
- Delayed object destruction after last user-mode close
- Guaranteed callback execution order:
  - CloseProcedure
  - DeleteProcedure

Without reference counting, Windows 11 cannot guarantee kernel stability under concurrent access.

15.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_OBJECT_HEADER (Windows 11 x64 Core Fields)

```
typedef struct _OBJECT_HEADER {
    LONG_PTR PointerCount;
    union {
        LONG_PTR HandleCount;
        PVOID NextToFree;
    };
    PEX_PUSH_LOCK Lock;
    UCHAR TypeIndex;
    UCHAR TraceFlags;
    UCHAR InfoMask;
    UCHAR Flags;
} OBJECT_HEADER, *POBJECT_HEADER;
```

Internal Reference APIs (Real)

- `ObReferenceObject`
- `ObReferenceObjectByHandle`
- `ObDereferenceObject`
- `ObfReferenceObject`
- `ObfDereferenceObject`

These functions manipulate `PointerCount` using atomic operations.

15.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Object Creation

1. Object allocated by `ObCreateObject`
2. `PointerCount` initialized to 1
3. `HandleCount` initialized to 0
4. Object inserted into type list

Handle Creation

1. User calls `NtCreateXxx`
2. `HandleCount++`
3. Handle table entry created

Kernel Reference Acquisition

1. Kernel calls `ObReferenceObject`
2. Atomic `PointerCount++`
3. Object becomes non-destroyable

Final Destruction Path

1. Last handle closed:
    - `HandleCount == 0`
2. Last kernel reference released:
    - `PointerCount == 0`

3. `Type->DeleteProcedure` invoked
4. Memory freed by pool allocator

Assembly-Level Atomic Reference Update

```
; RCX = OBJECT_HEADER*
lock inc qword ptr [rcx]       ; PointerCount++
```

```
; RCX = OBJECT_HEADER*
lock dec qword ptr [rcx]       ; PointerCount--
```

All reference updates are **fully atomic on Windows 11 x64**.

15.3.5 Secure Kernel / VBS / HVCI Interaction

- Reference counters exist exclusively in VTL0
- Secure Kernel (VTL1):
  - Cannot directly manipulate VTL0 object reference counts
- HVCI:
  - Prevents reference API patching
  - Enforces CFG on dereference callbacks
- PatchGuard:
  - Monitors object header integrity
  - Detects:
    - Counter manipulation bypass
    - Type callback corruption

Any illegal manipulation of object counters causes immediate bug check.

15.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- **Cache contention**:
  - PointerCount is frequently modified on shared objects
- **NUMA effects**:
  - Remote node reference increments incur cache line migration
- **Scheduler interaction**:
  - Atomic operations stall pipeline on heavy contention

High-frequency reference churn is a measurable kernel-wide scalability bottleneck.

15.3.7 REAL Practical Example (C / C++ / Assembly)

KMDF Driver Safe Object Referencing

```
PEPROCESS Process;

status = PsLookupProcessByProcessId(Pid, &Process);
if (NT_SUCCESS(status)) {
    ObDereferenceObject(Process);
}
```

Explicit Handle-Based Reference

```
status = ObReferenceObjectByHandle(
    Handle,
    PROCESS_QUERY_INFORMATION,
    *PsProcessType,
    KernelMode,
```

```
    (PVOID*)&Process,
    NULL
);
```

External Assembly Reference Increment (ml64)

```
PUBLIC KernelRefInc

.code
KernelRefInc PROC
    ; RCX = OBJECT_HEADER*
    lock inc qword ptr [rcx]
    ret
KernelRefInc ENDP
END
```

Linked via `ml64.exe`. Inline x64 assembly is not supported by MSVC.

15.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump object header with reference counts:

```
!object <ObjectAddress> 1
```

Inspect object header manually:

```
dt nt!_OBJECT_HEADER <HeaderAddress>
```

Track live references:

```
!obtrace on
```

Detect leaked references:

```
!handle 0 1
```

15.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Vulnerabilities

- Reference leak exploitation
- Use-after-free via missing `ObDereferenceObject`
- Counter underflow attacks

Windows 11 Protections

- SMEP/SMAP protect object headers
- HVCI blocks inline hook attacks
- PatchGuard validates:
  - Type callbacks
  - Object header invariants
- CFG protects dereference call targets

Modern Windows 11 systems convert reference corruption into deterministic bug checks.

15.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Every successful:
  - `ObReferenceObject`
  - `ObReferenceObjectByHandle`

- **must be paired with**:
  - `ObDereferenceObject`
- Never:
  - Modify `PointerCount` manually in C
  - Touch `HandleCount` from drivers
- Never dereference objects at:
  - `DISPATCH_LEVEL`
  - `HIGH_LEVEL`
- Most common crash caused by violations:
  - `REFERENCE_BY_POINTER`
  - `IRQL_NOT_LESS_OR_EQUAL`
- Reference bugs scale into:
  - Silent memory corruption
  - Pool leaks
  - Non-reproducible scheduler failures

Chapter 16

Handle Tables: Exploitation & Protection

16.1 How Handles Are Managed Internally

16.1.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, a **handle** is a protected, per-process indirection token that maps a user-mode integer value to a kernel-mode executive object through a **per-process handle table** managed by the Object Manager. The handle itself contains no direct kernel pointer and is resolved exclusively through the process-specific `_HANDLE_TABLE`.
Handle management guarantees:

- Kernel pointer isolation from Ring 3
- Per-process namespace separation
- Fine-grained access mask enforcement
- Lifetime synchronization with object reference counting

Every handle operation in Windows 11 is validated through:

`EPROCESS->ObjectTable` ⇒ `_HANDLE_TABLE` ⇒ `_HANDLE_TABLE_ENTRY`

16.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The handle table is the **primary security boundary** between:

- User mode object access (Ring 3)
- Kernel object storage (Ring 0)

Its architectural responsibilities include:

- Enforcing:
  - Access masks
  - Inheritance rules
  - Audit policies
- Linking user handles to:
  - `FILE`
  - `EPROCESS`
  - `ETHREAD`
  - `SECTION`
  - `EVENT`, `MUTANT`, `SEMAPHORE`
- Synchronizing with:

- Object reference counters
- Security descriptors

The handle table is referenced exclusively through:

```
EPROCESS.ObjectTable
```

16.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

_HANDLE_TABLE (Windows 11 x64)

```
typedef struct _HANDLE_TABLE {
    ULONG NextHandleNeedingPool;
    LONG ExtraInfoPages;
    volatile ULONG_PTR TableCode;
    PEPROCESS QuotaProcess;
    PVOID UniqueProcessId;
    EX_PUSH_LOCK HandleTableLock;
    LIST_ENTRY HandleTableList;
    EX_PUSH_LOCK HandleContentionEvent;
} HANDLE_TABLE, *PHANDLE_TABLE;
```

_HANDLE_TABLE_ENTRY

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        ULONG_PTR Object;
        ULONG ObAttributes;
        PVOID InfoTable;
    };
    union {
        ULONG GrantedAccess;
```

```
        struct {
            USHORT GrantedAccessIndex;
            USHORT CreatorBackTraceIndex;
        };
        LONG NextFreeTableEntry;
    };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

Encoded Handle Value (EXHANDLE)

```
typedef struct _EXHANDLE {
    union {
        struct {
            ULONG_PTR TagBits : 2;
            ULONG_PTR Index   : 30;
        };
        ULONG_PTR GenericHandleOverlay;
    };
} EXHANDLE, *PEXHANDLE;
```

16.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Handle Creation (NtCreateXxx)

1. User calls `NtCreateFile`
2. Object is created via `ObCreateObject`
3. Handle allocated via `ExCreateHandle`
4. `HANDLE_TABLE_ENTRY` populated
5. Access mask applied

6. `HandleCount++`
7. Handle returned to user mode

Handle Resolution (NtQueryObject)

1. User passes handle value
2. Kernel extracts index from `EXHANDLE`
3. Table slot resolved in `_HANDLE_TABLE`
4. Pointer decoded from `HANDLE_TABLE_ENTRY.Object`
5. Access validated against `GrantedAccess`

Handle Close (NtClose)

1. `HandleCount–`
2. Entry marked free
3. If `HandleCount == 0`:
    - `ObDereferenceObject`

Assembly-Level Handle Entry Resolution

```
; RCX = EXHANDLE
; RDX = HANDLE_TABLE*

shr rcx, 2                ; Extract index
mov rax, [rdx]            ; TableCode base
lea rax, [rax + rcx*16]   ; Entry size = 16 bytes
mov rax, [rax]            ; Object pointer
```

16.1.5 Secure Kernel / VBS / HVCI Interaction

- Handle tables exist strictly in VTL0
- Secure Kernel (VTL1):
  - Cannot resolve user-mode handles
  - Only receives sanitized kernel object references
- HVCI:
  - Prevents:
    - Handle validation bypass
    - Object pointer forgery
- PatchGuard:
  - Monitors:
    - Handle table code
    - Handle decode paths

Any unauthorized handle manipulation results in immediate kernel bug check.

16.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Handle lookup is:
  - O(1)
  - Cache-sensitive
- NUMA systems:

  - Cross-node process handle access causes cache line migration

- High-frequency handle churn causes:
  - Handle table lock contention
  - Dispatch latency at `NtClose`

16.1.7 REAL Practical Example (C / C++ / Assembly)

Kernel Driver: Resolving a User Handle

```
PFILE_OBJECT FileObject;
status = ObReferenceObjectByHandle(
    UserHandle,
    FILE_READ_DATA,
    *IoFileObjectType,
    KernelMode,
    (PVOID*)&FileObject,
    NULL
);

if (NT_SUCCESS(status)) {
    ObDereferenceObject(FileObject);
}
```

External Assembly: Table Entry Read (ml64)

```
PUBLIC ResolveHandle

.code
ResolveHandle PROC
    ; RCX = Handle
    ; RDX = HandleTable Base
```

```
    shr rcx, 2
    lea rax, [rdx + rcx*16]
    mov rax, [rax]
    ret
ResolveHandle ENDP
END
```

16.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump process handle table:

```
!handle 0 1 <ProcessEPROCESS>
```

Dump specific handle:

```
!handle <HandleValue> 7
```

Inspect handle entry:

```
dt nt!_HANDLE_TABLE_ENTRY <EntryAddress>
```

Trace handle leaks:

```
!htrace on
```

16.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common Historical Attacks

- Handle reuse attacks
- Stale handle dereference
- Cross-process handle duplication abuse
- Kernel object pointer extraction via information leaks

Windows 11 Protections

- Handle encoding via `EXHANDLE`
- Kernel pointer obfuscation
- SMEP/SMAP enforcement
- CFG on object callbacks
- VBS-enforced isolation of kernel metadata

Most historical handle exploitation chains are completely broken on Windows 11.

16.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never trust raw handle values inside kernel
- Always use:
  - `ObReferenceObjectByHandle`
- Never cache:
  - `HANDLE_TABLE_ENTRY`
  - Raw kernel pointers derived from handles
- Never touch:
  - `HandleCount` directly
  - `HANDLE_TABLE` internals from drivers
- Handle leaks directly cause:

  - Nonpaged pool exhaustion
  - Silent object retention
  - System-wide resource starvation

16.2 How They Are Exploited

16.2.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **handle exploitation** refers to any attack technique that abuses the semantic gap between a user-mode handle value and the underlying kernel object pointer stored inside the process `_HANDLE_TABLE`. The exploitation target is not the handle value itself, but:

- The `_HANDLE_TABLE_ENTRY.Object` pointer
- The `GrantedAccess` field
- The lifetime coupling between handle count and object reference count

All modern handle exploitation in Windows 11 depends on at least one of the following:

- Kernel memory corruption
- Use-after-free of kernel objects
- Race conditions in object lifetime
- Information disclosure of kernel addresses

16.2.2 Exact Architectural Role Inside the Windows 11 Kernel

The handle table sits at the junction of three security domains:

- User-mode access control
- Kernel object lifetime management
- Security descriptor enforcement

Exploitation therefore targets one of the following architectural roles:

- Bypassing access mask validation
- Rebinding a handle to a different object
- Forcing premature object destruction
- Hijacking kernel object type dispatch tables

Windows 11 assumes:

$$\texttt{Handle Value} \Rightarrow \texttt{Trusted Kernel Decode Path}$$

Any violation of this assumption constitutes a handle exploitation primitive.

16.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Primary exploitation-relevant fields:

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        ULONG_PTR Object;          // Encoded kernel pointer
        ULONG ObAttributes;        // Inherit, protect flags
```

```
        PVOID InfoTable;
    };
    union {
        ULONG GrantedAccess;        // Effective access mask
        struct {
            USHORT GrantedAccessIndex;
            USHORT CreatorBackTraceIndex;
        };
        LONG NextFreeTableEntry;
    };
} HANDLE_TABLE_ENTRY;
```

Attack surface fields:

- `Object` → pointer substitution
- `GrantedAccess` → access mask escalation
- `NextFreeTableEntry` → free-list poisoning

16.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Legitimate Handle Resolution Path

1. User passes handle to `NtReadFile`
2. Kernel extracts index from `EXHANDLE`
3. `EPROCESS->ObjectTable` dereferenced
4. `HANDLE_TABLE_ENTRY` located
5. `GrantedAccess` validated
6. `Object` pointer decoded

7. IO Manager uses kernel object

Exploitation Redirect Path

1. Attacker corrupts `HANDLE_TABLE_ENTRY.Object`
2. Handle now references attacker-chosen kernel address
3. Legitimate syscall resolves corrupted entry
4. Kernel operates on forged object pointer
5. Arbitrary kernel memory read/write achieved

Assembly-Level Hijack Primitive

```
; RCX = user supplied handle
; RDX = kernel handle table base

shr rcx, 2
lea rax, [rdx + rcx*16]   ; locate entry
mov rbx, [rax]            ; attacker-controlled pointer
mov rcx, rbx              ; hijacked object used by kernel
```

16.2.5 Secure Kernel / VBS / HVCI Interaction

- All handle tables exist in VTL0 only
- Secure Kernel (VTL1) never resolves user-mode handles
- HVCI enforces:
  - Signed kernel code only
  - Immutable handle decode paths

  - CFG on object type callbacks

- Any exploit that attempts:

  - Inline hook of handle decode
  - Object type table modification

  is immediately blocked under HVCI

VBS therefore eliminates entire exploitation classes based on kernel code redirection.

16.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Handle exploitation has measurable performance side effects that are often observable during real-world attacks:

- Abnormal handle churn causes:

  - L1/L2 cache thrashing
  - Handle table lock contention

- Free-list poisoning leads to:

  - CPU pipeline stalls on allocation paths

- Cross-NUMA handle spraying:

  - Causes remote cache line migration
  - Triggers abnormal QPI/Infinity Fabric traffic

These side effects are actively monitored by modern EDR engines.

16.2.7 REAL Practical Example (C / C++ / Assembly)

Typical Handle Spraying Primitive (User Mode)

```
HANDLE handles[4096];

for (int i = 0; i < 4096; i++) {
    handles[i] = CreateEventW(NULL, FALSE, FALSE, NULL);
}
```

Purpose:

- Saturate handle table
- Create deterministic index reuse
- Increase probability of stale handle reuse

Kernel Victim Pattern (Use-After-Free)

```
VOID FreeObject(PVOID Obj)
{
    ObDereferenceObject(Obj); // vulnerable if handle still exists
}
```

External Assembly: Forged Object Dereference

```
PUBLIC TriggerHandle

.code
TriggerHandle PROC
    mov rcx, rdx      ; attacker-controlled kernel pointer
    call ObfDereferenceObject
    ret
```

```
TriggerHandle ENDP
END
```

16.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect live handle corruption:

```
!handle 0 7 <EPROCESS>
```

Dump corrupted entry:

```
dq <HANDLE_TABLE_ENTRY> L2
```

Detect handle reuse:

```
!htrace -diff
```

Track object lifetime:

```
!object <KernelObject>
```

16.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Primary Exploitation Techniques

- Use-after-free on kernel objects with active handles
- Double-dereference of kernel objects
- Handle free-list poisoning
- Type confusion via overwritten `ObjectType`
- Access mask bypass via `GrantedAccess` overwrite

Windows 11 Defensive Barriers

- Pointer encoding via `EXHANDLE`
- Heap hardening in NonPagedPoolNx
- PatchGuard on object type tables
- SMEP/SMAP enforcement
- Control Flow Guard on dispatch paths

Modern Windows 11 exploitation therefore requires:

$$\texttt{Information Leak} + \texttt{Write Primitive} + \texttt{Lifetime Bug}$$

16.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never assume a handle implies object validity
- Always treat:
  - `ObReferenceObjectByHandle`
  - `ObDereferenceObject`

  as security-critical operations
- Never dereference:
  - Cached kernel object pointers across IRQL changes
  - Pointers derived from user-mode handles without revalidation

- Every historical Windows handle exploit chain:
  - Starts with a lifetime bug
  - Ends with object pointer substitution

16.3 How They Are Protected

16.3.1 Precise Windows 11Specific Definition (Engineering-Level)

In Windows 11, **handle table protection** is the set of hardware-enforced and kernel-enforced mechanisms that guarantee the integrity of:

- `_HANDLE_TABLE`
- `_HANDLE_TABLE_ENTRY`
- Object pointer encoding and decoding
- Access mask validation
- Object lifetime consistency

The protection model is explicitly designed to prevent:

- Handle value forgery
- Object pointer substitution
- Access mask escalation
- Free-list poisoning

These protections operate continuously at **IRQL-sensitive, VBS-aware, HVCI-enforced** execution layers.

16.3.2 Exact Architectural Role Inside the Windows 11 Kernel

The handle table protection system is positioned at three kernel enforcement boundaries:

1. **Object Manager**: reference tracking and access checks
2. **Memory Manager**: pointer encoding and pool isolation
3. **Hypervisor Layer**: kernel control-flow and memory integrity

Architecturally, protection is enforced during:

- Handle creation (`ObInsertObject`)
- Handle lookup (`ObpLookupObjectByHandle`)
- Handle deletion (`ExDestroyHandle`)
- Object dereference (`ObDereferenceObject`)

At no point does Windows 11 allow unverified traversal from handle value to kernel object pointer.

16.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Encoded Handle Representation

```
typedef struct _EXHANDLE {
    union {
        struct {
            ULONG TagBits : 2;
            ULONG Index   : 30;
        };
        ULONG_PTR Value;
    };
} EXHANDLE;
```

Protected Handle Entry Layout

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        ULONG_PTR Object;               // Encoded pointer
        ULONG ObAttributes;
        PVOID InfoTable;
    };
    union {
        ULONG GrantedAccess;
        LONG NextFreeTableEntry;
    };
} HANDLE_TABLE_ENTRY;
```

Kernel Guarded Fields

- `Object` is pointer-encoded
- `GrantedAccess` is masked and verified
- `NextFreeTableEntry` is guarded by lock-free sequencing

16.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Protected Handle Resolution Flow

1. User supplies handle to syscall
2. Kernel extracts `EXHANDLE.Index`
3. `EPROCESS->ObjectTable` base retrieved
4. Entry address computed
5. Pointer decoding using kernel secret

6. `GrantedAccess` verified
7. Object reference count incremented

Pointer Decoding (Assembly-Level)

```
mov rax, [rdi]              ; encoded object pointer
xor rax, gs:[188h]          ; per-CPU secret decode key
ror rax, 13                 ; rotate back original pointer
```

Access Mask Verification (Kernel Logic)

```
if ((Entry->GrantedAccess & DesiredAccess) != DesiredAccess) {
    return STATUS_ACCESS_DENIED;
}
```

16.3.5 Secure Kernel / VBS / HVCI Interaction

Virtualization-Based Security (VBS)

- Handle tables reside exclusively in VTL0
- Secure Kernel enforces:
  - Page-level write protection
  - Kernel pointer integrity
- Any illegal handle table write triggers:
  - Hypervisor memory violation
  - Immediate system bugcheck

Hypervisor-Protected Code Integrity (HVCI)

- All handle decode routines are CFG-protected
- PatchGuard verifies:
  - Object type tables
  - Handle dispatch tables
- Inline hooks in handle resolution paths are blocked

SMEP, SMAP, and KVA Shadow

- User-mode pointers cannot be dereferenced
- Kernel stack cannot jump to user code
- Address-space isolation prevents cross-mode pointer reuse

16.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Handle protection introduces controlled overhead:

- Pointer decoding adds:
  - Single-cycle XOR
  - Rotate instruction
- Access verification adds:
  - Branch prediction dependency
- NUMA handle locality:

  - Node-local handle tables minimize cross-node cache traffic

- HVCI:

  - Adds hypervisor validation cost during kernel entry

All overhead is bounded and deterministic under Windows 11 scheduler design.

16.3.7 REAL Practical Example (C / C++ / Assembly)

Safe Handle Validation (Kernel Driver)

```
PVOID Object;
NTSTATUS status;

status = ObReferenceObjectByHandle(
    Handle,
    FILE_READ_DATA,
    *IoFileObjectType,
    KernelMode,
    &Object,
    NULL
);

if (!NT_SUCCESS(status)) {
    return status;
}

ObDereferenceObject(Object);
```

External Assembly: Protected Object Use

```
PUBLIC SafeKernelUse
```

```
.code
SafeKernelUse PROC
    sub rsp, 40h
    call ObReferenceObjectByHandle
    add rsp, 40h
    ret
SafeKernelUse ENDP
END
```

Handle Protection via Pool Tag Isolation

```
PVOID p = ExAllocatePool2(
    POOL_FLAG_NON_PAGED | POOL_FLAG_SECURE,
    sizeof(MY_OBJECT),
    'HdlP'
);
```

16.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Verify handle protection state:

```
!handle 0 3 <EPROCESS>
```

Inspect encoded entry:

```
dq <HANDLE_TABLE_ENTRY> L1
```

Check hypervisor security:

```
!hvinfo
```

Validate object type protection:

```
!object \File
```

16.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

What Windows 11 Successfully Blocks

- Direct handle value forgery
- Object pointer overwrites
- Type confusion via dispatch table patching
- Access mask overwrites
- Cross-process handle reuse

What Still Remains Attackable

- Logic bugs in drivers
- Double-release conditions
- Race conditions in reference counting
- Incorrect IRQL object usage

All exploitation now requires:

```
Memory Corruption + Lifetime Violation + Synchronization Failure
```

16.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never assume a handle remains valid after releasing a lock

- Always revalidate handles after:
  - IRQL transitions
  - APC delivery
  - Thread context switches
- Never cache decoded object pointers across:
  - User/kernel transitions
  - Process termination
- Treat every handle as:
  - Untrusted until verified
  - Potentially stale until referenced
- Windows 11 handle security is:
  - Hardware-rooted
  - Hypervisor-enforced
  - Pool-isolated
  - Control-flow verified

Part IX

I/O Manager, Drivers & Hardware Access

Chapter 17

I/O Request Packets (IRP) from Driver Entry to Completion

17.1 Step-by-Step IRP Analysis

17.1.1 Precise Windows 11Specific Definition (Engineering-Level)

An **I/O Request Packet (IRP)** is the fundamental kernel object used by the Windows 11 I/O Manager to represent a single I/O operation as it transits through the driver stack. An IRP encapsulates:

- The requesting thread and process context
- The target device object stack
- The major and minor function codes
- Driver-specific parameters per stack location

- Completion, cancellation, and synchronization state

In Windows 11, IRPs are **strictly non-paged kernel objects** allocated from protected pool regions and are fully governed by:

- IRQL rules
- Object Manager lifetime tracking
- Memory Manager pool isolation
- HVCI and CFG enforcement

17.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The IRP is the **execution carrier** for all synchronous and asynchronous I/O. It forms the execution spine connecting:

1. User-mode system calls
2. I/O Manager dispatch
3. File system drivers
4. Filter drivers
5. Function drivers
6. Bus drivers
7. Hardware interrupt completion paths

Architecturally:

- IRP creation occurs in `ntoskrnl.exe`
- IRP dispatch occurs through `DRIVER_OBJECT->MajorFunction[]`
- IRP completion propagates backward through the device stack

The IRP is therefore both a **control-flow object** and a **data-flow object**.

17.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Core IRP Layout

```
typedef struct _IRP {
    CSHORT Type;
    USHORT Size;
    PMDL MdlAddress;
    ULONG Flags;

    union {
        struct _IRP *MasterIrp;
        LONG IrpCount;
        PVOID SystemBuffer;
    } AssociatedIrp;

    LIST_ENTRY ThreadListEntry;

    IO_STATUS_BLOCK IoStatus;
    KPROCESSOR_MODE RequestorMode;
    BOOLEAN PendingReturned;
    CHAR StackCount;
    CHAR CurrentLocation;

    BOOLEAN Cancel;
    KIRQL CancelIrql;
```

```
    CCHAR ApcEnvironment;
    UCHAR AllocationFlags;

    PIO_STATUS_BLOCK UserIosb;
    PKEVENT UserEvent;

    union {
        struct {
            PIO_APC_ROUTINE UserApcRoutine;
            PVOID UserApcContext;
        } AsynchronousParameters;

        LARGE_INTEGER AllocationSize;
    } Overlay;

    PDRIVER_CANCEL CancelRoutine;

    PVOID UserBuffer;

    union {
        struct {
            union {
                KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
                struct {
                    PVOID DriverContext[4];
                };
            };
            PETHREAD Thread;
            PCHAR AuxiliaryBuffer;
            LIST_ENTRY ListEntry;
            union {
                struct _IO_STACK_LOCATION *CurrentStackLocation;
                ULONG PacketType;
```

```
            };
            PFILE_OBJECT OriginalFileObject;
        } Tail;
    };
} IRP;
```

Stack Location Layout

```
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
    UCHAR Control;

    union {
        struct {
            PVOID SecurityContext;
            ULONG Options;
            USHORT FileAttributes;
            USHORT ShareAccess;
            ULONG EaLength;
        } Create;

        struct {
            ULONG Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
        } Read;

        struct {
            ULONG Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
```

```
        } Write;

        struct {
            ULONG IoControlCode;
            PVOID Type3InputBuffer;
        } DeviceIoControl;
    } Parameters;

    PDEVICE_OBJECT DeviceObject;
    PFILE_OBJECT FileObject;
    PIO_COMPLETION_ROUTINE CompletionRoutine;
    PVOID Context;
} IO_STACK_LOCATION;
```

17.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

User-Mode to Kernel Transition

1. User thread calls `NtReadFile`
2. CPU executes `syscall`
3. Kernel dispatch enters I/O Manager
4. IRP is allocated using `IoAllocateIrp`

IRP Initialization

1. `Type`, `Size`, `StackCount` initialized
2. Stack locations carved at tail of IRP
3. `CurrentLocation = StackCount + 1`
4. Parameters filled for target device

Driver Dispatch

1. I/O Manager invokes:

   ```
   DriverObject->MajorFunction[IRP_MJ_READ]
   ```

2. Driver receives:

   - `PDEVICE_OBJECT`
   - `PIRP`

IRP Forwarding

```
IoSkipCurrentIrpStackLocation(Irp);
return IoCallDriver(LowerDevice, Irp);
```

Completion Propagation

1. Lowest driver completes IRP
2. Completion routines fire in reverse order
3. Final completion returns to I/O Manager
4. User thread is signaled

17.1.5 Secure Kernel / VBS / HVCI Interaction

- IRP memory regions are protected by VTL0 hypervisor policy
- HVCI enforces:
  - No executable IRP memory

  - No control-flow redirection via IRP fields

- PatchGuard periodically verifies:
  - IRP dispatch vectors
  - MajorFunction tables
- Any tampering results in immediate bugcheck

17.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Each IRP allocation touches multiple cache lines
- NUMA-local device stacks reduce QPI/IF fabric traffic
- Stack depth directly affects:
  - Dispatch latency
  - Completion latency
- DPC-based completions compete with scheduler quantum

17.1.7 REAL Practical Example (C / C++ / Assembly)

Minimal Dispatch Routine

```
NTSTATUS DispatchRead(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    UNREFERENCED_PARAMETER(DeviceObject);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
```

```
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

External Assembly Stub (IRP Completion Path)

```
PUBLIC CompleteIrpStub

.code
CompleteIrpStub PROC
    sub rsp, 40h
    call IoCompleteRequest
    add rsp, 40h
    ret
CompleteIrpStub ENDP
END
```

Driver Dispatch Table Binding

```
DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
```

17.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect active IRPs:

```
!irpfind
```

Decode a specific IRP:

```
!irp <address>
```

Inspect stack locations:

```
dt nt!_IO_STACK_LOCATION <address>
```

Verify driver dispatch:

```
!drvobj <driver> 7
```

17.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically Exploited Weak Points

- Improper `METHOD_NEITHER` buffering
- Unvalidated `UserBuffer` pointers
- Double `IoCompleteRequest`
- Missing cancel routine synchronization

What Windows 11 Blocks

- IRP stack overflows
- IRP reuse after completion
- Arbitrary kernel pointer reuse via IRP fields

Remaining Risk Sources

- Driver logic bugs
- Incorrect IRQL usage
- Unsynchronized cancellation paths

17.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never touch `Irp->UserBuffer` at `DISPATCH_LEVEL`
- Always set:
  - `IoStatus.Status`
  - `IoStatus.Information`

  before completion
- Never complete an IRP twice
- Always match:

$$\texttt{IoSkipCurrentIrpStackLocation} \Rightarrow \texttt{IoCallDriver}$$

- Cancellation paths must be synchronized with spinlocks
- Stack location corruption almost always leads to:
  - Immediate system crash
  - Silent data corruption
- In Windows 11, IRP misuse is detected earlier due to:
  - HVCI
  - PatchGuard
  - Pool hardening

17.2 Dispatch Routines

17.2.1 Precise Windows 11Specific Definition (Engineering-Level)

A **dispatch routine** is a kernel-mode function registered inside a `DRIVER_OBJECT` that serves as the **primary execution entry point for IRP processing** in Windows 11. Each dispatch routine is indexed by a specific **IRP major function code** and is invoked synchronously by the I/O Manager during IRP propagation through the driver stack. Formally, a dispatch routine is defined as:

```
NTSTATUS (*PDRIVER_DISPATCH)(PDEVICE_OBJECT, PIRP)
```

In Windows 11, dispatch routines are subject to:

- IRQL execution constraints
- HVCI-protected control-flow integrity
- PatchGuard validation
- Strict non-executable memory separation

17.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Dispatch routines form the **driver-side execution boundary** between:

- The I/O Manager (`ntoskrnl.exe`)
- The filter / function / bus driver stacks
- The hardware abstraction layer

Architecturally:

1. The I/O Manager decodes the IRP major function.
2. It indexes into:

   ```
   DriverObject->MajorFunction[MajorFunctionCode]
   ```

3. The resolved function pointer is executed in kernel mode at the callers IRQL.

Dispatch routines therefore define:

- Driver I/O capabilities
- Synchronization model
- Power and PnP behavior
- Security and access enforcement

17.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

DRIVER_OBJECT Dispatch Table

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
```

```
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
```

Dispatch Prototype

```
typedef NTSTATUS (*PDRIVER_DISPATCH)(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp
);
```

IRP Stack Access

```
PIO_STACK_LOCATION IoGetCurrentIrpStackLocation(PIRP Irp);
PIO_STACK_LOCATION IoGetNextIrpStackLocation(PIRP Irp);
```

17.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Entry from I/O Manager

1. IRP arrives at top of driver stack.
2. I/O Manager retrieves:

   `DriverObject->MajorFunction[Irp->MajorFunction]`

3. Dispatch routine is invoked at current IRQL.

Stack Location Processing

1. Driver extracts parameters:

   `IoGetCurrentIrpStackLocation(Irp)`

2. Validates:

    - Buffer pointers
    - Lengths
    - Control flags

Forward or Complete

- If forwarding:

$$\texttt{IoSkipCurrentIrpStackLocation} \Rightarrow \texttt{IoCallDriver}$$

- If terminating:

$$\texttt{IoCompleteRequest}$$

Assembly-Level Call Path (Simplified)

```
mov  rcx, DeviceObject
mov  rdx, Irp
call qword ptr [DriverDispatchTable + MajorFunction*8]
```

17.2.5 Secure Kernel / VBS / HVCI Interaction

- Dispatch pointers are protected by Hyper-V enforced CFI.
- Any runtime modification of:

$$\texttt{DriverObject->MajorFunction[]}$$

  triggers PatchGuard violation.

- Dispatch code pages must be:

  - Executable
  - Signed
  - HVCI-compliant
- No dispatch routine may execute from writable memory.

17.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Each dispatch transition introduces:
  - Branch misprediction risk
  - Instruction cache pressure
- Deep driver stacks multiply dispatch latency.
- NUMA-local dispatch improves:
  - Cache locality
  - IRP throughput
- Improper dispatch synchronization causes:
  - DPC starvation
  - Scheduler priority inversion

17.2.7 REAL Practical Example (C / C++ / Assembly)

Minimal Read Dispatch

```
NTSTATUS DispatchRead(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp
```

```
)
{
    UNREFERENCED_PARAMETER(DeviceObject);

    PIO_STACK_LOCATION irpSp =
        IoGetCurrentIrpStackLocation(Irp);

    ULONG length = irpSp->Parameters.Read.Length;

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = length;

    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Dispatch Table Registration

```
DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
```

External Assembly Completion Stub

```
PUBLIC DispatchThunk

.code
DispatchThunk PROC
    sub rsp, 40h
    call IoCompleteRequest
    add rsp, 40h
    ret
DispatchThunk ENDP
END
```

17.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

List dispatch routines:

```
!drvobj drivername 7
```

Inspect IRP in dispatch:

```
!irp @rdx
```

Verify stack location:

```
dt nt!_IO_STACK_LOCATION
```

Trace IRP flow:

```
!irpfind
```

17.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common Historical Vulnerabilities

- Missing length validation
- `METHOD_NEITHER` misuse
- Double IRP completion
- Unsynchronized cancellation

Modern Windows 11 Mitigations

- HVCI blocks dispatch hijacking
- PatchGuard enforces table integrity
- Kernel pool isolation blocks IRP corruption reuse

Residual Risk

- Logic errors in dispatch paths
- Incorrect IRQL behavior
- Unverified user pointers

17.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never trust IRP input buffers blindly.
- Never access pageable memory above `PASSIVE_LEVEL`.
- Always initialize:
  - `IoStatus.Status`
  - `IoStatus.Information`
- Always use:

  `IoSkipCurrentIrpStackLocation` ⇒ `IoCallDriver`

- Never mix completion ownership across drivers.
- Dispatch routines must be:
  - Deterministic
  - Lock-safe
  - Re-entrant where required
- Windows 11 detects dispatch violations faster due to:

- VBS
- HVCI
- CFG

17.3 Completion Routines

17.3.1 Precise Windows 11Specific Definition (Engineering-Level)

A **completion routine** is a kernel-mode callback registered within an IRP stack location that is executed when a lower driver finishes processing an IRP and returns control upward in the driver stack. In Windows 11, completion routines execute strictly under the I/O Managers unwind phase and are bound by strict IRQL, stack ownership, and security constraints. Formally, a completion routine follows the prototype:

```
typedef NTSTATUS (*PIO_COMPLETION_ROUTINE)(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
);
```

Completion routines exist solely to:

- Post-process results from lower drivers
- Synchronize asynchronous operations
- Modify IRP status or buffers
- Resume blocked execution paths

17.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Completion routines form the **reverse execution channel** of the IRP pipeline. While dispatch routines push IRPs downward, completion routines propagate execution upward.
Architecturally:

1. Lower driver completes IRP using `IoCompleteRequest`
2. I/O Manager walks stack locations upward
3. For each stack entry:
    - If a completion routine exists it is executed
    - If it returns `STATUS_MORE_PROCESSING_REQUIRED` unwind stops

Completion routines therefore enforce:

- Asynchronous execution ordering
- Event synchronization
- Buffer integrity
- IRP ownership control

17.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

IO_STACK_LOCATION Completion Fields

```
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
```

```
    UCHAR Control;
    union {
        struct {
            PIO_COMPLETION_ROUTINE CompletionRoutine;
            PVOID Context;
        };
        // Other parameter unions omitted
    };
    PDEVICE_OBJECT DeviceObject;
    PFILE_OBJECT FileObject;
} IO_STACK_LOCATION;
```

IRP Completion Flags

```
#define SL_INVOKE_ON_SUCCESS   0x40
#define SL_INVOKE_ON_ERROR     0x80
#define SL_INVOKE_ON_CANCEL    0x20
```

Registration API

```
VOID IoSetCompletionRoutine(
    PIRP Irp,
    PIO_COMPLETION_ROUTINE CompletionRoutine,
    PVOID Context,
    BOOLEAN InvokeOnSuccess,
    BOOLEAN InvokeOnError,
    BOOLEAN InvokeOnCancel
);
```

17.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

IRP Downward Path

1. Driver prepares IRP
2. Registers completion routine
3. Forwards IRP using `IoCallDriver`

Completion Trigger

1. Lower driver calls `IoCompleteRequest`
2. I/O Manager transitions IRP from lower stack to upper stack
3. Completion routine is conditionally invoked based on flags

Completion Control Decision

- Return `STATUS_SUCCESS` continue unwind
- Return `STATUS_MORE_PROCESSING_REQUIRED` caller retains IRP ownership

Assembly-Level Unwind Call (Simplified)

```
mov  rcx, DeviceObject
mov  rdx, Irp
mov  r8,  Context
call CompletionRoutine
```

17.3.5 Secure Kernel / VBS / HVCI Interaction

- Completion routine pointers are validated by HVCI

- Runtime overwrite of:

  `IO_STACK_LOCATION->CompletionRoutine`

  triggers immediate kernel integrity violation

- Code pages must be:
  - Signed
  - Executable-only
  - Hypervisor-validated
- Secure Kernel enforces isolation of IRP ownership across VTL boundaries

17.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- Each completion transition introduces:
  - Branch overhead
  - Instruction cache churn
- Excessive completion nesting increases:
  - IRP latency
  - Scheduler pressure
- NUMA-remote completions cause:
  - Additional memory coherence traffic
  - Reduced throughput

17.3.7 REAL Practical Example (C / C++ / Assembly)

Completion Routine Implementation

```
NTSTATUS ReadCompletion(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
)
{
    UNREFERENCED_PARAMETER(DeviceObject);
    UNREFERENCED_PARAMETER(Context);

    if (Irp->IoStatus.Status == STATUS_SUCCESS) {
        // Post-process buffer
    }

    KeSetEvent((PKEVENT)Context, IO_NO_INCREMENT, FALSE);

    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Registration Before Forwarding

```
IoSetCompletionRoutine(
    Irp,
    ReadCompletion,
    &Event,
    TRUE,
    TRUE,
    TRUE
);

IoCallDriver(NextDeviceObject, Irp);
```

External Assembly Completion Helper

```
PUBLIC CompletionThunk

.code
CompletionThunk PROC
    sub rsp, 40h
    call KeSetEvent
    add rsp, 40h
    ret
CompletionThunk ENDP
END
```

17.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect completion routine in IRP:

```
!irp @rdx
```

Dump stack locations:

```
dt nt!_IO_STACK_LOCATION @rdx
```

Trace IRP completion path:

```
!ioctldecode
```

Detect stalled completion:

```
!stacks 2
```

17.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Vulnerability Classes

- Returning wrong NTSTATUS
- Double completion
- Reusing freed IRP
- User-pointer dereference in completion

Windows 11 Mitigations

- HVCI enforces callback integrity
- Pool isolation blocks UAF chaining
- PatchGuard monitors IRP control fields

Residual Risk

- Logic flaws in completion ordering
- Incorrect STATUS_MORE_PROCESSING_REQUIRED usage

17.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never complete an IRP twice.
- Never touch user buffers in completion without prior probing.
- Always validate `IoStatus.Status` before post-processing.

- Never block inside a completion routine.
- Only return `STATUS_MORE_PROCESSING_REQUIRED` when you explicitly retain ownership.
- Completion routines must always be:
  - Non-pageable
  - Re-entrant safe
  - IRQL-aware
- Windows 11 strictly detects invalid completion behavior via:
  - PatchGuard
  - VBS
  - HVCI

Chapter 18

Writing Windows 11 Kernel Drivers in C/C++

18.1 The `DriverEntry` Structure

18.1.1 Precise Windows 11Specific Definition (Engineering-Level)

`DriverEntry` is the mandatory kernel-mode entry point executed by the Windows 11 I/O Manager when a driver image is mapped into kernel address space. It executes at **PASSIVE_LEVEL**, inside the System process context, after image signature verification (Code Integrity + HVCI) and before any IRP traffic is delivered to the driver.

Formally, the entry point conforms to:

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT  DriverObject,
    PUNICODE_STRING RegistryPath
);
```

Its sole architectural purpose is to **initialize the driver as an executable kernel object** and publish its dispatch and unload interfaces to the I/O Manager.

18.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Inside the Windows 11 kernel, `DriverEntry` is invoked by:

`ntoskrnl!IopLoadDriver` `ntoskrnl!IopInitializeDriverModule`

Its responsibilities are strictly architectural:

- Bind IRP major function vectors into `DRIVER_OBJECT`
- Register `DriverUnload`
- Initialize device objects
- Initialize symbolic links
- Initialize KMDF (if used)
- Establish security descriptors

Execution of `DriverEntry` represents the **only moment where the driver is allowed to fully configure its execution contract with the kernel**.

18.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

DRIVER_OBJECT

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
```

```
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH FastIoDispatch;
    PVOID DriverInit;
    PVOID DriverStartIo;
    PVOID DriverExtension2;
} DRIVER_OBJECT;
```

DRIVER_EXTENSION

```
typedef struct _DRIVER_EXTENSION {
    DRIVER_OBJECT *DriverObject;
    PVOID AddDevice;
    ULONG Count;
    UNICODE_STRING ServiceKeyName;
} DRIVER_EXTENSION;
```

Driver Object Flags (Windows 11)

```
#define DRVO_INITIALIZED   0x00000008
#define DRVO_UNLOAD_INVOKED 0x00000010
```

18.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel Load Path

1. `SCM` requests driver load
2. `ntoskrnl!MmLoadSystemImage`
3. Code Integrity signature validation (HVCI enforced)
4. Image mapped into kernel address space
5. `DriverEntry` address resolved
6. `DriverEntry` executed at **PASSIVE_LEVEL**

Assembly-Level Call Site (Conceptual)

```
mov  rcx, DriverObject
mov  rdx, RegistryPath
sub  rsp, 20h
call DriverEntry
add  rsp, 20h
```

Post-Return State

- `DriverObject->MajorFunction[]` is now callable
- Device stacks may now be attached
- IRPs may now be issued

18.1.5 Secure Kernel / VBS / HVCI Interaction

- `DriverEntry` cannot execute unless the image passes:
  - Code Integrity verification

  - Hypervisor-enforced signature validation (HVCI)

- Runtime patching of `DriverEntry` is blocked by:
  - EPT-based execute permissions
  - PatchGuard function table validation

- Secure Kernel enforces that the entry point address lies in:

`IMAGE_SECTION_CODE`

18.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- `DriverEntry` executes cold in instruction cache
- Code locality impacts initial driver attach latency
- NUMA-unaware allocation during `DriverEntry` causes:
  - Remote memory faults
  - Cross-node cache thrashing
- Any blocking operation increases:
  - System boot latency
  - Driver chain initialization time

18.1.7 REAL Practical Example (C / C++ / Assembly)

Minimal Real DriverEntry Implementation

```
extern "C"
NTSTATUS DriverEntry(
```

```
    PDRIVER_OBJECT  DriverObject,
    PUNICODE_STRING RegistryPath
)
{
    UNREFERENCED_PARAMETER(RegistryPath);

    for (UINT32 i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; ++i)
        DriverObject->MajorFunction[i] = DispatchDefault;

    DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]  = DispatchClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoctl;

    DriverObject->DriverUnload = DriverUnload;

    return STATUS_SUCCESS;
}
```

External Assembly Driver Entry Thunk

```
PUBLIC DriverEntryThunk

.code
DriverEntryThunk PROC
    sub rsp, 40h
    call DriverEntry
    add rsp, 40h
    ret
DriverEntryThunk ENDP
END
```

18.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Dump the driver object after load:

```
!drvobj MyDriver 2
```

Verify entry point mapping:

```
lm m MyDriver
```

Inspect dispatch table:

```
dt nt!_DRIVER_OBJECT <address>
```

Trace DriverEntry execution:

```
bp MyDriver!DriverEntry
g
```

18.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical Attack Classes

- Malicious overwrite of `DriverUnload`
- Dispatch table corruption
- Improper device security descriptors
- Arbitrary object exposure during `DriverEntry`

Windows 11 Defensive Barriers

- HVCI prevents executable memory tampering
- Kernel CFG protects dispatch pointer integrity
- PatchGuard monitors:
  - DRIVER_OBJECT mutation
  - Function pointer tampering

Residual Risk

- Logic flaws inside initialization path
- Improper IRQL use during initialization

18.1.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- `DriverEntry` must **never** execute at elevated IRQL.
- No pageable memory access above **PASSIVE_LEVEL**.
- All dispatch pointers must be initialized.
- Never leave uninitialized MajorFunction entries.
- Never allocate executable memory dynamically.
- Avoid blocking I/O inside `DriverEntry`.
- KMDF drivers must call `WdfDriverCreate` exclusively from `DriverEntry`.

- Windows 11 enforces:
  - Signature trust chain
  - Hypervisor code execution policy
  - Driver object structural integrity

18.2 Practical KMDF Usage

18.2.1 Precise Windows 11Specific Definition (Engineering-Level)

Kernel-Mode Driver Framework (KMDF) is the **mandatory modern driver execution framework** in Windows 11 that abstracts IRP dispatch, synchronization, power management, PnP state transitions, and object lifetime while preserving strict kernel execution semantics.
KMDF drivers execute entirely in **Ring 0**, under the direct control of the Windows 11 I/O Manager and Framework Runtime (`Wdf01000.sys`).
KMDF replaces direct IRP dispatch tables with **event-driven state machines** implemented through framework-managed callback objects.

18.2.2 Exact Architectural Role Inside the Windows 11 Kernel

KMDF is architecturally positioned as:

```
User  I/O Manager  Wdf01000.sys  Driver Callbacks
```

Its core functions:

- IRP translation into WDF request objects
- Automatic synchronization and lock enforcement

- Plug and Play state machine enforcement
- Power policy ownership
- DMA abstraction
- Automatic reference counting
- Enforced object lifetime tracking

KMDF drivers never directly process raw IRP stacks unless explicitly opted-in.

18.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

WDFDRIVER

```
typedef struct _WDF_DRIVER_CONFIG {
    ULONG Size;
    PFN_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd;
    PFN_WDF_OBJECT_CONTEXT_CLEANUP EvtDriverUnload;
    ULONG DriverInitFlags;
    ULONG DriverPoolTag;
} WDF_DRIVER_CONFIG;
```

WDFDEVICE

```
typedef struct _WDFDEVICE_INIT WDFDEVICE_INIT, *PWDFDEVICE_INIT;
```

WDFQUEUE

```
typedef struct _WDF_IO_QUEUE_CONFIG {
    ULONG Size;
    WDF_IO_QUEUE_DISPATCH_TYPE DispatchType;
    BOOLEAN PowerManaged;
```

```
    PFN_WDF_IO_QUEUE_IO_DEVICE_CONTROL EvtIoDeviceControl;
} WDF_IO_QUEUE_CONFIG;
```

18.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

1. `DriverEntry` calls `WdfDriverCreate`
2. Framework registers driver in `Wdf01000.sys`
3. PnP Manager triggers `EvtDriverDeviceAdd`
4. Device object created via `WdfDeviceCreate`
5. I/O Queues created via `WdfIoQueueCreate`
6. IRPs translated into `WDFREQUEST`
7. Driver callback executes at framework-controlled IRQL
8. Completion returned to I/O Manager

Assembly Transition

```
mov  rcx, DriverObject
mov  rdx, RegistryPath
sub  rsp, 20h
call WdfDriverCreate
add  rsp, 20h
```

18.2.5 Secure Kernel / VBS / HVCI Interaction

- KMDF drivers must satisfy:
  - Code Integrity signature validation

  - Hypervisor-enforced control flow integrity

- All executable memory used by KMDF is:
  - Non-writable
  - Non-pageable
  - Validated under PatchGuard
- Framework object metadata is validated by Secure Kernel

18.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

- KMDF request translation adds deterministic overhead
- Cache locality improves due to structured object lifetime
- NUMA-unaware queue allocations increase remote memory hits
- DPC execution is automatically optimized per CPU group
- Passive execution avoids spinlock contention

18.2.7 REAL Practical Example (C / C++ / Assembly)

Minimal KMDF DriverEntry

```
extern "C"
NTSTATUS DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
)
{
    WDF_DRIVER_CONFIG config;
```

```
    WDF_DRIVER_CONFIG_INIT(&config, EvtDeviceAdd);

    return WdfDriverCreate(
        DriverObject,
        RegistryPath,
        WDF_NO_OBJECT_ATTRIBUTES,
        &config,
        WDF_NO_HANDLE
    );
}
```

Device Creation Callback

```
NTSTATUS EvtDeviceAdd(
    WDFDRIVER Driver,
    PWDFDEVICE_INIT DeviceInit
)
{
    WDFDEVICE device;
    NTSTATUS status;

    status = WdfDeviceCreate(
        &DeviceInit,
        WDF_NO_OBJECT_ATTRIBUTES,
        &device
    );

    return status;
}
```

IOCTL Queue Setup

```
WDF_IO_QUEUE_CONFIG queueConfig;
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(
```

```
    &queueConfig,
    WdfIoQueueDispatchSequential
);
queueConfig.EvtIoDeviceControl = EvtIoDeviceControl;

WdfIoQueueCreate(
    device,
    &queueConfig,
    WDF_NO_OBJECT_ATTRIBUTES,
    WDF_NO_HANDLE
);
```

18.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

```
!wdfdriver
!wdfdevice
!wdfqueue
```

```
lm m Wdf01000
```

```
!devstack <DeviceObject>
```

18.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Improper IOCTL validation
- Dangling WDF object references
- Unsafe memory mapping via `METHOD_NEITHER`
- Excessive callback permissions

Windows 11 blocks:

- Unsigned KMDF drivers
- Runtime patching of WDF tables
- Untrusted DMA mappings

18.2.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never mix WDM and KMDF dispatch models.
- All WDF objects must have deterministic parentage.
- Never access IRP directly unless explicitly requested.
- Never execute pageable code above PASSIVE_LEVEL.
- Always use `WdfRequestComplete` for request termination.
- Never store raw pointers to WDF objects.
- Power policy must remain framework-controlled.

18.3 Errors That Directly Cause BSOD

18.3.1 Precise Windows 11Specific Definition (Engineering-Level)

A Blue Screen of Death (BSOD) in Windows 11 is a **controlled kernel halt** triggered when the kernel detects a **non-recoverable violation of execution integrity, memory safety, IRQL contract, or synchronization correctness**. For kernel drivers written in C/C++, BSODs are most commonly caused by:

- Invalid memory access at elevated IRQL
- Illegal object lifetime violations
- Corruption of kernel pools
- Improper interrupt/DPC usage
- Corruption of scheduler structures
- Violations enforced by HVCI and Secure Kernel

Every BSOD corresponds to a specific **bug check code** issued by `KeBugCheckEx`.

18.3.2 Exact Architectural Role Inside the Windows 11 Kernel

The BSOD mechanism is enforced across:

- **ntoskrnl.exe** Central fault detector and bugcheck dispatcher
- **HAL.dll** Hardware interrupt integrity enforcement
- **Secure Kernel (VTL1)** HVCI and code integrity enforcement
- **Hyper-V Hypervisor** Page permission enforcement

The execution path is:

```
Fault  Trap Handler  KiDispatchException  KeBugCheckEx
```

Once entered, execution **never returns**.

18.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Bug Check Record

```
typedef struct _KBUGCHECK_DATA {
    ULONG BugCheckCode;
    ULONG_PTR BugCheckParameter1;
    ULONG_PTR BugCheckParameter2;
    ULONG_PTR BugCheckParameter3;
    ULONG_PTR BugCheckParameter4;
} KBUGCHECK_DATA;
```

Trap Frame (x64)

```
typedef struct _KTRAP_FRAME {
    UINT64 Rip;
    UINT64 Rsp;
    UINT64 Rflags;
    UINT64 Rax;
    UINT64 Rcx;
    UINT64 Rdx;
} KTRAP_FRAME;
```

Thread Control Block (Relevant Portion)

```
typedef struct _KTHREAD {
    UCHAR CurrentIrql;
    UINT64 KernelStack;
} KTHREAD;
```

18.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

1. Driver executes invalid instruction or memory access

2. CPU raises a fault (#PF, #GP, #UD, #DF)
3. IDT handler transfers control to `KiDispatchException`
4. Trap frame is captured
5. IRQL contract is validated
6. Secure Kernel validates instruction origin
7. `KeBugCheckEx` is invoked
8. Memory dump is written
9. System halts

Assembly Entry to BugCheck

```
KiDispatchException:
    call    KeBugCheckEx
    hlt
```

18.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11:

- Writable-executable kernel memory immediately triggers:
  - `ATTEMPT_EXECUTE_OF_NOEXECUTE_MEMORY`
- Invalid indirect call targets cause:
  - `KERNEL_SECURITY_CHECK_FAILURE`
- Unauthorized DMA mappings cause:

- `DRIVER_VERIFIER_DMA_VIOLATION`

Secure Kernel enforces:

- Control-flow integrity
- Stack shadow protection
- Code page immutability

18.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Incorrect driver behavior can:

- Corrupt per-CPU scheduler queues
- Invalidate TLB entries incorrectly
- Destroy NUMA locality mappings
- Stall DPC pipelines
- Deadlock spinlock acquisition across cores

These errors frequently lead to:

- `DPC_WATCHDOG_VIOLATION`
- `CLOCK_WATCHDOG_TIMEOUT`

18.3.7 REAL Practical Examples (C / C++ / Assembly)

Illegal Memory Access at DISPATCH_LEVEL (BSOD)

```
VOID BadDpcRoutine(
    KDPC* Dpc,
    PVOID DeferredContext,
    PVOID SystemArgument1,
    PVOID SystemArgument2
)
{
    int* ptr = (int*)ExAllocatePoolWithTag(
        PagedPool,
        sizeof(int),
        'bad1'
    );

    *ptr = 0xDEADBEEF;  % Causes PAGE_FAULT_IN_NONPAGED_AREA
}
```

Stack Corruption via Assembly (External ASM)

```
global CorruptStack
CorruptStack:
    sub  rsp, 8
    ret
```

Leads to:

- KERNEL_STACK_INPAGE_ERROR
- UNEXPECTED_KERNEL_MODE_TRAP

Double Free Kernel Pool (BSOD)

```
PVOID p = ExAllocatePoolWithTag(NonPagedPoolNx, 64, 'dbl1');
ExFreePool(p);
ExFreePool(p); % Causes BAD_POOL_CALLER
```

18.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

```
!analyze -v
```

```
kv
```

```
!thread
!process 0 1
```

```
r
dt nt!_KTRAP_FRAME
```

```
!irql
```

18.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common exploitation primitives:

- Use-after-free on WDF objects
- IOCTL METHOD_NEITHER buffer overwrite
- Dangling MDL mappings
- Kernel stack pivoting

Windows 11 defenses:

- SMAP/SMEP

- HVCI
- Kernel Control Flow Guard
- Pool metadata encryption
- VTL1 page protection

Most BSOD-triggering bugs are now **non-exploitable by design**.

18.3.10 Professional Kernel Engineering Notes (Real-World Pitfalls & Rules)

- Never touch paged memory above PASSIVE_LEVEL.
- Never free memory you did not allocate.
- Never execute user pointers without probing.
- Never assume IRQL.
- Never mix KMDF and raw IRP completion.
- Never return without completing IRP or WDFREQUEST.
- Never modify kernel code pages.
- Never bypass framework synchronization.

All BSODs in Windows 11 represent **enforced architectural correctness**, not instability.

Chapter 19

DMA, MDL & Hardware Memory Access

19.1 Memory Descriptor Lists (MDL)

19.1.1 Precise Windows 11Specific Definition

A **Memory Descriptor List (MDL)** in Windows 11 is a kernel-resident data structure used by the Memory Manager and I/O Manager to describe a **virtual memory buffer** in terms of the **physical page frames** that back it. An MDL provides a secure, validated, and pageable-aware abstraction that allows kernel subsystems, drivers, DMA engines, and the I/O stack to operate on physical memory without directly trusting user-mode virtual addresses.
MDLs are mandatory for:

- Direct Memory Access (DMA)
- Zero-copy I/O
- Locked user buffers
- Scatter/Gather hardware transactions

19.1.2 Architectural Role Inside the Windows 11 Kernel

Within Windows 11, MDLs form the **fundamental translation layer** between:

- User-mode virtual memory
- Kernel-mode virtual memory
- Physical page frames
- IOMMU DMA remapping domains (VT-d / AMD-Vi)

The MDL is consumed by:

- **I/O Manager** during IRP processing
- **Memory Manager** during page locking and mapping
- **HAL** during DMA adapter programming
- **Secure Kernel (VTL1)** for DMA isolation under VBS

MDLs also define whether pages are:

- Pageable
- Locked
- Mapped into system space
- Accessible by DMA engines

19.1.3 Internal Kernel Data Structures

The real Windows 11 MDL structure (ntoskrnl.exe):

```
typedef struct _MDL {
    struct _MDL *Next;
    SHORT Size;
    SHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

Key flag fields (subset):

```
#define MDL_MAPPED_TO_SYSTEM_VA     0x0001
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004
#define MDL_PAGES_LOCKED           0x0002
#define MDL_IO_PAGE_READ           0x0008
```

The actual PFN array is stored immediately after the MDL header.

19.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 User buffer submitted via IRP

User-mode buffer enters kernel via:

- `NtReadFile`
- `NtDeviceIoControlFile`

Step 2 MDL allocation

```
PMDL mdl = IoAllocateMdl(
    UserBuffer,
    BufferLength,
    FALSE,
    FALSE,
    Irp
);
```

Step 3 Page locking

```
MmProbeAndLockPages(
    mdl,
    UserMode,
    IoReadAccess
);
```

Step 4 System mapping (optional)

```
PVOID sysVa = MmGetSystemAddressForMdlSafe(
    mdl,
    NormalPagePriority
);
```

Step 5 DMA translation through HAL

The HAL programs the DMA remapping hardware using PFN list from the MDL.

Step 6 Completion

```
MmUnlockPages(mdl);
IoFreeMdl(mdl);
```

19.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- All DMA mappings are validated through **Secure Kernel (VTL1)**
- IOMMU enforces isolation per-device
- MDL PFN lists are cross-validated with VTL1 memory ownership
- Unauthorized DMA attempts trigger machine-check level faults

This prevents:

- DMA-based kernel memory corruption
- PCIe DMA rootkits
- Firmware-level memory scraping

19.1.6 Performance Implications

MDL operations impact performance in the following ways:

- Page locking causes TLB shootdowns
- Scatter/Gather MDLs increase IOMMU descriptor overhead
- Large MDLs reduce cache locality
- Frequent MDL allocation causes NonPagedPool pressure

Best performance occurs when:

- Buffers reside in NonPagedPool
- MDLs are reused
- Scatter/Gather lists remain short

19.1.7 Real Practical Example (KMDF)

```
VOID EvtIoRead(
    WDFQUEUE Queue,
    WDFREQUEST Request,
    size_t Length
)
{
    PMDL mdl;
    NTSTATUS status;

    status = WdfRequestRetrieveOutputWdmMdl(
        Request,
        &mdl
    );

    if (NT_SUCCESS(status)) {
        PVOID sysVa = MmGetSystemAddressForMdlSafe(
            mdl,
            NormalPagePriority
        );

        RtlZeroMemory(sysVa, Length);
    }

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

19.1.8 Kernel Debugging & Inspection (WinDbg)

```
!mdl <MDL_ADDRESS>
!pte <VirtualAddress>
!memusage
```

```
!iommu
```

Dump raw PFN array:

```
dt _MDL <MDL_ADDRESS>
db <MDL_ADDRESS + sizeof(_MDL)>
```

19.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Common exploitation techniques involving MDLs:

- Double-unlock of MDL pages
- MDL reuse after free
- Malicious MappedSystemVa redirection
- Forged PFN arrays in vulnerable drivers

Security boundaries:

- VTL1 DMA enforcement
- HVCI code integrity on HAL
- IOMMU remapping isolation

19.1.10 Professional Kernel Engineering Notes

- Never trust user MDL content in custom IOCTL paths
- Always match MmProbeAndLockPages with MmUnlockPages
- Never free MDL before IRP completion
- Avoid large MDLs in DISPATCH_LEVEL paths
- Always use MmGetSystemAddressForMdlSafe instead of unsafe mapping

19.2 Direct Memory Access (DMA)

19.2.1 Precise Windows 11Specific Definition

Direct Memory Access (DMA) in Windows 11 is a hardware-assisted data transfer mechanism that allows peripheral devices to read from or write to system memory **without continuous CPU intervention**. All DMA transactions are mediated by the Windows 11 **DMA subsystem**, the **HAL**, and the **IOMMU** (Intel VT-d / AMD-Vi) under strict enforcement by the **Secure Kernel (VTL1)** when VBS is enabled.
In Windows 11, DMA is **never raw physical access**. Every transaction is:

- Translated through the IOMMU
- Validated against MDL-derived PFN lists
- Isolated per-device via remapping domains
- Enforced by Secure Kernel DMA policies

19.2.2 Exact Architectural Role Inside the Windows 11 Kernel

DMA is implemented as a cooperative pipeline spanning:

- **I/O Manager** IRP-based I/O orchestration
- **Memory Manager** page locking and PFN extraction via MDLs
- **HAL.dll** DMA adapter programming
- **IOMMU Hardware** physical address translation and isolation
- **Secure Kernel (VTL1)** DMA security policy enforcement

The DMA engine never sees raw physical memory directly. All addresses are:

- Virtualized
- Remapped
- Permission-scoped

19.2.3 Internal Kernel Data Structures (REAL STRUCTS)

Core DMA-related kernel structures include:

DMA Adapter Object:

```
typedef struct _DMA_ADAPTER {
    USHORT Version;
    USHORT Size;
    struct _DMA_OPERATIONS *DmaOperations;
} DMA_ADAPTER, *PDMA_ADAPTER;
```

Common Buffer Descriptor:

```
typedef struct _COMMON_BUFFER_HEADER {
    SLIST_ENTRY ListEntry;
    PVOID VirtualAddress;
    PHYSICAL_ADDRESS LogicalAddress;
    ULONG Length;
} COMMON_BUFFER_HEADER;
```

Scatter/Gather DMA List:

```
typedef struct _SCATTER_GATHER_ELEMENT {
    PHYSICAL_ADDRESS Address;
    ULONG Length;
    ULONG Reserved;
} SCATTER_GATHER_ELEMENT, *PSCATTER_GATHER_ELEMENT;
```

These are consumed strictly through **HAL DMA operation tables**.

19.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 IRP arrives with MDL

DMA always originates from an IRP that already contains a locked MDL.

Step 2 DMA adapter acquisition

```
PDMA_ADAPTER adapter;
ULONG mapRegisters;

adapter = IoGetDmaAdapter(
    PhysicalDeviceObject,
    &deviceDescription,
    &mapRegisters
);
```

Step 3 Map transfer via HAL

```
PHYSICAL_ADDRESS logicalAddress;

logicalAddress = adapter->DmaOperations->MapTransfer(
    adapter,
    mdl,
    mapRegisterBase,
    currentVa,
    &length,
    TRUE
);
```

Step 4 Hardware DMA engine executes transfer

The device performs:

- PCIe bus mastering
- IOMMU-translated memory access

- Secure Kernelvalidated physical transfers

Step 5 DMA completion interrupt

ISR signals completion and halts the DMA channel.

Step 6 Unmap transfer

```
adapter->DmaOperations->FlushAdapterBuffers(
    adapter,
    mdl,
    mapRegisterBase,
    currentVa,
    length,
    TRUE
);
```

19.2.5 Secure Kernel / VBS / HVCI Interaction

With Windows 11 VBS enabled:

- DMA accesses are restricted to Secure Kernelapproved PFN ranges
- Each PCIe device receives a **separate IOMMU translation domain**
- DMA remap tables are owned by VTL1
- Unauthorized DMA causes immediate IOMMU fault injection

This blocks:

- PCIe DMA attacks
- Thunderbolt memory scraping
- Malicious firmware DMA injections

19.2.6 Performance Implications

DMA performance is influenced by:

- IOMMU TLB pressure
- Scatter/Gather entry count
- Cache coherency snooping overhead
- NUMA node placement of DMA buffers

Best performance is achieved by:

- Using contiguous common buffers
- Pinning memory on the same NUMA node as the device
- Minimizing MDL fragmentation

19.2.7 Real Practical Example (KMDF DMA Setup)

```
WDF_DMA_ENABLER_CONFIG dmaConfig;
WDFDMAENABLER dmaEnabler;

WDF_DMA_ENABLER_CONFIG_INIT(
    &dmaConfig,
    WdfDmaProfileScatterGather64,
    MAX_TRANSFER_SIZE
);

status = WdfDmaEnablerCreate(
    Device,
    &dmaConfig,
```

```
    WDF_NO_OBJECT_ATTRIBUTES,
    &dmaEnabler
);
```

Start a DMA transaction:

```
status = WdfDmaTransactionInitializeUsingRequest(
    dmaTransaction,
    Request,
    EvtProgramDma,
    direction
);
```

External x64 Assembly (DMA doorbell write example):

```
; File: dma_kick.asm (external, no inline ASM)
PUBLIC DmaKick

.code
DmaKick PROC
    sub rsp, 40h
    mov rcx, [rsp+48h]    ; MMIO base
    mov eax, 1
    mov [rcx], eax        ; Ring device doorbell
    add rsp, 40h
    ret
DmaKick ENDP
END
```

19.2.8 Kernel Debugging & Inspection (WinDbg)

```
!dma
!iommu
!hal
```

```
!devnode
!pcitree
```

Inspect MDL backing a DMA:

```
!mdl <MDL_ADDRESS>
!pte <VirtualAddress>
```

19.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historic DMA attack vectors:

- Thunderbolt DMA memory extraction
- PCIe bus mastering injection
- Malicious DMA firmware implants
- Forged MDL PFN lists in vulnerable drivers

Windows 11 protections:

- IOMMU mandatory enforcement
- Secure Kernel DMA ownership validation
- HVCI enforcement on HAL DMA code paths
- Driver Signature Enforcement on DMA-capable drivers

19.2.10 Professional Kernel Engineering Notes

- Never expose raw physical addresses to DMA hardware
- Always synchronize DMA completion with ISR/DPC

- Never reuse DMA map registers without full teardown
- Always use MDL-backed buffers for DMA
- Never perform DMA mapping above DISPATCH_LEVEL
- All DMA errors must be treated as fatal device faults

19.3 Secure DMA with IOMMU

19.3.1 Precise Windows 11Specific Definition

In Windows 11, **Secure DMA** is the enforcement of **hardware-isolated Direct Memory Access** through the **IOMMU** (Intel VT-d / AMD-Vi) under joint control of the **HAL** and the **Secure Kernel (VTL1)**. No PCIe or SoC-attached device is permitted to issue unrestricted physical memory transactions. Every DMA request is:

- Translated through per-device IOMMU page tables
- Validated against **MDL-derived PFN sets**
- Restricted to **device-specific DMA domains**
- Audited by **VBS and HVCI**

Secure DMA in Windows 11 is **mandatory** when:

- VBS is enabled
- Kernel DMA Protection is active
- A modern PCIe root complex with IOMMU exists

19.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Secure DMA enforcement spans five execution layers:

- **I/O Manager** issues IRPs that define data direction and access intent
- **Memory Manager** locks pages and builds MDLs
- **HAL.dll** programs DMA adapters and IOMMU mappings
- **Secure Kernel (VTL1)** validates DMA mapping ownership
- **IOMMU Hardware** enforces physical memory isolation

At no point does the device obtain unrestricted physical memory visibility. All DMA addresses are **remapped addresses**, not true physical addresses.

19.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

DMA Remapping Context (HAL-owned):

```
typedef struct _DMA_IOMMU_CONTEXT {
    PVOID Domain;
    ULONG DeviceId;
    ULONG ProtectionFlags;
} DMA_IOMMU_CONTEXT;
```

MDL PFN List Source for IOMMU Mapping:

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    PVOID Process;
    PVOID MappedSystemVa;
```

```
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

DMA Adapter Programming Interface:

```
typedef struct _DMA_OPERATIONS {
    PVOID PutDmaAdapter;
    PVOID AllocateCommonBuffer;
    PVOID FreeCommonBuffer;
    PVOID MapTransfer;
    PVOID FlushAdapterBuffers;
    PVOID GetDmaAlignment;
} DMA_OPERATIONS, *PDMA_OPERATIONS;
```

19.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 IRP arrives with locked MDL

User-mode buffer is validated and converted into an MDL with locked PFNs.

Step 2 Secure DMA domain allocation (HAL)

```
adapter = IoGetDmaAdapter(
    PhysicalDeviceObject,
    &deviceDescription,
    &mapRegisters
);
```

Step 3 IOMMU mapping creation

```
logicalAddress = adapter->DmaOperations->MapTransfer(
    adapter,
    mdl,
    mapRegisterBase,
```

```
    currentVa,
    &length,
    TRUE
);
```

At this point:

- Secure Kernel validates domain ownership
- IOMMU page tables are updated
- TLB invalidation is issued for DMA domains

Step 4 Device performs remapped DMA

The device only accesses remapped physical addresses, not real PFNs.

Step 5 DMA completion and domain teardown

```
adapter->DmaOperations->FlushAdapterBuffers(
    adapter,
    mdl,
    mapRegisterBase,
    currentVa,
    length,
    TRUE
);
```

19.3.5 Secure Kernel / VBS / HVCI Interaction

With Secure Kernel active:

- DMA domains are owned exclusively by VTL1
- DMA page tables are not writable from Ring 0
- All HAL DMA operations are HVCI-validated

- Device firmware DMA is sandboxed

Blocked attacks include:

- PCIe bus mastering memory dumps
- Thunderbolt cold-boot attacks
- Malicious NIC DMA implants
- DMA-based credential extraction

19.3.6 Performance Implications

Secure DMA introduces:

- IOMMU page walk overhead
- DMA TLB miss penalties
- NUMA-crossing penalties for remote memory
- Interrupt remapping latency

Performance optimizations include:

- Large contiguous DMA buffers
- NUMA-aware DMA allocation
- Scatter/gather minimization

19.3.7 Real Practical Example (Secure KMDF DMA)

```
WDF_DMA_ENABLER_CONFIG dmaConfig;

WDF_DMA_ENABLER_CONFIG_INIT(
    &dmaConfig,
    WdfDmaProfileScatterGather64,
    0x20000
);

dmaConfig.Flags |= WDF_DMA_ENABLER_CONFIG_REQUIRE_SECURE_DMA;

status = WdfDmaEnablerCreate(
    Device,
    &dmaConfig,
    WDF_NO_OBJECT_ATTRIBUTES,
    &dmaEnabler
);
```

External x64 Assembly (secure MMIO DMA doorbell trigger):

```
; secure_dma_kick.asm
PUBLIC SecureDmaKick

.code
SecureDmaKick PROC
    sub rsp, 40h
    mov rcx, [rsp+48h]    ; Secure-mapped MMIO
    mov eax, 1
    mov [rcx], eax
    add rsp, 40h
    ret
SecureDmaKick ENDP
END
```

19.3.8 Kernel Debugging & Inspection (WinDbg)

```
!iommu
!hal
!dma
!devnode
!pci
```

Inspect DMA remapped PFNs:

```
!mdl <MDL_ADDRESS>
!pte <RemappedAddress>
```

19.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Blocked attack classes:

- PCIe firmware DMA implants
- Thunderbolt hot-plug memory extraction
- DMA keyloggers
- Bus-mastering rootkits

Remaining attack surface:

- Signed vulnerable drivers with DMA privilege
- Firmware below IOMMU enforcement

19.3.10 Professional Kernel Engineering Notes

- Never assume physical addresses equal DMA addresses
- Always require Secure DMA for external hardware
- Treat DMA faults as security violations
- Never bypass HAL DMA operations
- Never map DMA buffers without MDLs
- Always expect IOMMU faults during early driver debugging

Part X

Native NTAPI & System Call Internals

Chapter 20

NTAPI from User Mode to Kernel Mode

20.1 System Service Dispatch Table (SSDT)

20.1.1 Precise Windows 11Specific Definition

In Windows 11, the **System Service Dispatch Table (SSDT)** is the kernel-resident dispatch structure used by the **NT system call fast path** to translate a **user-mode system call index** into the actual **kernel function pointer** that implements the service inside `ntoskrnl.exe`. Windows 11 uses:

- **KeServiceDescriptorTableShadow** (internally)
- **MSR-based syscall/sysret transition**
- **Per-build randomized SSDT layout**
- **HVCI-protected, read-only SSDT memory**

Direct SSDT patching is **blocked** under:

- VBS
- HVCI
- Kernel DMA Protection
- PatchGuard

20.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The SSDT is positioned in the **highest-frequency execution path of the entire OS**. Every NT system call passes through:

1. User-mode `ntdll.dll` syscall stub
2. CPU `SYSCALL` instruction
3. Kernel entry at `KiSystemCall64`
4. SSDT index decoding
5. Final kernel routine dispatch

It is therefore an:

- **Execution gate**
- **Security boundary**
- **Performance-critical structure**

20.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Windows 11 internally uses the following descriptor structure (not exported):

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR ServiceTableBase;
    PULONG ServiceCounterTableBase;
    ULONG_PTR NumberOfServices;
    PUCHAR ParamTableBase;
} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;
```

The shadow table exists for WoW64 and native separation:

```
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[];
```

Each SSDT entry contains:

- A **relative offset** to the real kernel function
- Encoded with **bit rotation and shift**

20.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 User-mode syscall stub in `ntdll.dll`

```
mov r10, rcx
mov eax, <SystemCallIndex>
syscall
ret
```

Step 2 CPU transition

The CPU jumps to the kernel entry point stored in:

```
IA32_LSTAR MSR
```

Step 3 Kernel dispatcher (`KiSystemCall64`)

- Stack switch to kernel stack
- GS base swap
- Security cookie validation
- Index range validation

Step 4 SSDT lookup

Internally:

```
ServiceAddress =
    KeServiceDescriptorTableShadow[0].ServiceTableBase[SystemCallIndex];
```

Step 5 Final dispatch

The resolved kernel routine is called using the Windows x64 ABI.

20.1.5 Secure Kernel / VBS / HVCI Interaction

With Secure Kernel enabled:

- SSDT memory is mapped read-only in VTL0
- Only Secure Kernel owns write permission
- All kernel images are verified by HVCI
- PatchGuard validates:
  - SSDT base
  - SSDT entry count

- SSDT code pointers

Any unauthorized modification results in:

- Immediate bug check
- Or delayed PatchGuard enforcement

20.1.6 Performance Implications

The SSDT path is optimized for:

- Single cache-line access
- Branch predictor stability
- Zero memory allocation
- Constant-time index resolution

Performance penalties occur when:

- EPT/IOMMU isolation is active
- IBRS/IBPB is enabled
- Kernel Shadow Stacks are enabled

20.1.7 Real Practical Example (User NTAPI to SSDT)

Native NTAPI Call from User Mode:

```
typedef NTSTATUS (NTAPI *NtQuerySystemInformation_t)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

NtQuerySystemInformation_t NtQuerySystemInformation =
    (NtQuerySystemInformation_t)
    GetProcAddress(GetModuleHandleW(L"ntdll.dll"),
                   "NtQuerySystemInformation");

NTSTATUS status = NtQuerySystemInformation(
    0x05,
    buffer,
    size,
    &returned
);
```

This call maps to:

- SSDT index resolved at runtime
- Kernel function `NtQuerySystemInformation`
- Internal execution as `ZwQuerySystemInformation`

External x64 Assembly syscall stub (VS2022-compatible):

```
; nt_syscall_stub.asm
PUBLIC NtSyscallStub

.code
NtSyscallStub PROC
```

```
    mov r10, rcx
    mov eax, edx
    syscall
    ret
NtSyscallStub ENDP
END
```

20.1.8 Kernel Debugging & Inspection (WinDbg)

```
x nt!KeServiceDescriptorTableShadow*
dd nt!KeServiceDescriptorTableShadow L4
!syscall
u nt!KiSystemCall64
```

To validate an SSDT entry:

```
dd nt!KeServiceDescriptorTableShadow+0x20
u <ResolvedKernelAddress>
```

20.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historic attack techniques (now blocked):

- SSDT hooking
- Shadow SSDT patching
- Inline syscall redirection

Modern attack surface:

- Vulnerable signed drivers
- DMA-based kernel memory overwrite (without IOMMU)

- Firmware-level syscall tampering

Security boundaries:

- SSDT is non-writable in VTL0
- PatchGuard validates SSDT integrity
- HVCI enforces signed kernel code only

20.1.10 Professional Kernel Engineering Notes

- Never attempt SSDT modification on Windows 11
- All syscall tracing must be done via:
  - ETW
  - Hypervisor-based monitoring
  - Kernel debugging instrumentation
- Always assume SSDT layout is randomized per build
- Syscall indices are **not stable across updates**
- Never hardcode SSDT offsets
- Never assume Zw == Nt internally under virtualization

20.2 Shadow SSDT

20.2.1 Precise Windows 11Specific Definition

In Windows 11, the **Shadow System Service Dispatch Table (Shadow SSDT)** is an internal kernel dispatch structure used exclusively to separate:

- Native 64-bit system calls
- WoW64 (32-bit) system calls

It is implemented as part of the internal structure:

- `KeServiceDescriptorTableShadow`

Unlike the primary SSDT, the Shadow SSDT is used for:

- 32-bit NTAPI transitions
- WoW64 syscall emulation path
- Cross-architecture validation layers

It is fully protected by:

- PatchGuard
- HVCI
- Secure Kernel (VTL1)

20.2.2 Exact Architectural Role Inside the Windows 11 Kernel

The Shadow SSDT acts as the **secondary syscall dispatch layer** used when:

- A 32-bit process executes under WoW64
- A compatibility NTAPI call is issued
- A legacy subsystem interacts with native kernel services

Execution path:

1. 32-bit user-mode stub inside `wow64.dll`
2. Transition into WoW64 layer
3. Translation to 64-bit syscall index
4. Dispatch via `KeServiceDescriptorTableShadow[1]`
5. Execution inside native `ntoskrnl.exe`

This prevents:

- Direct access to the native SSDT from 32-bit space
- Index collision across architectures

20.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

The Shadow SSDT uses the same descriptor layout as the primary SSDT:

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR ServiceTableBase;
    PULONG ServiceCounterTableBase;
    ULONG_PTR NumberOfServices;
    PUCHAR ParamTableBase;
} KSERVICE_TABLE_DESCRIPTOR;
```

Internal array:

```
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[2];
```

Index usage:

- `[0]` Native 64-bit SSDT

- `[1]` Shadow SSDT for WoW64

Each entry stores:

- Encoded function offset
- Argument count metadata

20.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 32-bit user-mode stub

```
mov eax, <32-bit Syscall Index>
call wow64cpu!X86SwitchTo64BitMode
```

Step 2 WoW64 transition

- CPU switches to long mode
- Syscall index is remapped
- Arguments are widened to 64-bit ABI

Step 3 Kernel entry

Control enters:

```
nt!KiSystemCall64
```

Step 4 Shadow SSDT lookup

```
ServiceAddress =
    KeServiceDescriptorTableShadow[1].ServiceTableBase[RemappedIndex];
```

Step 5 Native kernel execution

Kernel executes the resolved 64-bit service routine.

20.2.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 security:

- Shadow SSDT is mapped read-only in VTL0
- Write access exists only in VTL1
- All Shadow SSDT pointers are validated by:
  - PatchGuard
  - Secure Kernel Code Integrity
- Any corruption triggers:
  - Immediate bug check
  - Or delayed PatchGuard enforcement

WoW64 transitions themselves are monitored by:

- Hyper-V EPT controls
- CET shadow stacks (where enabled)

20.2.6 Performance Implications

Shadow SSDT introduces:

- One additional dispatch indirection
- One argument translation stage
- One mode-switch overhead (x86 x64)

Performance penalties are amplified when:

- IBRS mitigations are active
- Kernel Control-Flow Enforcement is enabled
- EPT-based address translation is enforced

This is why Windows 11 strongly favors:

- Native 64-bit applications

20.2.7 Real Practical Example (WoW64 NTAPI to Shadow SSDT)

32-bit user-mode syscall:

```
NTSTATUS status =
    NtQuerySystemInformation(
        SystemProcessInformation,
        buffer,
        size,
        &returnLength
    );
```

Runtime flow:

- Executed inside `ntdll.dll (32-bit)`
- Translated by `wow64.dll`
- Remapped into native syscall index
- Dispatched via Shadow SSDT

External x64 Assembly syscall stub used after WoW64 translation:

```
PUBLIC Wow64SyscallStub

.code
Wow64SyscallStub PROC
    mov r10, rcx
    mov eax, edx
    syscall
    ret
Wow64SyscallStub ENDP
END
```

20.2.8 Kernel Debugging & Inspection (WinDbg)

```
x nt!KeServiceDescriptorTableShadow*
dd nt!KeServiceDescriptorTableShadow L8
!syscall
```

To inspect Shadow SSDT:

```
dd nt!KeServiceDescriptorTableShadow+0x20
```

To trace WoW64 transitions:

```
u nt!KiSystemCall64
!wow64exts.sw
```

20.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Legacy attacks (no longer viable):

- Shadow SSDT hooking
- WoW64 syscall table overwrites

- 32-bit trampoline redirection

Modern attack surface:

- Vulnerable signed drivers modifying page tables
- Firmware-based syscall manipulation
- DMA attacks when IOMMU is disabled

Security boundaries:

- VTL1 isolation
- PatchGuard continuous validation
- HVCI mandatory code signing

20.2.10 Professional Kernel Engineering Notes

- Shadow SSDT must never be accessed directly by drivers
- WoW64 syscall indices are unstable across builds
- Never assume parity between SSDT and Shadow SSDT ordering
- Always perform syscall tracing using:
  - ETW
  - Hypervisor instrumentation
  - Kernel debug breakpoints
- All Shadow SSDT analysis must be treated as version-specific
- Any attempt to modify Shadow SSDT guarantees system crash

20.3 System Call Numbers

20.3.1 Precise Windows 11Specific Definition

In Windows 11, a **System Call Number** is the unique integer index used by the processor and kernel to resolve a user-mode NTAPI request into a specific kernel-mode service routine through the **System Service Dispatch Table (SSDT)** or the **Shadow SSDT**.
Each system call number represents:

- An offset into the active service table
- A strict ABI contract between `ntdll.dll` and `ntoskrnl.exe`
- A version-specific kernel entry index

System call numbers are:

- **Not stable across Windows versions**
- **Not stable across Windows 11 builds**
- **Different between native 64-bit and WoW64**

20.3.2 Exact Architectural Role Inside the Windows 11 Kernel

System call numbers provide the **numerical binding layer** between:

- User-mode NTAPI stubs in `ntdll.dll`
- Kernel dispatch via `KiSystemCall64`
- SSDT / Shadow SSDT resolution

Architectural usage path:

1. User-mode loads syscall number into `EAX`
2. `SYSCALL` instruction transfers control to kernel
3. Kernel validates the index
4. Index resolves into:
    - `KeServiceDescriptorTable[0]`
    - or `KeServiceDescriptorTableShadow[1]`
5. Target kernel service executes

This design provides:

- ABI separation between user and kernel
- Hotpatch flexibility
- Secure service gating

20.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

System call numbers index into:

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR ServiceTableBase;
    PULONG ServiceCounterTableBase;
    ULONG_PTR NumberOfServices;
    PUCHAR ParamTableBase;
} KSERVICE_TABLE_DESCRIPTOR;
```

Global tables:

```
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[2];
```

Index validation:

```
if (SyscallIndex >= KeServiceDescriptorTable.NumberOfServices)
    BugCheck(...);
```

20.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Step 1 User-mode syscall stub

Example for `NtClose` (x64):

```
mov r10, rcx
mov eax, 0x0F
syscall
ret
```

Step 2 CPU transition

- `SYSCALL` switches:
  - CPL 3 → CPL 0
  - User stack → kernel stack

Step 3 Kernel entry

Control arrives at:

```
nt!KiSystemCall64
```

Step 4 System call number extraction

```
movzx ecx, ax
```

Step 5 SSDT indexing

```
ServiceAddress =
    KeServiceDescriptorTable.ServiceTableBase[SyscallIndex];
```

Step 6 Kernel service execution

Control jumps directly to the resolved kernel function.

20.3.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces strict protection of system call numbers:

- SSDT and Shadow SSDT mapped read-only
- All syscall dispatch validated inside VTL0
- HVCI enforces:
  - No writable executable syscall targets
  - No runtime SSDT patching
- Secure Kernel (VTL1) monitors:
  - Indirect syscall redirection attempts
  - Page table permission tampering

Any corruption of:

- SSDT base
- Shadow SSDT base
- System call index validation path

results in:

- Immediate bug check
- Or deferred PatchGuard violation

20.3.6 Performance Implications

System call number handling introduces:

- Branch predictor pressure
- Speculative execution barriers
- IBRS overhead on Intel
- Indirect call validation on AMD Zen

Additional penalties apply when:

- CET is active
- Kernel shadow stacks are enabled
- Hyper-V EPT is enforcing syscall page permissions

Native 64-bit syscalls remain significantly faster than WoW64 syscalls due to:

- Lack of Shadow SSDT translation
- Direct ABI alignment

20.3.7 Real Practical Example (C / Assembly)

Native 64-bit user-mode syscall example

```
NTSTATUS status;
HANDLE h;

status = NtCreateEvent(
    &h,
    EVENT_ALL_ACCESS,
    NULL,
    NotificationEvent,
    FALSE
);
```

External x64 syscall stub (Visual Studio 2022 compatible)

```
PUBLIC NtCreateEventStub

.code
NtCreateEventStub PROC
    mov r10, rcx
    mov eax, 0x48
    syscall
    ret
NtCreateEventStub ENDP
END
```

20.3.8 Kernel Debugging & Inspection (WinDbg)

Inspect current SSDT base:

```
dd nt!KeServiceDescriptorTable
```

Inspect Shadow SSDT:

```
dd nt!KeServiceDescriptorTableShadow L4
```

Trace active syscall:

```
bp nt!KiSystemCall64
g
```

Disassemble target syscall handler:

```
u nt!NtCreateEvent
```

20.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Legacy attacks (deprecated):

- SSDT hooking
- Syscall index redirection
- Shadow SSDT overwrite

Modern attack surface:

- Vulnerable signed drivers modifying page tables
- DMA attacks without IOMMU
- Firmware-level syscall manipulation

Security boundaries enforcing protection:

- PatchGuard
- HVCI mandatory integrity checks
- Secure Kernel syscall validation
- Hypervisor-enforced EPT permissions

20.3.10 Professional Kernel Engineering Notes

- Never hardcode system call numbers in production drivers
- System call numbers change between Windows 11 builds
- WoW64 syscall numbers differ from native 64-bit
- Never attempt SSDT enumeration outside a debugger
- Syscall tracing must rely on:
  - WinDbg
  - ETW
  - Hypervisor instrumentation
- Any runtime syscall table modification guarantees system failure

20.4 Hooking & Detection

20.4.1 Precise Windows 11Specific Definition

In Windows 11, **system call hooking** refers to any unauthorized runtime modification of:

- The **System Service Dispatch Table (SSDT)**
- The **Shadow SSDT**
- The **NTAPI syscall entry stubs in `ntdll.dll`**
- The **kernel syscall entry path** (`KiSystemCall64`)

Detection refers to the multi-layered mechanisms implemented by Windows 11 to identify, prevent, and react to any deviation from the cryptographically verified syscall execution model using:

- PatchGuard
- Hypervisor-Protected Code Integrity (HVCI)
- Secure Kernel (VTL1)
- Extended Page Table (EPT) enforcement

Any form of syscall hooking in Windows 11 is **architecturally prohibited**.

20.4.2 Exact Architectural Role Inside the Windows 11 Kernel

Hooking detection occupies a **cross-boundary security enforcement role** spanning:

- User-mode syscall stubs
- Kernel-mode dispatch
- Hypervisor memory permission enforcement
- Secure Kernel integrity verification

Detection is enforced across the full syscall pipeline:

1. User-mode stub validation
2. Syscall instruction transition
3. Kernel SSDT resolution

4. Kernel target function execution
5. Return path verification

This prevents:

- API redirection
- Rootkit persistence
- Kernel privilege escalation
- EDR bypass

20.4.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Primary syscall integrity structures:

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR ServiceTableBase;
    PULONG ServiceCounterTableBase;
    ULONG_PTR NumberOfServices;
    PUCHAR ParamTableBase;
} KSERVICE_TABLE_DESCRIPTOR;
```

Secure Kernel validation target:

```
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[2];
```

PatchGuard internal verification regions include:

- SSDT base address
- Shadow SSDT base address
- `KiSystemCall64` code pages
- MSR `LSTAR` syscall entry address

20.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

Normal Execution Path

1. User loads syscall number into `EAX`
2. `SYSCALL` triggers CPL transition
3. CPU loads entry from MSR `LSTAR`
4. Kernel executes `KiSystemCall64`
5. SSDT lookup occurs
6. Target NT function executes
7. Control returns via `SYSRET`

Hooking Attempt Path

1. Attacker modifies SSDT entry
2. Secure Kernel integrity scan detects mismatch
3. PatchGuard delayed verification triggers
4. **Immediate SYSTEM_SERVICE_EXCEPTION or CRITICAL_STRUCTURE_CORRUPTION**

20.4.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces syscall integrity using:

- VTL1 Secure Kernel memory shadowing

- Hypervisor-enforced read-only kernel text
- Second-level EPT permission checks
- Runtime control-flow validation

Direct consequences of violation:

- Secure Kernel panic
- Immediate bugcheck before PatchGuard
- Hypervisor-enforced instruction abort

No kernel driver is permitted to modify:

- SSDT
- Shadow SSDT
- MSR `LSTAR`
- NTAPI dispatch handlers

20.4.6 Performance Implications

Hook-detection introduces:

- Additional EPT permission checks
- Secure Kernel memory verification
- IBRS / CET branch validation
- Return address validation on SYSRET

Modern Intel and AMD Zen microarchitectures mitigate overhead using:

- Speculation barriers
- Indirect branch predictors
- Microcode-assisted syscall validation

Performance impact remains below 36% under heavy syscall load.

20.4.7 Real Practical Example (C / Assembly)

External syscall stub (legal, unhooked)

```
PUBLIC NtQuerySystemTimeStub

.code
NtQuerySystemTimeStub PROC
    mov r10, rcx
    mov eax, 0x2D
    syscall
    ret
NtQuerySystemTimeStub ENDP
END
```

Detection Attempt Simulation (Read-only check)

```
PVOID base = KeServiceDescriptorTable.ServiceTableBase;
if (!MmIsAddressValid(base)) {
    DbgPrint("SSDT validation failure");
}
```

This detects:

- Invalid remapped tables
- DMA-injected SSDT overlays

20.4.8 Kernel Debugging & Inspection (WinDbg)

Check syscall entry MSR:

```
rdmsr 0xC0000082
```

Inspect SSDT base:

```
dd nt!KeServiceDescriptorTable
```

Verify `KiSystemCall64` integrity:

```
u nt!KiSystemCall64
!chkimg nt
```

Trace illegal writes:

```
ba w8 nt!KeServiceDescriptorTable
g
```

20.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

Blocked Attack Classes

- SSDT Hooking
- Shadow SSDT Manipulation
- MSR `LSTAR` Hijacking
- Syscall Stub Patching
- Inline NTAPI Patching

Remaining Attack Surfaces

- Firmware-level DMA injection
- Vulnerable signed driver page-table overwrite
- IOMMU bypass on misconfigured systems
- Hypervisor escape vulnerabilities

All remain subject to VBS and Secure Kernel monitoring.

20.4.10 Professional Kernel Engineering Notes

- Any syscall hooking guarantees eventual system crash
- PatchGuard is delayed, not absent
- HVCI blocks even signed drivers from syscall tampering
- Do not rely on SSDT for any kernel instrumentation
- Use ETW and hypervisor tracing for syscall inspection
- Never attempt inline hooking in Windows 11 kernel
- For malware research, operate only inside full nested hypervisors

Part XI

Kernel Debugging for Professionals

Chapter 21

WinDbg for Assembly & C Developers

21.1 Live Kernel Debugging

21.1.1 Precise Windows 11Specific Definition

Live kernel debugging in Windows 11 is the process of attaching a privileged kernel debugger to a running operating system instance at runtime in order to observe, control, inspect, and manipulate execution within `ntoskrnl.exe`, `HAL.dll`, device drivers, the scheduler, memory manager, interrupt handlers, and secure execution paths. Unlike crash-dump analysis, live debugging operates on an actively executing kernel with full access to register state, memory, processor topology, and Secure Kernel coordination state.

21.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The live kernel debugger is architecturally positioned:

- Outside the target OS instance

- Below Ring 0 control flow
- Above the hypervisor transport layer (KDNET, 1394, USB, COM)

WinDbg interacts directly with:

- Kernel exception dispatcher
- Interrupt routing logic
- System call transition layer
- Scheduler quantum control
- Virtual memory translation layer

Live debugging is therefore a **direct supervisory control channel** over the Windows 11 execution domain.

21.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

During live debugging, WinDbg directly inspects:

- `_KPRCB` Per-processor control block
- `_KTHREAD` Kernel thread structure
- `_EPROCESS` Process descriptor
- `_ETHREAD` Executive thread object
- `_KDPC` Deferred procedure call
- `_EXCEPTION_FRAME`

- `_TRAP_FRAME`
- Page table entries (PML4E, PDPTE, PDE, PTE)

These structures are read in real-time through kernel virtual memory translation.

21.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Live Debugging Activation Sequence

1. Windows boot loader initializes KD transport
2. Kernel enables debug trap gates in IDT
3. WinDbg establishes encrypted debug channel
4. Kernel enters debug listen state
5. Breakpoints generate debug interrupts
6. Execution is suspended on all logical processors

Breakpoint Entry Flow

1. INT3 instruction executed
2. CPU raises #BP exception
3. IDT routes to KiBreakpointTrap
4. Kernel suspends all other cores
5. WinDbg gains full register and memory control

21.1.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled:

- Debugging remains possible only in VTL0
- Secure Kernel memory is not directly writable
- HVCI prevents kernel page permission modification
- Hypervisor validates KD memory access requests

The debugger cannot:

- Modify Secure Kernel code
- Alter HVCI-enforced page protections
- Patch kernel text while HVCI is enabled

Live debugging remains observational in protected regions.

21.1.6 Performance Implications

Live kernel debugging introduces:

- Reduced interrupt throughput
- Forced global processor synchronization
- Increased DPC latency

However:

- No steady-state performance degradation occurs without active breakpoints
- No measurable TLB or cache penalty during normal execution

Performance impact is entirely breakpoint-driven.

21.1.7 Real Practical Example (C / C++ / Assembly)

Kernel Breakpoint on System Call Entry

```
bp nt!KiSystemCall64
g
```

Inspect Current Thread

```
!thread
```

Dump Current Process

```
!process 0 1
```

External Assembly Example (Single-Step Trap)

```
int3
nop
nop
```

This triggers immediate debugger entry when executed in kernel context.

21.1.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

CPU and PRCB State

```
!cpuinfo
dt nt!_KPRCB
```

Thread and Scheduling State

```
!thread
!running
```

Memory Translation

```
!pte <virtual_address>
```

Interrupt Descriptor Table

```
r idtr
```

21.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Live debugging is constrained by:

- Secure Boot enforcement
- KD authentication keys
- Hypervisor debug mediation
- HVCI page protection

Live debugging cannot be used to bypass:

- PatchGuard
- HVCI
- Secure Kernel isolation

Unauthorized debugger attachment requires:

- Physical access
- Boot policy modification
- Hypervisor compromise

21.1.10 Professional Kernel Engineering Notes

- Always enable kernel debugging before driver development
- Never rely on crash dumps alone for synchronization issues
- Always inspect `_TRAP_FRAME` on exception entry
- Avoid placing breakpoints inside high-frequency interrupt paths
- Live debugging with HVCI enabled is inherently read-only
- System responsiveness degrades under heavy breakpoint load
- Live debugging remains the only authoritative method for kernel execution verification

21.2 Memory Inspection

21.2.1 Precise Windows 11Specific Definition

Memory inspection in Windows 11 live kernel debugging is the controlled, real-time examination of virtual and physical memory regions belonging to the Windows 11 kernel, device drivers, kernel pools, page tables, and user-mode mappings as translated through the active paging hierarchy enforced by the Memory Manager and the Hyper-V hypervisor. This inspection is performed through the WinDbg KD engine using kernel-aware virtual-to-physical translation with full awareness of VTL0, VTL1, and HVCI memory ownership.

21.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Memory inspection operates at the intersection of the following architectural layers:

- Memory Manager (Mm)

- Virtual Address Descriptor (VAD) subsystem
- Hardware page-walk logic (CR3, PML4, PDPTE, PDE, PTE)
- Kernel pool allocator
- Hypervisor-enforced page access policies
- Secure Kernel memory ownership rules

WinDbg does not bypass the Memory Manager; instead, it queries memory through kernel debug transports that respect page ownership, permission bits, and VTL separation.

21.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

During memory inspection, WinDbg directly interprets the following real structures:

- `_EPROCESS`
- `_ETHREAD`
- `_MMVAD`
- `_MMPTE`
- `_MI_PHYSICAL_VIEW`
- `_POOL_HEADER`
- `_KPRCB`
- `_TRAP_FRAME`

Core paging fields include:

- `MMPTE.Hard.Valid`
- `MMPTE.Hard.Write`
- `MMPTE.Hard.ExecuteDisable`
- `MMPTE.Hard.PageFrameNumber`

21.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Live Memory Read Execution Path

1. WinDbg issues a memory read request
2. KD transport delivers the request to `ntoskrnl.exe`
3. Mm resolves the virtual address through the active PML4
4. Page table walk is executed using CR3
5. HVCI validation is applied to the PTE
6. Physical frame is mapped for debugger access
7. Memory contents are returned to WinDbg

Assembly-Level Address Resolution

```
mov     rax, cr3
and     rax, 0FFFFFFFFFFFFF000h
```

This identifies the base of the active page table used by the debugger.

21.2.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS and HVCI enabled:

- Kernel memory marked executable is write-protected
- Secure Kernel pages are not readable via WinDbg
- Hypervisor validates all debug memory requests
- VTL1 memory remains fully isolated

Attempted inspection of Secure Kernel memory results in blocked access or zeroed output depending on hypervisor policy.

21.2.6 Performance Implications

Memory inspection introduces:

- Temporary breakpoint-level execution stall
- Global TLB synchronization when stepping through page tables
- Interrupt latency increase proportional to inspection frequency

There is no persistent performance degradation when inspection is inactive.

21.2.7 Real Practical Example (C / C++ / Assembly)

Dump Kernel Memory at Specific Virtual Address

```
dq nt!MmNonPagedPoolStart L20
```

Dump Physical Page by Frame Number

```
!ddpfn <pfn>
```

Inspect a Kernel Object from C

```
EPROCESS* p = PsGetCurrentProcess();
KdPrint(("EPROCESS at %p\n", p));
```

External Assembly Memory Probe

```
mov     rax, [rcx]
nop
nop
```

This safely triggers debugger-visible memory access without modifying state.

21.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Raw Memory Dump

```
db <address>
dw <address>
dd <address>
dq <address>
```

Virtual-to-Physical Translation

```
!vtop <cr3> <virtual_address>
!pte <virtual_address>
```

Kernel Pool Inspection

```
!pool <address>
```

Process Address Space

```
!process 0 1
```

21.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Memory inspection is protected by:

- Secure Boot debug policy
- Hypervisor-enforced page ownership
- HVCI write protection
- PatchGuard page integrity validation

Malicious memory scraping requires:

- Debug policy bypass
- Hypervisor compromise
- Physical DMA attack

WinDbg itself cannot be used to bypass kernel memory protections under Windows 11 security policy.

21.2.10 Professional Kernel Engineering Notes

- Always validate page permissions with `!pte` before modifying memory
- Never assume pool memory contiguity
- Always inspect VAD layout before touching user memory
- Avoid repeated large memory scans on live systems
- Never attempt Secure Kernel inspection through KD

- Physical address inspection is blocked under HVCI
- Memory inspection remains the primary verification tool for pool corruption, stack corruption, and object lifetime analysis

21.3 Kernel Stack Tracing

21.3.1 Precise Windows 11Specific Definition

Kernel stack tracing in Windows 11 is the deterministic reconstruction of active and historical execution call chains within Ring 0 threads, using validated frame pointers, trap frames, and context records as maintained by `ntoskrnl.exe`, the scheduler, and the interrupt/exception dispatching mechanisms. It provides a time-consistent execution history across normal execution, system calls, APC/DPC delivery, page faults, and hardware interrupts under full Hyper-V and HVCI enforcement.

21.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Kernel stack tracing operates directly within the following architectural subsystems:

- Scheduler context switching (`KiSwapContext`)
- Interrupt dispatching (`KiInterruptDispatch`)
- System service dispatching (`KiSystemCall64`)
- Exception delivery (`KiDispatchException`)
- APC and DPC injection mechanisms

Each kernel thread maintains an independent kernel-mode stack allocated from NonPaged memory and mapped into the system address space.

21.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Stack tracing directly relies on these real internal structures:

- `_ETHREAD`
- `_KTHREAD`
- `_KTRAP_FRAME`
- `_CONTEXT`
- `_KPRCB`
- `_EXCEPTION_RECORD`

Critical fields used during stack reconstruction:

- `KTHREAD.StackBase`
- `KTHREAD.StackLimit`
- `KTHREAD.InitialStack`
- `KTRAP_FRAME.Rip`
- `KTRAP_FRAME.Rsp`
- `KTRAP_FRAME.Rflags`

21.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

System Call Stack Construction

1. User thread executes `syscall`
2. CPU switches to kernel stack using MSR-defined stack pointer
3. `KiSystemCall64` builds a `KTRAP_FRAME`
4. Service routine executes
5. Stack unwinding occurs during `sysret`

Assembly-Level Stack Entry (Conceptual)

```
swapgs
mov     rsp, [gs:0x1A8]
push    r11
push    rcx
```

This reflects the entry transition into the kernel stack on Windows x64.

21.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 security enforcement:

- Secure Kernel stacks (VTL1) are completely inaccessible to WinDbg
- Kernel stacks (VTL0) are readable but write-protected
- HVCI enforces executable integrity on all kernel stack return addresses
- Hypervisor validates context restoration during unwinding

Attempted manipulation of kernel return addresses causes immediate system termination.

21.3.6 Performance Implications

Kernel stack tracing introduces:

- Temporary scheduler freeze during live inspection
- TLB and instruction cache synchronization during unwinding
- No persistent performance degradation
- No context corruption when performed passively

Recursive unwinds on deep interrupt nesting may briefly delay interrupt dispatch.

21.3.7 Real Practical Example (C / C++ / Assembly)

Kernel Stack Dump of Current Thread

```
k
```

Full Stack with Parameters

```
kv
```

Verbose Stack with Frame Pointers

```
kp
```

C Kernel Stack Trace Trigger

```
KeBugCheckEx(0xDEADBEEF, 0, 0, 0, 0);
```

External Assembly Call Stack Anchor

```
call    SomeKernelRoutine
int3
```

This guarantees a debugger-visible stack frame transition.

21.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Basic Stack Trace

```
k
```

Stack with Arguments

```
kv
```

Stack with Frame Pointer Unwind

```
kp
```

Trap Frame Inspection

```
.trap <address>
```

Exception Frame Inspection

```
.exr <address>
```

Thread Stack Range

```
!thread
```

21.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Modern attack surfaces involving stack tracing include:

- Stack pivot attempts blocked by HVCI
- Return-oriented programming mitigated by CET and SMEP
- Trap frame corruption detected by PatchGuard
- Shadow stack violations detected by Intel CET

Kernel stack corruption invariably leads to immediate system crash under Windows 11.

21.3.10 Professional Kernel Engineering Notes

- Always validate stack bounds using `!thread`
- Never trust raw return addresses without symbol resolution
- Deep interrupt nesting complicates unwind reliability
- Always combine `k`, `kv`, and `kp` during crash analysis
- APC and DPC delivery always creates distinct stack frames
- Shadow stacks invalidate legacy exploitation techniques
- Stack traces are the primary truth source for IRQL violations, deadlocks, and driver misbehavior

Chapter 22

Crash Dump Analysis (Real BSOD Cases)

22.1 Bug Check Codes

22.1.1 Precise Windows 11Specific Definition

A **Bug Check Code** in Windows 11 is a 32-bit architectural stop identifier raised exclusively by `ntoskrnl.exe` or the Secure Kernel when the operating system enters a non-recoverable integrity or execution failure state. The bug check terminates all processor execution, freezes scheduling across all logical CPUs, and transitions the system into a controlled memory preservation and crash dump generation sequence.
Bug check invocation is performed through:

- `KeBugCheck`
- `KeBugCheckEx`
- Hypervisor-enforced fatal VM exits under VBS/HVCI

22.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Bug check codes serve as the terminal enforcement mechanism for:

- Kernel memory integrity
- Scheduler correctness
- IRQL enforcement
- Page table consistency
- Driver security boundaries
- Hypervisor trust isolation

Once triggered, execution transitions into:

- `KiBugCheckDispatch`
- `KeCapturePersistentThreadState`
- Crash dump writer in `ntoskrnl.exe`
- Optional Secure Kernel intervention (VTL1)

22.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Bug check analysis directly involves the following verified structures:

- `_KBUGCHECK_CALLBACK_RECORD`
- `_KTRAP_FRAME`
- `_EXCEPTION_RECORD`

- `_DUMP_HEADER`
- `_CONTEXT`
- `_KPRCB`
- `_KDDEBUGGER_DATA64`

Critical stored crash fields:

- Bug check code
- Four architecture-defined parameters
- Failing instruction pointer
- Active process and thread
- IRQL at time of failure

22.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel Bug Check Entry Flow

1. Fatal condition detected
2. `KeBugCheckEx(Code, P1, P2, P3, P4)` invoked
3. All CPUs halted via IPI
4. Current CPU captures `_KTRAP_FRAME`
5. Persistent thread state saved
6. Dump writer stores memory snapshot

7. System reset or halt occurs

Real C-Level Invocation

```
KeBugCheckEx(IRQL_NOT_LESS_OR_EQUAL, 0x1, 0x2, 0x3, 0x4);
```

External Assembly Fault Injection

```
int 3
hlt
```

This guarantees a controlled trap-frame-based bug check path when attached to KD.

22.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 VBS/HVCI enforcement:

- Secure Kernel validates crash origin (VTL0 vs VTL1)
- Kernel instruction integrity is revalidated before dump write
- HVCI blocks execution of any modified kernel code at bug check time
- Shadow stack integrity is verified under Intel CET

Attempted malicious triggering from unsigned drivers is blocked before crash propagation.

22.1.6 Performance Implications

Bug checks impose:

- Immediate global scheduler suspension
- Forced TLB shootdown across all processors

- Hypervisor-enforced state freeze
- No performance recovery since execution terminates

Live analysis performance impact is therefore irrelevant post-trigger.

22.1.7 Real Practical Example (C / C++ / Assembly)

Classic Driver-Originated Bug Check

```
if (Irql > DISPATCH_LEVEL)
{
    KeBugCheckEx(IRQL_NOT_LESS_OR_EQUAL, Irql, 0, 0, 0);
}
```

External Assembly Fault Trigger

```
mov     rax, 0
mov     qword ptr [rax], 1
```

This produces a deterministic `PAGE_FAULT_IN_NONPAGED_AREA`.

22.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Bug Check Identification

```
!analyze -v
```

Bug Check Code Only

```
.lastevent
```

Raw Parameters

```
r
```

Exception Record

```
.exr -1
```

Trap Frame

```
.trap -1
```

22.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Windows 11 blocks historical bug check exploitation via:

- SMEP / SMAP
- Intel CET shadow stacks
- HVCI code integrity
- PatchGuard verification cycles
- Hypervisor-enforced kernel isolation

Modern attacks must operate pre-crash or evade crash dispatch entirely, which is architecturally restricted.

22.1.10 Professional Kernel Engineering Notes

- Bug check parameters always encode architectural truth
- Parameter interpretation varies per code
- Never diagnose without full `!analyze -v`

- Trap frames outperform raw stack traces for crash origin
- Hypervisor crash origin differs fundamentally from VTL0 crashes
- Incorrect IRQL transitions remain the dominant real-world BSOD cause
- Memory corruption almost always manifests as delayed crash

22.2 Driver Fault Isolation

22.2.1 Precise Windows 11Specific Definition

Driver fault isolation in Windows 11 is the deterministic process of identifying the exact kernel-mode driver, execution path, and memory operation that triggered a system bug check. This isolation is performed using crash dump metadata captured by `ntoskrnl.exe`, validated by the Secure Kernel when VBS is enabled, and reconstructed through stack, trap, and IRQL state analysis.
Unlike heuristic debugging, Windows 11 driver isolation is based on:

- Verified instruction pointer provenance
- Verified stack unwinding under CET shadow stacks
- Hypervisor-validated memory access classification

22.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Driver fault isolation operates at the convergence of:

- I/O Manager dispatch control
- Memory Manager fault reporting

- Scheduler IRQL enforcement
- Secure Kernel telemetry (VTL1)

The architectural isolation boundary is enforced by:

- `KiDispatchException`
- `KeBugCheckEx`
- `KiProcessFault`
- Hypervisor crash classification logic

Faults are attributed strictly to the last executing non-Microsoft kernel module unless proven otherwise by memory corruption tracking.

22.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Driver fault isolation relies directly on the following real kernel structures:

- `_KTRAP_FRAME`
- `_EXCEPTION_RECORD`
- `_KTHREAD`
- `_EPROCESS`
- `_KPRCB`
- `_LDR_DATA_TABLE_ENTRY`
- `_DUMP_STACK_FRAME`

Key forensic fields:

- `RIP` exact faulting instruction
- `CR2` faulting virtual address
- `CurrentIrql`
- `StackBase / StackLimit`
- `ImageBase / ImageSize`

22.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Fault Isolation Execution Path

1. Driver issues invalid memory or IRQL operation
2. CPU raises page fault, GP fault, or interrupt violation
3. `KiDispatchException` captures trap frame
4. `KeBugCheckEx` invoked
5. Crash dump records:
    - Faulting RIP
    - Calling stack
    - Loaded module list
    - IRQL state
6. Secure Kernel validates code origin if VBS enabled

Representative Faulting Assembly

```
mov     rax, 0FFFF800000000000h
mov     rcx, qword ptr [rax]
```

This reliably triggers `PAGE_FAULT_IN_NONPAGED_AREA`.

22.2.5 Secure Kernel / VBS / HVCI Interaction

When VBS and HVCI are enabled:

- Faulting instruction is validated against hypervisor-maintained executable page tables
- Code origin is verified as:
  - Signed kernel image
  - Validly mapped executable page
- Any attempt to spoof module identity is rejected at VTL1
- Kernel stack corruption is verified against CET shadow stacks

This eliminates traditional post-crash driver spoofing techniques.

22.2.6 Performance Implications

Driver fault isolation imposes zero runtime overhead because:

- All forensic data is extracted post-mortem
- Shadow stack validation is always active under CET
- Hypervisor validation occurs only at fault time

There is no steady-state performance impact.

22.2.7 Real Practical Example (C / C++ / Assembly)

Classic Faulting Driver Code

```
void FaultRoutine(void)
{
    volatile int* Ptr = (int*)0xFFFFFFFFFFFFFFFF;
    *Ptr = 0x1234;
}
```

This isolates the faulting driver immediately through:

- Faulting RIP
- Module base lookup
- Instruction disassembly

External Assembly Fault Trigger

```
xor     rax, rax
mov     qword ptr [rax], 1
```

22.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Full Crash Analysis

```
!analyze -v
```

Faulting Module Isolation

```
lmvm <driver_name>
```

Call Stack

```
k
```

Trap Frame

```
.trap -1
```

Faulting Address

```
r cr2
```

22.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Modern Windows 11 eliminates classic fault obfuscation via:

- SMEP / SMAP
- HVCI executable validation
- Secure Kernel trust enforcement
- Shadow stack validation
- PatchGuard verification of kernel data

Attackers can no longer:

- Blind-stack spoof
- Fake return addresses
- Obfuscate crash ownership

22.2.10 Professional Kernel Engineering Notes

- The faulting instruction pointer is always authoritative
- Stack traces without trap frames are insufficient
- Deferred corruption often blames innocent drivers
- Direct IRQL violations are the fastest to isolate
- Memory pool corruption may surface minutes after cause
- Hypervisor-originated crashes require VTL-aware analysis
- Never trust the blamed module without validating CR2

22.3 Stack Corruption Detection

22.3.1 Precise Windows 11Specific Definition

Stack corruption detection in Windows 11 is the kernels capability to identify unauthorized modification of kernel-mode stack memory, return addresses, shadow stacks, or stack metadata during exception handling, interrupt dispatch, and system call execution. Windows 11 enforces stack integrity using hardware-assisted mechanisms including:

- Control-flow Enforcement Technology (CET)
- Supervisor Mode Execution Prevention (SMEP)
- Supervisor Mode Access Prevention (SMAP)
- Hypervisor-enforced stack validation (VBS)

Any violation of stack consistency is classified as a fatal kernel integrity breach and is resolved through immediate bug check invocation.

22.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Stack corruption detection is enforced across the following kernel execution paths:

- System call transitions (`syscall/sysret`)
- Interrupt dispatch (`KiInterruptDispatch`)
- Deferred Procedure Calls (DPC)
- Exception handling (`KiDispatchException`)
- Context switching (`KiSwapThread`)

Architecturally, the kernel guarantees that:

- Each kernel thread operates on a private, validated stack
- Stack pointers remain within `StackBase` and `StackLimit`
- Return addresses are validated against CET shadow stacks

Violations are escalated to `KeBugCheckEx` with corruption-specific codes.

22.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Stack corruption detection operates directly on these real kernel structures:

- `_KTHREAD`
- `_ETHREAD`
- `_KTRAP_FRAME`
- `_KSPECIAL_REGISTERS`

- `_CONTEXT`

Critical fields involved in detection:

- `StackBase`
- `StackLimit`
- `KernelStack`
- `RSP`
- `RIP`

CET shadow stack validation is performed in parallel using CPU-maintained return address stacks.

22.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Stack Corruption Detection Flow

1. Kernel thread executes at elevated IRQL
2. Corrupt write overwrites a return address or stack frame
3. Function executes `ret`
4. CPU verifies return address against CET shadow stack
5. Mismatch triggers a control protection fault
6. Fault propagates to `KiDispatchException`
7. `KeBugCheckEx` is invoked with a stack violation bug check

Example of Corrupting Stack via Assembly

```
sub     rsp, 40h
mov     qword ptr [rsp+38h], 0FFFFFFFFFFFFFFFFh
add     rsp, 40h
ret
```

This produces an invalid return target and triggers a CET validation failure.

22.3.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled:

- Shadow stacks are enforced at the CPU level
- Hypervisor verifies kernel stack page permissions
- Kernel stacks are non-executable and strictly guarded
- Stack pivot attacks are blocked at VTL1

Any mismatch between architectural stack and shadow stack causes an unrecoverable security exception.

22.3.6 Performance Implications

Stack corruption detection introduces:

- No measurable runtime overhead for validated returns
- One-cycle shadow stack validation per return instruction
- Zero scheduling overhead

All integrity checks are hardware-assisted and not software-polling based.

22.3.7 Real Practical Example (C / C++ / Assembly)

Fault-Inducing Kernel C Code

```
void CorruptStack(void)
{
    volatile unsigned long long* Ret;
    Ret = (unsigned long long*)__builtin_frame_address(0);
    Ret[1] = 0xFFFFFFFFFFFFFFFFULL;
}
```

This overwrites the return address and triggers a stack validation bug check.

External Assembly Stack Corruption

```
push    rax
mov     qword ptr [rsp], 0
ret
```

22.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Automatic Corruption Detection

```
!analyze -v
```

Trap Frame Inspection

```
.trap -1
```

Stack Bounds Validation

```
dt nt!_KTHREAD @$thread StackBase StackLimit
```

Current Stack Trace

```
k
```

Disassembly Around Fault

```
u @rip-40 @rip+40
```

22.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Classic kernel stack exploitation techniques are neutralized in Windows 11 via:

- Return-oriented programming prevention (CET)
- Stack pivot detection (SMAP + VBS)
- Stack execution blocking (NX)
- Kernel stack isolation per thread
- PatchGuard stack integrity verification

Attackers can no longer:

- Replace return addresses
- Chain kernel ROP gadgets
- Pivot kernel stacks into attacker-controlled memory

22.3.10 Professional Kernel Engineering Notes

- Stack corruption always manifests as secondary crashes
- CET eliminates silent return-address overwrites
- IRQL corruption often precedes stack failures
- Pool corruption frequently migrates into stack regions
- Shadow stack mismatches are always fatal
- Stack corruption bugs often implicate innocent drivers
- Manual frame pointer analysis is still required in optimized builds

Part XII

Reverse Engineering Windows 11 Kernel

Chapter 23

Reverse Engineering `ntoskrnl.exe`

23.1 Symbol Resolution

23.1.1 Precise Windows 11Specific Definition

Symbol resolution in Windows 11 kernel reverse engineering is the deterministic process of mapping **raw virtual addresses inside `ntoskrnl.exe`** to their corresponding **function names, data symbols, structure layouts, and type metadata** using Microsofts **PDB (Program Database)** symbol system.
In Windows 11, symbol resolution is **cryptographically bound to the exact kernel build**, including:

- Kernel build number
- Minor revision
- Security patch level
- HVCI state

- Secure Kernel version

Any symbol mismatch results in **structural misinterpretation and invalid reverse engineering conclusions**.

23.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Symbol resolution is not a runtime kernel feature; it is a **reverse engineering and debugging dependency layer** that enables:

- Accurate disassembly annotation
- Correct kernel data structure decoding
- Stack trace reconstruction
- IRP flow tracking
- PatchGuard and HVCI target identification

Architecturally, it bridges:

$$\textbf{Raw Kernel Memory} \longleftrightarrow \textbf{Human-Readable Kernel Semantics}$$

Without correct symbols, all reverse engineering of Windows 11 kernel behavior becomes **speculative**.

23.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Symbol resolution enables correct decoding of real kernel structures such as:

- `_EPROCESS`

- `_ETHREAD`
- `_KTHREAD`
- `_KPRCB`
- `_OBJECT_TYPE`
- `_HANDLE_TABLE`
- `_IRP`
- `_DRIVER_OBJECT`
- `_DEVICE_OBJECT`

Example verified fields resolved only via symbols:

- `_EPROCESS.ImageFileName`
- `_EPROCESS.Token`
- `_ETHREAD.Tcb`
- `_KPRCB.CurrentThread`

Without symbols, these offsets change between every Windows 11 build and cannot be reliably inferred.

23.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Symbol Resolution Pipeline in Live Kernel Debugging

1. WinDbg connects to KD over Hyper-V or COM

2. Kernel image base is enumerated
3. PDB GUID is extracted from PE header
4. Symbol server is queried
5. Verified PDB is downloaded
6. Type database is mapped
7. Live memory is decoded using PDB metadata

Disassembly Annotation Flow

1. Raw RIP address obtained from trap frame
2. Address mapped to nearest symbol
3. Function prototype reconstructed
4. Stack frame parsed using unwind metadata

This pipeline converts raw machine state into a **semantically valid kernel execution trace**.

23.1.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 introduces symbol visibility constraints due to:

- **VTL1 Secure Kernel isolation**
- **Hypervisor-enforced kernel code integrity**
- **Secure Kernel-only memory regions**

Consequences:

- VTL1 memory is invisible to Ring 0 debuggers
- Secure Kernel symbols are never publicly distributed
- HVCI blocks runtime modification of symbol-bearing code pages

Therefore, symbol resolution in Windows 11 applies only to:

- VTL0 (`ntoskrnl.exe`)
- HAL
- Loaded kernel drivers

23.1.6 Performance Implications

Symbol resolution itself has **zero runtime cost inside the operating system**. However, during live kernel debugging it incurs:

- Network I/O for PDB downloads
- Disk I/O for symbol cache
- Host-side memory overhead for type databases
- Disassembly annotation latency

Kernel execution performance is **not affected** by symbol resolution.

23.1.7 Real Practical Example (C / Assembly)

Kernel Symbol Verification from WinDbg

```
x nt!KeBugCheckEx
```

Disassemble with Resolved Symbol

```
u nt!KiSystemCall64
```

Structure Decoding Using Symbols

```
dt nt!_EPROCESS
dt nt!_ETHREAD
dt nt!_KTHREAD
```

External System Call Stub (User Mode, Symbol-Aligned)

```
PUBLIC NtQuerySystemTimeStub

.code
NtQuerySystemTimeStub PROC
    mov r10, rcx
    mov eax, 0x36
    syscall
    ret
NtQuerySystemTimeStub ENDP
END
```

This stub relies on **correct syscall number alignment**, which is resolved using symbol-aware reverse engineering.

23.1.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

Configure Official Symbol Server

```
.symfix
.reload
```

Verify Kernel Symbol Load

```
!sym noisy
.reload /f ntoskrnl.exe
```

Validate Symbol Integrity

```
!chkimg nt
```

Resolve Current Instruction Pointer

```
ln @rip
```

Stack Trace with Full Symbol Resolution

```
k
kv
```

23.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Symbol Resolution Does NOT Enable Exploitation By Itself

However, it enables precise targeting for:

- Kernel vulnerability research
- PatchGuard surface mapping
- HVCI validation boundary identification

- Driver security auditing

Blocked in Windows 11

- Symbol-based kernel patching
- Symbol-driven SSDT modification
- Live code redirection via resolved addresses

These are blocked by:

- PatchGuard
- HVCI
- EPT-based code immutability

23.1.10 Professional Kernel Engineering Notes

- Windows 11 symbols are strictly build-locked
- Never reuse symbols across different kernel revisions
- Always verify symbol load using `!chkimg`
- Secure Kernel symbols are never public
- Symbol errors invalidate all reverse engineering conclusions
- Use private symbol caches for forensic reproducibility
- Never infer offsets without validated PDBs
- Windows 11 introduces silent structure expansion across updates

23.2 Disassembly

23.2.1 Precise Windows 11Specific Definition

Kernel disassembly in Windows 11 is the deterministic reconstruction of **x86-64 machine instructions executed by `ntoskrnl.exe`** into their corresponding assembly representations under the constraints of:

- Windows 11 virtual memory layout
- SMEP, SMAP, and UMIP enforcement
- HVCI-backed code immutability
- PatchGuard integrity zones
- VTL0 vs VTL1 execution separation

Disassembly in Windows 11 is not merely static decoding; it must account for **runtime-enforced execution boundaries and hypervisor-backed memory protections**.

23.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Disassembly provides direct visibility into:

- System call transition logic (`syscall/sysret`)
- IRQL transitions
- APC/DPC dispatch paths
- Scheduler context switching

- Memory manager fault handlers
- Driver I/O dispatch paths

Architecturally, disassembly is the only authoritative method to:

Validate the real execution behavior of the Windows 11 kernel

independent of documentation, symbols, or headers.

23.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Disassembly directly reveals access patterns to verified Windows 11 structures such as:

- `_KTRAP_FRAME`
- `_KTHREAD`
- `_ETHREAD`
- `_EPROCESS`
- `_IRP`
- `_KPRCB`

Typical instruction patterns extracted via disassembly:

- `gs:[188h]` → current `_KTHREAD`
- `gs:[180h]` → current `_KPCR`
- `[rcx+offset]` → structure field dereference

These offsets are build-dependent and only validated through real kernel disassembly.

23.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Live Kernel Disassembly Pipeline

1. WinDbg attaches to KD session
2. RIP is captured from trap frame
3. Virtual address is translated via active page tables
4. SMEP/SMAP permissions are validated
5. Instruction bytes are fetched
6. x86-64 decoder reconstructs instruction semantics
7. Control-flow targets are resolved using symbols and RIP-relative displacement

System Call Entry Path (High-Level)

1. User-mode stub loads syscall number into `EAX`
2. `SYSCALL` instruction executed
3. CPU jumps to MSR `IA32_LSTAR`
4. `KiSystemCall64` executes
5. Thread is switched to kernel stack
6. SSDT entry is dispatched

All of the above is verified solely through **disassembly**.

23.2.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces:

- **Executable kernel code pages as read-only under HVCI**
- **Secure Kernel (VTL1) code never directly disassembled from VTL0**
- **EPT-backed execute-only code regions**

Consequences:

- Live patching of disassembled instructions is blocked
- Control-flow redirection into Secure Kernel is impossible from Ring 0
- Disassembly of Secure Kernel relies only on hypervisor forensic extraction

Thus, Windows 11 disassembly is constrained to **VTL0 kernel execution only**.

23.2.6 Performance Implications

Kernel disassembly itself does not impact runtime performance. However:

- Frequent single-stepping drastically increases VM-exit frequency
- Trap handling during debugging induces scheduler distortion
- HVCI page validation increases instruction fetch latency under debugging
- Disassembly inside ISR or DPC context heavily impacts interrupt latency

Therefore, real-world kernel disassembly must avoid:

- Long-running breakpoints in hot paths
- Single-step tracing in scheduler and memory fault paths

23.2.7 Real Practical Example (C / Assembly)

Live Kernel Disassembly of System Call Entry

```
u nt!KiSystemCall64
```

Disassemble IRP Dispatch Entry

```
u nt!IofCallDriver
```

External Assembly Stub (Windows x64 ABI, User Mode)

```
PUBLIC NtReadFileStub

.code
NtReadFileStub PROC
    mov r10, rcx
    mov eax, 0x3F
    syscall
    ret
NtReadFileStub ENDP
END
```

Disassembly is used to:

- Validate the MSR target
- Confirm syscall frame layout
- Verify stack shadow space preservation

23.2.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

Disassemble at Current RIP

```
u @rip
```

Disassemble with Symbol Resolution

```
uf nt!KiDispatchInterrupt
```

Single-Step One Instruction

```
t
```

Trace Until Return

```
pt
```

Display Active Trap Frame

```
.trap
```

23.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Disassembly reveals the exact enforcement points for:

- SMEP checks
- SMAP enforcement
- PatchGuard verification routines
- HVCI page validation
- Indirect branch validation

However, Windows 11 blocks:

- SSDT execution redirection
- Kernel inline hooks
- ISR patching
- Driver dispatch table rewriting

via:

- PatchGuard
- HVCI
- EPT-based execute-only memory

Disassembly is therefore a **research and validation tool**, not a reliable exploitation primitive.

23.2.10 Professional Kernel Engineering Notes

- Always disassemble with correct symbols loaded
- Never trust static offsets without live verification
- Avoid disassembling Secure Kernel assumptions
- Validate RIP-based control flow under KASLR
- Never place breakpoints inside hot scheduler paths
- Expect instruction reordering under HVCI validation
- Use instruction tracing sparingly in Windows 11
- Windows 11 frequently changes internal inlining patterns between updates

23.3 Function Flow Reconstruction

23.3.1 Precise Windows 11Specific Definition

Function flow reconstruction in Windows 11 is the deterministic recovery of the exact execution graph of a kernel routine inside `ntoskrnl.exe`, including:

- Entry sequence and prologue logic
- Control-flow transitions and indirect branches
- Conditional paths
- Call targets and inlining boundaries
- Exit sequence and epilogue logic

In Windows 11, this reconstruction must explicitly account for:

- KASLR randomization
- Control Flow Guard (CFG)
- HVCI-backed execute-only memory
- PatchGuard integrity zones
- VTL0 vs VTL1 execution isolation

23.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Function flow reconstruction is the only method to:

- Validate real scheduler transitions

- Confirm memory manager fault paths
- Verify IRP dispatch lifecycles
- Analyze system call entry and exit sequences
- Track APC and DPC delivery paths

Architecturally, it enables:

True execution verification beyond symbols and documentation

and exposes compiler-driven optimizations that do not exist at the source level.

23.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Function flows explicitly traverse real Windows 11 structures, including:

- `_KTRAP_FRAME` system call and interrupt state
- `_KTHREAD` scheduling, APC, IRQL state
- `_ETHREAD` executive thread management
- `_EPROCESS` process ownership and address space
- `_KPRCB` per-core scheduling state
- `_IRP` I/O request propagation

Typical instruction patterns observed during reconstruction:

- `gs:[188h]` → `CurrentThread`
- `gs:[180h]` → `KPCR`
- `[rcx+offset]` → structure field dereference
- `lock inc qword ptr [...]` → reference counting

23.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Reconstruction Pipeline

1. Disassemble target function entry
2. Identify stack frame setup
3. Track register parameter propagation
4. Resolve all direct and indirect call targets
5. Recover conditional branch relationships
6. Build directed acyclic control-flow graph
7. Validate exit paths and stack cleanup

Example: Executive Function Call Chain

1. `NtReadFile` user stub
2. `KiSystemCall64`
3. `KiServiceInternal`
4. `IofCallDriver`
5. Driver dispatch routine
6. Completion routine
7. `KeSetEvent`
8. Return to user mode

This chain is recoverable only through full execution flow reconstruction.

23.3.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 enforces:

- HVCI-backed execute-only kernel pages
- Secure Kernel execution in VTL1
- Hypervisor-controlled EPT permissions
- Indirect call enforcement

Implications for reconstruction:

- Secure Kernel functions cannot be stepped from VTL0
- Inline patching during tracing is blocked
- Indirect branch targets are validated by the hypervisor
- CFG tables restrict call target resolution

Thus, full flow reconstruction is constrained to **VTL0 kernel execution only**.

23.3.6 Performance Implications

Live flow reconstruction introduces:

- High VM-exit rates during single-stepping
- Scheduler distortion under trap-heavy tracing
- Cache pollution from repeated instruction fetches
- Increased TLB pressure during breakpoint activity

Professional-grade analysis therefore requires:

- Selective tracing of cold paths
- Avoidance of scheduler hot loops
- Offline graph reconstruction when possible

23.3.7 Real Practical Example (C / Assembly)

Reconstructing the Flow of `IofCallDriver`

```
uf nt!IofCallDriver
```

Tracking the Next Branch Target

```
u @rip L40
```

External User-Mode System Call Stub (Windows x64 ABI)

```
PUBLIC NtReadFileStub
.code
NtReadFileStub PROC
    mov     r10, rcx
    mov     eax, 3Fh
    syscall
    ret
NtReadFileStub ENDP
END
```

This stub enables flow tracing through:

- `SYSCALL` entry
- SSDT dispatch
- I/O manager execution path

23.3.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

Disassemble Full Function

```
uf nt!KiSystemCall64
```

Trace One Instruction

```
t
```

Trace Until Return

```
pt
```

View Current Trap Frame

```
.trap
```

Show Current Stack

```
k
```

23.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Reconstructed flows reveal:

- Exact PatchGuard validation paths
- Indirect call validation boundaries
- SMEP and SMAP enforcement locations
- Reference count enforcement sites

However, Windows 11 blocks:

- Inline kernel hooks
- SSDT replacement
- Dispatch table rewriting
- ISR and DPC patching

via:

- PatchGuard
- HVCI
- Hypervisor-backed memory protections

Function flow reconstruction is therefore strictly a **defensive and verification discipline**.

23.3.10 Professional Kernel Engineering Notes

- Never trust symbol-based flow without disassembly verification
- Always resolve indirect call tables manually
- Expect aggressive inlining in Windows 11 builds
- Control-flow graphs change between cumulative updates
- Avoid flow tracing in scheduler and memory fault hot paths
- Never attempt live patching during reconstruction under HVCI
- Always validate prologue and epilogue against ABI rules
- Treat Secure Kernel paths as opaque from VTL0

Chapter 24

PatchGuard, HVCI & Kernel Protections

24.1 How PatchGuard Works

24.1.1 Precise Windows 11Specific Definition

Kernel Patch Protection (PatchGuard) in Windows 11 is a **runtime integrity enforcement system** embedded inside `ntoskrnl.exe` and coordinated with the **Secure Kernel (VTL1)** and **Hyper-V hypervisor**. Its sole purpose is to **detect unauthorized modification of critical kernel code and data structures** and to **force an immediate system bugcheck upon violation**.
In Windows 11, PatchGuard is no longer a standalone kernel-only mechanism; it is **cryptographically coupled with HVCI and VBS**, meaning that its validation domain extends across:

- Ring 0 (Kernel Mode)
- VTL1 (Secure Kernel)
- Hypervisor-enforced memory translation

PatchGuard does **not** attempt remediation. It is a **pure detection-and-terminate system**.

24.1.2 Exact Architectural Role Inside the Windows 11 Kernel

PatchGuard occupies the following enforcement layers:

- Post-boot kernel self-verification
- Runtime delayed integrity validation
- Secure Kernelassisted attestation
- Hypervisor-backed write protection

It operates as a **distributed integrity system**:

1. Secure Kernel initializes integrity baselines during boot
2. Kernel embeds encrypted PatchGuard contexts
3. Hypervisor enforces write-protection on verified pages
4. Delayed validation tasks execute at unpredictable intervals

This architecture ensures **no deterministic timing window** exists for stable kernel tampering.

24.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

PatchGuard does **not expose public symbols**. However, it actively validates the integrity of the following **real kernel objects**:

- `KeServiceDescriptorTable`
- `KeServiceDescriptorTableShadow`

- `KiSystemCall64`
- `IDT` and `GDT`
- `MSR LSTAR`
- `PsLoadedModuleList`
- `HalDispatchTable`
- `Object Type Tables`

These structures are validated via **hashed snapshots** stored in **encrypted PatchGuard contexts** allocated from non-paged kernel pools and shadowed into VTL1.

24.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Boot-Time Phase

1. Secure Boot validates kernel image
2. Hypervisor initializes EPT protections
3. Secure Kernel establishes PatchGuard trust domain
4. PatchGuard encrypted contexts are seeded

Runtime Phase

1. Random delayed work item scheduled
2. Context is decrypted using per-boot entropy
3. Target kernel objects are re-hashed

4. Hashes compared with baseline
5. On mismatch → immediate `KeBugCheckEx`

Violation Path

1. Unauthorized write occurs
2. Hypervisor may allow transient modification
3. PatchGuard delayed validator executes
4. Integrity mismatch detected
5. **CRITICAL_STRUCTURE_CORRUPTION or ATTEMPTED_WRITE_TO_READONLY_MEMORY**

24.1.5 Secure Kernel / VBS / HVCI Interaction

Windows 11 elevates PatchGuard into a **multi-layer enforcement system**:

- **HVCI**: Enforces kernel code immutability using second-level EPT
- **VBS**: Isolates verification routines in VTL1
- **Secure Kernel**: Owns the integrity root-of-trust

Even a signed kernel driver:

- Cannot modify validated kernel code
- Cannot remap verified kernel data
- Cannot suppress PatchGuard timers

This eliminates traditional driver-based PatchGuard suppression techniques.

24.1.6 Performance Implications

PatchGuard introduces the following verified microarchitectural costs:

- Secure Kernel context switching
- Encrypted memory context handling
- TLB invalidation after Secure Kernel exit
- EPT permission verification

However, Windows 11 mitigates overhead using:

- Batched integrity scans
- Low-frequency randomized timers
- Deferred execution at passive IRQL

Observed overhead remains below **24%** under syscall-intensive workloads on modern Intel and AMD Zen systems.

24.1.7 Real Practical Example (C / Assembly)

Legitimate integrity verification from a kernel driver (read-only)

```
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;

VOID VerifySsdtReadOnly(VOID)
{
    volatile PVOID base = KeServiceDescriptorTable.ServiceTableBase;

    if (!MmIsAddressValid(base))
```

```
    {
        DbgPrint("SSDT address invalid\n");
    }
}
```

External syscall stub (legal execution path)

```
PUBLIC NtCloseStub

.code
NtCloseStub PROC
    mov r10, rcx
    mov eax, 0x0F
    syscall
    ret
NtCloseStub ENDP
END
```

These examples demonstrate **inspection without modification**, which remains fully PatchGuard-compliant.

24.1.8 Kernel Debugging & Inspection (WinDbg)

Verify PatchGuard-related protections:

```
!chkimg nt
```

Inspect syscall entry point:

```
rdmsr 0xC0000082
u nt!KiSystemCall64
```

Monitor illegal write attempts:

```
ba w8 nt!KeServiceDescriptorTable
g
```

Check Hypervisor & VBS state:

```
!sysinfo hypervisor
!sysinfo securekernel
```

24.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Directly Blocked by PatchGuard

- SSDT Hooking
- IDT Hooking
- Inline Kernel Function Patching
- MSR LSTAR Hijacking
- Kernel Object Type Table Modification

Remaining Attack Surface (Outside PatchGuard Scope)

- Firmware-level DMA before IOMMU initialization
- Hypervisor escape vulnerabilities
- CPU microcode flaws
- Signed-driver logic vulnerabilities

All of these remain subject to **independent mitigations via HVCI, IOMMU, and Secure Boot**.

24.1.10 Professional Kernel Engineering Notes

- PatchGuard is not a module it is a distributed enforcement system
- There is no supported mechanism to disable PatchGuard in Windows 11
- Any kernel tampering guarantees a delayed but unavoidable bugcheck
- HVCI eliminates historical signed-driver bypass paths
- Do not rely on undocumented kernel modification for instrumentation
- Use ETW, hypervisor tracing, and KD-assisted memory inspection instead
- PatchGuard timing is cryptographically randomized per boot

24.2 How PatchGuard Prevents Hooking

24.2.1 Precise Windows 11Specific Definition

In Windows 11, **PatchGuard (Kernel Patch Protection)** is a hypervisor-assisted, cryptographically obfuscated kernel self-integrity enforcement system that continuously verifies that `ntoskrnl.exe`, `HAL.dll`, kernel data structures, and critical dispatch tables remain in their original, trusted state.
PatchGuard in Windows 11 operates under the combined enforcement of:

- Virtualization-Based Security (VBS)
- Hypervisor-Protected Code Integrity (HVCI)
- Secure Kernel execution in VTL1

It prevents hooking by **detecting and halting any unauthorized modification of executable code and protected kernel data at runtime**.

24.2.2 Exact Architectural Role Inside the Windows 11 Kernel

PatchGuard acts as a **distributed, non-persistent, asynchronous kernel integrity governor**. Its architectural role is to:

- Enforce immutable kernel execution paths
- Prevent SSDT, IDT, GDT, and MSR-based hooks
- Protect kernel code sections from runtime modification
- Eliminate traditional rootkit persistence vectors

In Windows 11, PatchGuard is not a single thread or driver. It is implemented as:

- Hypervisor-coordinated validation callbacks
- Deferred work via encrypted DPC routines
- Secure Kernel verification routines in VTL1
- Randomized validation intervals

24.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

PatchGuard actively validates the integrity of the following real kernel structures:

- `_KSERVICE_TABLE_DESCRIPTOR` SSDT
- `KiSystemCall64` syscall entry stub
- `_IDT` interrupt descriptor table
- `_GDT` global descriptor table

- `_KPRCB` per-core control blocks
- `_KTHREAD` kernel thread metadata
- `_EPROCESS` process executive block
- Kernel code sections: `.text`, `.pdata`, `.rdata`

Validation operates through:

- Hash verification of executable pages
- MSR validation (`IA32_LSTAR`, `SYSENTER_EIP`)
- Descriptor table base checks
- Control-flow target verification

24.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

PatchGuard Validation Cycle

1. Encrypted PatchGuard context is dynamically decrypted at runtime
2. DPC callback is scheduled on a random logical processor
3. Secure Kernel validation routines execute in VTL1
4. Kernel memory hashes are recomputed
5. MSR and descriptor tables are validated
6. If mismatch is detected, a delayed bugcheck is armed
7. System halts with a controlled crash

Typical PatchGuard Crash Sequence

1. Integrity failure recorded
2. Silent corruption window
3. Deferred validation triggers
4. `KeBugCheckEx` executed

This delayed execution makes live forensic evasion ineffective.

24.2.5 Secure Kernel / VBS / HVCI Interaction

PatchGuard in Windows 11 is inseparably bound to:

- Secure Kernel operating in VTL1
- HVCI-backed execute-only kernel pages
- Hypervisor-enforced EPT permissions
- CFG (Control Flow Guard) enforcement

Security impact:

- Kernel code pages cannot be written even by Ring 0
- RWX memory cannot exist in kernel space
- Hook payload execution is blocked by EPT
- Hypervisor intercepts illicit page remapping

Thus, even if PatchGuard were bypassed, **HVCI enforces execution-level denial**.

24.2.6 Performance Implications

PatchGuard incurs:

- Minimal runtime overhead due to distributed deferred validation
- Minor DPC scheduling cost
- Negligible TLB pressure due to read-only verification

Windows 11 shifts most enforcement overhead into:

- Hypervisor VM-exit handling
- Secure Kernel validation paths

No measurable impact occurs on:

- User-mode performance
- I/O path throughput
- Scheduler fairness

24.2.7 Real Practical Example (C / C++ / Assembly)

Example: Traditional SSDT Hook Attempt (Blocked by PatchGuard)

```
extern PVOID *KeServiceDescriptorTable;

void HookSSDT(void)
{
    KeServiceDescriptorTable[0x3F] = (PVOID)0xDEADBEEF;
}
```

Result on Windows 11:

- Immediate integrity mismatch
- Deferred PatchGuard response
- BugCheck triggered within seconds to minutes

MSR Hook Attempt (Blocked by Hypervisor)

```
mov ecx, 0C0000082h   ; IA32_LSTAR
rdmsr
mov eax, 0DEADBEEFh
wrmsr
```

Result:

- Hypervisor intercept
- Immediate system termination

24.2.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

Verify Syscall Entry

```
rdmsr C0000082
```

Inspect SSDT

```
dd nt!KeServiceDescriptorTable L4
```

Check Kernel Memory Protection

```
!pte nt!KiSystemCall64
```

Check Secure Kernel State

```
!vm 4
```

24.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Windows 11 fully blocks:

- SSDT hooks
- Inline kernel detours
- IDT modification
- MSR syscall redirection
- DKOM-based function hijacking

Remaining theoretical attack surface:

- Hypervisor escape vulnerabilities (VTL1 breach)
- CPU microcode flaws
- DMA-based physical memory attacks (without IOMMU)

All traditional Ring 0 rootkit models are obsolete under Windows 11.

24.2.10 Professional Kernel Engineering Notes

- PatchGuard is not a single routineit is a distributed integrity mesh
- Delayed crash logic defeats time-of-check bypass techniques
- HVCI enforces execute-only memory at hardware level
- No reliable PatchGuard bypass exists on modern Windows 11
- Any detected hook attempt is a guaranteed system termination

- Kernel debugging on production systems requires PG-aware methodology
- The only legitimate kernel modifications occur through signed, CI-validated drivers
- Hypervisor dominance makes Ring 0 no longer authoritative

24.3 How PatchGuard Prevents Kernel Modification

24.3.1 Precise Windows 11Specific Definition

In Windows 11, **PatchGuard prevents kernel modification** by enforcing a continuously verified trust boundary over all executable kernel regions, control-transfer mechanisms, and privileged processor configuration registers. Kernel modification is defined as any unauthorized attempt to alter:

- Executable code inside `ntoskrnl.exe` and `HAL.dll`
- Protected kernel dispatch tables
- Kernel control MSRs
- Descriptor tables and scheduling control paths
- Secure kernel coordination structures

The enforcement model is **hardware-rooted, hypervisor-assisted, and cryptographically obfuscated**, making live kernel modification computationally infeasible on Windows 11.

24.3.2 Exact Architectural Role Inside the Windows 11 Kernel

PatchGuard operates as a **non-resident integrity authority** that is architecturally positioned across:

- Ring 0 Kernel Execution (VTL0)
- Secure Kernel Enforcement (VTL1)
- Hypervisor Page Permission Control (EPT / NPT)

Its architectural responsibility is to ensure that:

- Kernel text sections remain immutable after boot
- System call entry points remain invariant
- Interrupt routing logic remains unmodified
- Kernel memory pools cannot be repurposed for executable payloads

PatchGuard therefore acts as a **runtime kernel immutability enforcement system**.

24.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Windows 11 PatchGuard continuously verifies integrity of:

- `_KSERVICE_TABLE_DESCRIPTOR`
- `KiSystemCall64`
- `_IDT`, `_GDT`
- `_KPRCB`
- `_KTHREAD`
- `_EPROCESS`
- Kernel image PE sections: `.text`, `.pdata`, `.rdata`

Additionally, the following MSRs are validated:

- `IA32_LSTAR`
- `IA32_SYSENTER_EIP`
- `IA32_SYSENTER_ESP`
- `IA32_STAR`

Hash validation of these elements is performed inside Secure Kernel execution context (VTL1).

24.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel Modification Prevention Cycle

1. PatchGuard validation context is decrypted dynamically
2. A DPC is queued on a random logical processor
3. Secure Kernel enters validation mode in VTL1
4. Kernel code page hashes are recomputed
5. Descriptor tables and MSRs are revalidated
6. Any deviation arms a deferred bugcheck
7. `KeBugCheckEx` terminates the system

Write Attempt Interception Path

1. Driver attempts to modify protected kernel page

2. Hypervisor blocks write via EPT
3. PatchGuard later detects hash mismatch
4. Delayed system crash is triggered

This dual-layer design ensures both **write prevention and post-modification detection**.

24.3.5 Secure Kernel / VBS / HVCI Interaction

PatchGuard kernel modification prevention is enforced through:

- VTL1 Secure Kernel memory verifier
- HVCI-enforced execute-only kernel pages
- Hypervisor EPT permission enforcement
- Kernel Control Flow Guard (KCFG)

Windows 11 enforces:

- No RWX kernel memory
- No executable pool memory
- No remapping of kernel text pages
- No modification of syscall entry paths

Thus, kernel modification is blocked both by **hardware permission enforcement and delayed cryptographic detection**.

24.3.6 Performance Implications

PatchGuard kernel modification defense introduces:

- Minimal DPC scheduling overhead
- Minor Secure Kernel execution overhead
- Negligible impact on cache and TLB behavior

Because HVCI enforces permissions at the EPT layer, no performance penalty occurs during normal execution. The enforcement cost is paid only during validation cycles.

24.3.7 Real Practical Example (C / C++ / Assembly)

Example: Inline Kernel Function Patch Attempt

```
void PatchKernelFunction(void* target)
{
    unsigned char patch[5] = { 0xE9, 0, 0, 0, 0 };
    memcpy(target, patch, sizeof(patch));
}
```

Result on Windows 11:

- Write access blocked by EPT if HVCI enabled
- If write succeeds through race or exploit, PatchGuard detects mismatch
- Deferred system bugcheck occurs

Example: Syscall Entry MSR Modification (External Assembly)

```
mov ecx, 0C0000082h
rdmsr
mov eax, 0FFFFFFFFh
wrmsr
```

Result:

- Hypervisor intercepts write
- Immediate system termination

24.3.8 Kernel Debugging & Inspection (REAL WinDbg Commands)

Verify Kernel Text Page Protection

```
!pte nt!KeBugCheckEx
```

Inspect Syscall MSR

```
rdmsr C0000082
```

Verify Secure Kernel State

```
!vm 4
```

Check Descriptor Table Bases

```
r idtr
r gdtr
```

24.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Windows 11 blocks all classical kernel modification techniques:

- Inline hooks
- SSDT hooks
- IDT hooks
- MSR redirection
- DKOM-based control hijacking
- Runtime code patching

Only theoretical remaining vectors:

- Hypervisor escape
- CPU microcode vulnerabilities
- Physical DMA attacks without IOMMU

Ring 0 alone is no longer sufficient to modify Windows 11 kernel behavior.

24.3.10 Professional Kernel Engineering Notes

- PatchGuard does not rely solely on signature scanning
- Kernel modification is blocked both pre-write and post-write
- HVCI enforces immutable kernel memory at hardware level
- Delayed bugcheck prevents real-time bypass attempts

- Any persistent kernel patch guarantees system termination
- Legitimate kernel behavior modification is only possible through CI-signed drivers
- Hypervisor dominance makes traditional kernel rootkits obsolete
- Windows 11 kernel is no longer a modifiable trust domain

Part XIII

Kernel Security & Exploitation

Chapter 25

Windows 11 Kernel Attack Surface

25.1 Pool Overflows

25.1.1 Precise Windows 11Specific Definition

A **kernel pool overflow** in Windows 11 is a memory safety violation where a write operation exceeds the bounds of a heap allocation obtained from the kernel pool allocator (`ExAllocatePool2`, `ExAllocatePoolWithTag`), corrupting adjacent pool metadata or neighboring allocations. Windows 11 classifies pool overflows as **fatal integrity violations** due to their direct impact on kernel control structures, object headers, and execution flow. Windows 11 distinguishes between:

- **Paged Pool** pageable kernel heap, valid only at `IRQL < DISPATCH_LEVEL`
- **NonPaged Pool** permanently resident kernel heap, valid at any IRQL

Overflow in either pool compromises kernel integrity and is actively monitored by Pool Hardening, VBS, and HVCI.

25.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The pool allocator serves all core kernel subsystems:

- I/O Manager (IRPs, MDLs)
- Object Manager (object headers)
- Security Reference Monitor (tokens)
- Scheduler (KTHREAD extensions)
- Networking stack (NDIS buffers)
- File systems (FCB, CCB structures)

A pool overflow directly threatens:

- Object reference counting
- Linked list integrity
- Function pointer tables
- Security descriptor chains

Windows 11 places the pool subsystem under continuous integrity verification through:

- Kernel VA Shadow
- Pool cookie encoding
- Pool header checksum validation
- PatchGuard periodic verification

25.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Pool overflow detection interacts with the following real internal structures:

- `_POOL_HEADER`
- `_GENERAL_LOOKASIDE`
- `_EX_POOL_NODE`
- `_EX_PUSH_LOCK`

Critical `_POOL_HEADER` fields:

- `BlockSize`
- `PoolType`
- `PoolTag`
- `ProcessBilled`

Each allocation is surrounded by encoded metadata used for corruption detection.

25.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Overflow Detection Flow in Windows 11

1. Driver requests pool memory using `ExAllocatePool2`
2. Pool header is initialized with tag, size, and cookie
3. Driver writes beyond allocated boundary
4. Neighboring pool header or object structure is corrupted

5. Corruption is detected during:
    - Pool free
    - Deferred integrity scan
    - PatchGuard cycle
    - HVCI verification
6. Kernel invokes `KeBugCheckEx`

Typical Bug Checks Associated With Pool Overflows

- `BAD_POOL_HEADER`
- `POOL_CORRUPTION_IN_FILE_AREA`
- `DRIVER_CORRUPTED_EXPOOL`
- `KERNEL_SECURITY_CHECK_FAILURE`

25.1.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled in Windows 11:

- Pool memory is protected using second-level address translation (SLAT)
- Kernel VA Shadow separates user-mode and kernel-mode mappings
- Indirect control data derived from pool memory is validated at execution
- PatchGuard validates pool-linked kernel structures periodically

Any corrupted pool-derived function pointer or object header causes:

- Immediate VM-exit into the Hyper-V secure kernel
- Forced kernel crash to preserve system trust

25.1.6 Performance Implications

Pool hardening introduces:

- One-time cookie encoding per allocation
- Lightweight checksum verification on free
- Periodic PatchGuard scanning

These protections introduce **no measurable performance degradation** in production systems due to hardware-assisted virtualization.

25.1.7 Real Practical Example (C / C++ / Assembly)

Fault-Inducing Kernel C Code (DO NOT USE IN PRODUCTION)

```
void PoolOverflowDemo(void)
{
    PUCHAR buffer = (PUCHAR)ExAllocatePool2(
        POOL_FLAG_NON_PAGED,
        16,
        'OVFL'
    );

    for (int i = 0; i < 64; i++)
    {
        buffer[i] = 0x41;  // Intentional overflow
    }

    ExFreePool(buffer);
}
```

This causes corruption of the adjacent _POOL_HEADER and reliably triggers a pool corruption bug check.

External Assembly Write Past Allocation (Conceptual)

```
mov     rcx, rdi
add     rcx, 20h
mov     qword ptr [rcx], 0FFFFFFFFFFFFFFFFh
```

This represents an out-of-bounds write relative to the base allocation.

25.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Initial Analysis

```
!analyze -v
```

Inspect the Faulting Allocation

```
!pool @rax
```

Verify Pool Headers

```
!poolval
```

Display Corrupted Memory

```
dd @rax-20
```

Locate Owning Driver

```
lmv m faulty_driver
```

25.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically, pool overflows enabled:

- Arbitrary kernel write primitives
- Token privilege escalation
- Function pointer overwrites
- Object header manipulation

In Windows 11, the following defenses fully neutralize traditional exploitation:

- Pool header cookie encoding
- Kernel VA Shadow
- CET enforcement
- HVCI indirect call validation
- PatchGuard structure verification
- NX enforcement on pool memory

Modern pool overflows result in:

- Guaranteed system crash
- No reliable exploitation primitives

25.1.10 Professional Kernel Engineering Notes

- Pool overflows are **driver correctness failures**, not exploitation primitives on Windows 11
- Silent corruption is no longer possible under HVCI
- Pool corruption is frequently detected far from the origin
- Bad IRQL handling often accompanies pool corruption
- Use pool tagging consistently for crash attribution
- Enable Driver Verifier for immediate detection
- Memory sanitization must be enforced at build time

25.2 Use-After-Free (UAF)

25.2.1 Precise Windows 11Specific Definition

A **Use-After-Free (UAF)** vulnerability in Windows 11 kernel mode occurs when a pointer to a kernel object or pool allocation is dereferenced after the backing memory has already been released via `ExFreePool` or equivalent object dereference paths. The stale pointer retains an address that may later be reallocated for an unrelated kernel object, causing unintended type confusion, control corruption, or data disclosure.
In Windows 11, UAF conditions are classified as **temporal memory safety violations** and are actively mitigated by:

- Kernel pool zeroing
- Delayed free quarantine

- Pool header cookies
- Reference counting hardening
- VBS-backed integrity verification

25.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Use-after-free conditions arise across multiple kernel subsystems:

- Object Manager (stale object handles)
- I/O Manager (IRPs, MDLs)
- ALPC ports
- Windows Filtering Platform
- Networking stack (NET_BUFFER, NET_BUFFER_LIST)
- GUI subsystem kernel objects

UAF corruption directly impacts:

- Object reference counters
- VTable pointers
- Callback tables
- Linked list integrity

Windows 11 enforces strict object lifetime validation using:

- Encoded object headers

- Secure handle validation
- Reference count saturation checks
- Hypervisor-assisted execution integrity

25.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Use-after-free frequently corrupts or misuses the following real kernel structures:

- `_OBJECT_HEADER`
- `_KTHREAD`
- `_EPROCESS`
- `_IRP`
- `_MDL`

Critical `_OBJECT_HEADER` lifetime fields:

- `PointerCount`
- `HandleCount`
- `TypeIndex`
- `Flags`

Any stale reference to these headers after free results in guaranteed kernel integrity failure.

25.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Canonical UAF Failure Flow in Windows 11

1. Kernel driver allocates object from pool
2. Pointer is stored in global or structure field
3. Object is freed through `ExFreePool` or `ObDereferenceObject`
4. Pointer is not cleared
5. Memory is reallocated for a different kernel object
6. Original stale pointer is dereferenced
7. Type confusion or control corruption occurs
8. PatchGuard, VBS, or pool validation detects corruption
9. Kernel invokes `KeBugCheckEx`

Common Bug Checks Triggered

- `KERNEL_SECURITY_CHECK_FAILURE`
- `PAGE_FAULT_IN_NONPAGED_AREA`
- `BAD_OBJECT_HEADER`
- `DRIVER_OVERRAN_STACK_BUFFER`

25.2.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled:

- Stale pointers cannot redirect indirect kernel execution
- Function pointers loaded from freed memory fail HVCI validation
- Control-flow redirection is blocked via CET
- Hypervisor detects invalid EPT-based kernel execution

Windows 11 enforces:

- Virtual Trust Levels (VTL0 vs VTL1)
- Secure kernel policing of indirect calls
- Immediate system termination upon pointer forgery

25.2.6 Performance Implications

UAF hardening introduces:

- Extra reference count validation on dereference
- Memory poisoning on free
- Delayed free quarantine buffer
- Secure zeroing of freed nonpaged pool

Modern hardware acceleration ensures these mechanisms introduce **no perceptible system overhead**.

25.2.7 Real Practical Example (C / C++ / Assembly)

Fault-Inducing Kernel C Code (DO NOT USE IN PRODUCTION)

```
typedef struct _UAF_OBJ {
    ULONG Value;
} UAF_OBJ, *PUAF_OBJ;

PUAF_OBJ GlobalPtr;

void UafDemo(void)
{
    GlobalPtr = (PUAF_OBJ)ExAllocatePool2(
        POOL_FLAG_NON_PAGED,
        sizeof(UAF_OBJ),
        'UAF1'
    );

    GlobalPtr->Value = 0x1337;

    ExFreePool(GlobalPtr);

    // Use-after-free
    GlobalPtr->Value = 0x41414141;
}
```

This results in either immediate page fault or latent pool corruption detected later.

External Assembly Stale Write Pattern (Conceptual)

```
mov     rax, qword ptr [stale_ptr]
mov     dword ptr [rax], 41414141h
```

25.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Initial Crash Analysis

```
!analyze -v
```

Inspect the Faulting Address

```
r
dd @rax
```

Check Pool State

```
!pool @rax
```

Inspect Object Header

```
dt nt!_OBJECT_HEADER @rax-30
```

Locate Faulting Driver

```
lmv m faulty_driver
```

25.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically, UAFs enabled:

- Kernel arbitrary read/write
- Token privilege escalation
- Callback pointer overwrites
- Type confusion exploitation

In Windows 11, exploitation is neutralized by:

- Kernel VA Shadow
- HVCI enforced indirect calls
- Pool cookie encoding
- Automatic memory zeroing on free
- Secure reference counting
- PatchGuard verification
- CET shadow stack enforcement

Any reliable UAF exploitation primitive in Windows 11 is considered **architecturally infeasible without hypervisor compromise**.

25.2.10 Professional Kernel Engineering Notes

- UAFs are primarily lifetime management bugs, not exploitation vectors
- Always clear freed global pointers
- Object reference counting must be perfectly symmetric
- Never retain raw kernel object pointers across async execution
- Driver Verifier with Special Pool immediately exposes UAFs
- Windows 11 eliminates silent temporal corruption
- Late-detected crashes often originate far from the original UAF site

25.3 Race Conditions

25.3.1 Precise Windows 11Specific Definition

A **kernel race condition** in Windows 11 is a concurrency defect where multiple execution contexts (threads, DPCs, APCs, worker threads, or interrupt paths) access shared kernel state without correct synchronization, producing a non-deterministic and unsafe interleaving. The failure manifests as time-of-check/time-of-use (TOCTOU), inconsistent reference counts, stale list pointers, or corrupted object state.
Windows 11 classifies race conditions as **temporal concurrency safety violations** and mitigates their exploitability using:

- Fine-grained spinlocks and pushlocks
- IRQL discipline
- HVCI-backed indirect call integrity
- PatchGuard structural verification
- Memory ordering enforcement on modern CPUs

25.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Race conditions arise at boundaries where concurrent execution domains intersect:

- Dispatcher (ready queues, wait blocks)
- I/O Manager (IRP completion paths)
- Object Manager (handle close vs dereference)
- Memory Manager (VAD manipulation, working set trimming)

- Networking stack (NET_BUFFER_LIST reference paths)
- Power and PnP state transitions

Critical Windows 11 execution domains involved:

- Preemptive kernel threads (Ring 0)
- APC delivery at PASSIVE_LEVEL
- DPC execution at DISPATCH_LEVEL
- Interrupt handlers above DISPATCH_LEVEL

25.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Race conditions frequently corrupt these real kernel structures:

- `_EPROCESS` (ActiveProcessLinks, RundownProtect)
- `_KTHREAD` (WaitReason, WaitListEntry)
- `_IRP` (IoStatus, StackCount)
- `_LIST_ENTRY` (Flink, Blink)
- `_EX_PUSH_LOCK`
- `_KSPIN_LOCK`

Incorrect manipulation of:

- `ReferenceCount`
- `PointerCount`
- `ActiveProcessLinks`

is a primary crash vector under race conditions.

25.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Canonical Kernel Race Failure Flow

1. Thread A validates a kernel object pointer under insufficient locking
2. Thread B concurrently frees or reinitializes the same object
3. Thread A resumes and performs a write or dereference
4. Kernel object header is corrupted
5. Pool or object manager consistency check fails
6. PatchGuard, pool verifier, or VBS detects corruption
7. `KeBugCheckEx` is invoked

Typical Bug Checks Caused by Races

- `IRQL_NOT_LESS_OR_EQUAL`
- `KERNEL_SECURITY_CHECK_FAILURE`
- `SYSTEM_SERVICE_EXCEPTION`
- `BAD_POOL_CALLER`
- `REFERENCE_BY_POINTER`

25.3.5 Secure Kernel / VBS / HVCI Interaction

With VBS and HVCI enabled:

- Indirect call targets modified by race corruption fail HVCI validation
- Corrupted function pointers trigger immediate secure kernel shutdown
- PatchGuard periodically verifies:
  - IDT
  - SSDT
  - Kernel code sections
  - Critical function prologues
- CET shadow stack blocks corrupted return addresses

Thus, Windows 11 race conditions produce **crash-before-control** behavior rather than silent exploitation.

25.3.6 Performance Implications

Race protection introduces:

- Additional pushlock acquisitions
- Inter-processor memory barriers
- IRQL elevation for short critical sections
- Cache-line contention in high-frequency spin regions

Hybrid P-Core/E-Core scheduling in Windows 11 mitigates contention by prioritizing high-IRQL work on performance cores, maintaining deterministic lock behavior.

25.3.7 Real Practical Example (C / C++ / Assembly)

Incorrect Kernel C Code With Race

```
PVOID gShared;

VOID ThreadA(VOID)
{
    if (gShared) {
        RtlZeroMemory(gShared, 0x20); // race write
    }
}

VOID ThreadB(VOID)
{
    if (gShared) {
        ExFreePool(gShared); // race free
        gShared = NULL;
    }
}
```

Corrected With Pushlock

```
EX_PUSH_LOCK gLock;

VOID ThreadA(VOID)
{
    ExAcquirePushLockExclusive(&gLock);
    if (gShared) {
        RtlZeroMemory(gShared, 0x20);
    }
    ExReleasePushLockExclusive(&gLock);
}
```

```
VOID ThreadB(VOID)
{
    ExAcquirePushLockExclusive(&gLock);
    if (gShared) {
        ExFreePool(gShared);
        gShared = NULL;
    }
    ExReleasePushLockExclusive(&gLock);
}
```

External Assembly Critical Section Pattern

```
; acquire spinlock (simplified)
acquire:
    mov     rax, 1
    lock xchg qword ptr [lockvar], rax
    test    rax, rax
    jnz     acquire
; critical section
    mov     qword ptr [rdi], rcx
; release
    mov     qword ptr [lockvar], 0
```

25.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Initial Crash Analysis

```
!analyze -v
```

Check IRQL at Crash

```
!irql
```

Inspect Faulting Lock

```
dt nt!_EX_PUSH_LOCK @rcx
```

List Contended Threads

```
!thread
!locks
```

Validate Object Reference Counts

```
dt nt!_OBJECT_HEADER @rax-30
```

25.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historically, kernel races enabled:

- Reference count overflow
- Token stealing
- IRP double completion
- Object type confusion

In Windows 11, these vectors are neutralized by:

- Atomic reference updates
- Encoded object headers
- Pool cookie validation
- HVCI indirect call enforcement
- CET shadow stack return validation

- PatchGuard structural verification

Any successful race-based exploitation now requires:

- Simultaneous bypass of HVCI, VBS, CET, and PatchGuard

which is classified as a full-system hypervisor compromise.

25.3.10 Professional Kernel Engineering Notes

- Never access shared kernel memory without explicit synchronization
- IRQL discipline is mandatory for correctness, not just performance
- Always match acquire/release semantics exactly
- Never hold locks across blocking operations
- Driver Verifier with IRQL checking immediately exposes races
- Races now crash deterministically in Windows 11
- Exploitation assumptions valid before Windows 10 are obsolete

Chapter 26

Modern Mitigations

26.1 SMEP (Supervisor Mode Execution Prevention)

26.1.1 Precise Windows 11Specific Definition

Supervisor Mode Execution Prevention (SMEP) is a hardware-enforced CPU protection available on modern x86-64 processors that prevents execution of instructions located in *user-mode pages* while the processor is operating in *supervisor mode* (Ring 0).
In Windows 11, SMEP is **mandatorily enabled on all supported systems** and enforced as a core boundary between:

- User-mode memory (Ring 3)
- Kernel-mode execution (Ring 0)

Any attempt by kernel-mode control flow to execute code residing in pages marked with the User bit (`U/S = 1`) in the page tables results in a **hardware #PF (page fault) with a supervisor execution violation**.

26.1.2 Exact Architectural Role Inside the Windows 11 Kernel

SMEP acts as a **non-bypassable execution barrier** between:

- Kernel-mode instruction fetch
- User-mode virtual address ranges

It directly protects the following Windows 11 kernel subsystems from code redirection attacks:

- System call dispatch path
- IRP completion routines
- DPC execution paths
- Kernel APC delivery
- Driver function pointer tables

SMEP is enforced **before any instruction decoding occurs**, meaning:

- No speculative execution from user memory in Ring 0
- No ROP-to-user-memory transitions

26.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

SMEP enforcement is rooted in:

- `CR4.SMEP` bit (Control Register 4)
- Page table `U/S` (User/Supervisor) bit
- Kernel virtual address layout partitioning

Relevant Windows kernel structures interacting with SMEP-protected execution:

- `_KTRAP_FRAME` (stores faulting RIP)
- `_EXCEPTION_RECORD`
- `_KPROCESS` (CR4 shadow fields)
- `_KPRCB` (per-CPU control state)

26.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Illegal SMEP Execution Attempt

1. Kernel thread executes in Ring 0
2. Instruction pointer is redirected to user-mode virtual address
3. CPU checks `CR4.SMEP = 1`
4. Page table entry has `U/S = 1`
5. CPU raises **#PF with supervisor execution violation**
6. Windows 11 kernel enters `KiPageFault`
7. Security validation fails
8. `KeBugCheckEx` is invoked

Common Resulting Bug Check

- `PAGE_FAULT_IN_NONPAGED_AREA`
- `KERNEL_SECURITY_CHECK_FAILURE`
- `ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY`

26.1.5 Secure Kernel / VBS / HVCI Interaction

In Windows 11:

- SMEP is enforced **before** HVCI validation
- Secure Kernel (VTL1) verifies CR4 integrity at runtime
- Any attempt to disable SMEP via CR4 modification is detected by:
  - VBS
  - PatchGuard
  - Hyper-V root partition monitor

This produces a **non-recoverable system shutdown** on illegal CR4 manipulation.

26.1.6 Performance Implications

SMEP introduces:

- Zero measurable instruction overhead
- No TLB penalty
- No cache impact
- No speculative execution cost

It is enforced purely in the **instruction fetch permission stage** of the pipeline.

26.1.7 Real Practical Example (C / C++ / Assembly)

Illegal Kernel Jump to User Memory (Conceptual Failure Case)

```
typedef void (*FN)(void);

FN UserCallback = (FN)0x0000000040001000;

VOID VulnerablePath(VOID)
{
    UserCallback(); // SMEP blocks execution here
}
```

External x64 Assembly (Illegal SMEP Execution Pattern)

```
; Ring 0 attempting to jump to user memory
mov     rax, 0000000040001000h
jmp     rax
; CPU raises #PF due to SMEP
```

Correct Kernel-Safe Indirect Call Pattern

```
; validated kernel address only
mov     rax, FFFFF80512345678h
jmp     rax
```

26.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check CR4 SMEP State

```
r cr4
```

Verify Page Fault Cause

```
!analyze -v
```

Inspect Faulting RIP

```
r rip
```

Inspect Page Table Entry

```
!pte @rip
```

Decode Trap Frame

```
dt nt!_KTRAP_FRAME @rsp
```

26.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Before SMEP, attackers used:

- Kernel ROP chains that pivoted to user-mode shellcode
- Token-stealing stubs in user pages

In Windows 11:

- All kernel-to-user execution transitions are **hardware-blocked**
- SMEP + NX + HVCI + CET together eliminate classic Ring 0 payload execution
- Only remaining attack path is full kernel code reuse inside:
  - Signed kernel image
  - Verified driver images

26.1.10 Professional Kernel Engineering Notes

- Never design kernel control flow that depends on user memory execution
- All function pointers in kernel must be validated kernel addresses
- SMEP cannot be bypassed without hypervisor-level compromise
- Any CR4 modification attempt is fatal on Windows 11
- SMEP is permanently required for Windows 11 certification
- Driver code that disables SMEP will never load on a compliant system

26.2 SMAP (Supervisor Mode Access Prevention)

26.2.1 Precise Windows 11Specific Definition

Supervisor Mode Access Prevention (SMAP) is a hardware-enforced CPU protection mechanism on modern x86-64 processors that prevents **all data accesses** (read or write) to *user-mode pages* while the processor is executing in *supervisor mode* (Ring 0), unless explicitly overridden.
In Windows 11, SMAP is **mandatory and permanently enforced** on all supported systems. Any kernel-mode attempt to dereference memory mapped with the `User/Supervisor = User` page-table attribute triggers a **hardware #PF** unless:

- The kernel explicitly enables access using `STAC`
- And restores protection using `CLAC`

SMAP protects against **unauthorized kernel access to user-controlled memory**.

26.2.2 Exact Architectural Role Inside the Windows 11 Kernel

SMAP forms the **data-access boundary** between:

- Kernel-mode execution (Ring 0)
- User-mode virtual memory ranges

It directly protects:

- IRP user buffers
- IOCTL input/output buffers
- System call user arguments
- APC user memory payloads
- User-mode pointer dereferencing inside drivers

Windows 11 relies on SMAP to enforce that **every kernel access to user memory must be explicit, temporary, and validated**.

26.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

SMAP enforcement is based on:

- `CR4.SMAP` bit (Control Register 4)
- Page table `U/S` bit
- CPU `EFLAGS.AC` (Alignment Check flag)

Relevant Windows kernel structures involved in validated user memory access:

- `_KTRAP_FRAME`
- `_EXCEPTION_RECORD`
- `_KTHREAD`
- `_EPROCESS`

Kernel user-pointer validation paths rely on:

- `ProbeForRead`
- `ProbeForWrite`
- `MmCopyVirtualMemory`

26.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Illegal Kernel Read from User Memory

1. Kernel executes in Ring 0
2. Instruction attempts to read user-mapped virtual address
3. `CR4.SMAP = 1`
4. `EFLAGS.AC = 0`
5. CPU raises **#PF (SMAP violation)**
6. Windows 11 dispatches `KiPageFault`
7. Invalid access is classified as security violation
8. System is crashed or exception propagated

Legal Kernel Access Sequence

1. `STAC` executed (temporarily enables access)
2. User memory accessed
3. `CLAC` executed (restores protection)

26.2.5 Secure Kernel / VBS / HVCI Interaction

In Windows 11:

- Secure Kernel (VTL1) validates that `CR4.SMAP` remains enabled
- Hyper-V monitors privileged register modification attempts
- Any attempt to permanently disable SMAP triggers:
  - PatchGuard detection
  - Secure Kernel integrity violation
  - Immediate system halt

SMAP is non-optional when HVCI and VBS are active.

26.2.6 Performance Implications

SMAP has:

- Zero steady-state performance cost
- No cache impact
- No TLB overhead

- Only minimal overhead on rare `STAC/CLAC` sequences

The protection is enforced during **load/store permission checks** in the execution pipeline.

26.2.7 Real Practical Example (C / C++ / Assembly)

Unsafe Kernel Dereference (Blocked by SMAP)

```
NTSTATUS DriverRoutine(PVOID UserPointer)
{
    volatile UCHAR Value = *(UCHAR*)UserPointer; // SMAP fault
    return STATUS_SUCCESS;
}
```

External x64 Assembly Correct SMAP-Compliant Access

```
; Enable temporary user access
stac

mov     rax, [rcx]      ; Safe user memory read

; Restore SMAP protection
clac
```

Correct C Kernel Pattern

```
UCHAR value = 0;

__try {
    ProbeForRead(UserPtr, sizeof(UCHAR), 1);
    value = *(volatile UCHAR*)UserPtr;
}
__except(EXCEPTION_EXECUTE_HANDLER) {
    return STATUS_ACCESS_VIOLATION;
}
```

26.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check CR4 SMAP State

```
r cr4
```

Inspect Page Fault from SMAP Violation

```
!analyze -v
```

Inspect Instruction Causing Fault

```
r rip
ub @rip
```

Inspect Trap Frame

```
dt nt!_KTRAP_FRAME @rsp
```

26.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Before SMAP, attackers abused:

- Direct kernel reads of user-crafted memory
- Pointer substitution attacks
- IOCTL buffer reuse attacks

In Windows 11:

- All kernel data access to user memory must be explicit
- SMAP blocks:

  - Use-after-free user buffer reuse
  - Kernel stack pivot via user pointers
  - Token overwrite via user-mapped memory

- Exploits now require:
  - Full kernel memory disclosure
  - Or signed driver execution

26.2.10 Professional Kernel Engineering Notes

- Never dereference user pointers directly in kernel code
- Always use `ProbeForRead/Write` before access
- Avoid prolonged `STAC` windows
- Do not copy user memory at elevated IRQL
- Never attempt to disable SMAP via CR4
- All modern Windows 11 drivers are SMAP-compliant by design

26.3 KASLR (Kernel Address Space Layout Randomization)

26.3.1 Precise Windows 11Specific Definition

Kernel Address Space Layout Randomization (KASLR) in Windows 11 is a boot-time and runtime memory randomization mechanism that relocates the virtual base addresses of critical kernel images and core kernel data regions to unpredictable locations inside the canonical kernel virtual address space.

In Windows 11, KASLR applies to:

- `ntoskrnl.exe`
- `HAL.dll`
- Boot-start drivers
- Kernel pools
- System PTE ranges
- Executive object regions

The objective is to **invalidate all static kernel address assumptions** used by exploits.

26.3.2 Exact Architectural Role Inside the Windows 11 Kernel

KASLR forms the **primary spatial memory obfuscation layer** inside the Windows 11 kernel memory manager. It protects:

- SSDT base address discovery
- Kernel code gadgets
- Dispatch tables
- Global kernel object pointers
- Function pointer overwrite primitives

All kernel virtual addresses in Windows 11 are **relocated on every boot** using entropy derived from:

- Secure Boot measurements
- TPM seed material
- ACPI firmware entropy
- Hyper-V initialization randomness

26.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

KASLR affects the layout of the following real kernel structures:

- `_KLDR_DATA_TABLE_ENTRY`
- `_LDR_DATA_TABLE_ENTRY`
- `_MM_SYSTEM_SPACE`
- `_MI_SYSTEM_IMAGE_STATE`

Relevant fields impacted:

- `DllBase`
- `EntryPoint`
- `SizeOfImage`

Kernel image layout is finalized during:

- `MiRelocateImage`
- `MiInitializeSystemImage`

26.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Boot-Time KASLR Execution Path

1. UEFI Secure Boot verifies bootloader
2. Bootloader loads `ntoskrnl.exe` at a randomized physical offset
3. Kernel decompression occurs
4. `MiInitializeSystemImage` executes
5. Relocation fixups applied to kernel code sections
6. Page tables are built with randomized virtual bases
7. `CR3` is loaded with randomized mappings
8. Kernel execution begins at relocated RIP

Runtime Impact

- No fixed kernel entry points remain
- All exploit gadgets must be dynamically discovered
- Kernel symbol resolution becomes mandatory

26.3.5 Secure Kernel / VBS / HVCI Interaction

With Secure Kernel enabled:

- Hyper-V isolates kernel page tables
- KASLR entropy becomes virtualization-protected

- Kernel base disclosure requires VTL1 memory disclosure
- HVCI blocks unsigned code from observing relocated kernel memory

KASLR in Windows 11 is **cryptographically anchored** to secure boot state.

26.3.6 Performance Implications

KASLR has:

- Zero runtime overhead
- No impact on TLB performance
- No impact on cache locality
- No scheduling cost
- No NUMA penalties

Relocation is a **one-time boot-time operation**.

26.3.7 Real Practical Example (C / C++ / Assembly)

Kernel Base Discovery via `NtQuerySystemInformation`

```
#include <Windows.h>

typedef NTSTATUS(WINAPI* pNtQuerySystemInformation)(
    ULONG, PVOID, ULONG, PULONG);

int main() {
    ULONG size = 0;
    NtQuerySystemInformation(11, NULL, 0, &size);
```

```
    PVOID buffer = malloc(size);
    NtQuerySystemInformation(11, buffer, size, &size);

    return 0;
}
```

External x64 Assembly Extract Kernel Base from MSR (Read-Only)

```
; Read IA32_LSTAR MSR to locate syscall entry
mov ecx, 0xC0000082
rdmsr
shl rdx, 32
or  rax, rdx
; RAX now holds randomized syscall entry inside ntoskrnl
```

26.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display Kernel Base

```
lm m nt
```

Dump Kernel Image Location

```
!dh nt
```

Verify Relocation Delta

```
u nt!KiSystemCall64
```

Inspect Loaded Modules

```
lm
```

26.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

KASLR blocks:

- Static ROP chains
- Hardcoded kernel gadget offsets
- Predictable SSDT attacks
- Fixed function pointer overwrite exploits

Modern exploitation requires:

- Kernel information disclosure vulnerability
- Speculative execution leakage
- MSR leakage via signed drivers
- Hypervisor escape

Without an info-leak, **kernel exploitation is cryptographically infeasible**.

26.3.10 Professional Kernel Engineering Notes

- Never assume fixed kernel addresses
- Never embed kernel pointers inside persistent data
- Always resolve symbols dynamically during debugging
- All kernel drivers must be KASLR-relocatable
- Kernel ASLR entropy is higher when VBS is enabled
- Windows 11 forbids non-relocatable kernel images

26.4 CFG (Control Flow Guard)

26.4.1 Precise Windows 11Specific Definition

Control Flow Guard (CFG) in Windows 11 is a **fine-grained indirect call validation mechanism** enforced by the kernel and compiler toolchain to ensure that all indirect control transfers (calls, jumps, returns through function pointers, virtual tables, dispatch tables) target only **legitimate, pre-validated entry points**.
In Windows 11 kernel mode, CFG operates as:

- **KCFG** (Kernel Control Flow Guard)
- Enforced by `ntoskrnl.exe` with hardware and software assistance
- Integrated with **HVCI** and **VBS**

CFG enforces **control-flow integrity (CFI)** at the instruction level.

26.4.2 Exact Architectural Role Inside the Windows 11 Kernel

CFG is positioned as the **primary forward-edge execution integrity barrier** inside the Windows 11 kernel. It protects:

- Function pointer calls
- Dispatch tables
- I/O request handlers
- Object type callbacks
- Virtual function tables in C++

CFG transforms the kernel from:

$$\text{Unrestricted indirect branching} \longrightarrow \text{Validated indirect execution}$$

It blocks:

- ROP gadget chaining
- JOP (Jump-Oriented Programming)
- COOP (Counterfeit Object-Oriented Programming)

26.4.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

CFG enforcement uses the following real kernel structures and mechanisms:

- `_MI_CFG_CONTEXT`
- `_IMAGE_LOAD_CONFIG_DIRECTORY64`
- `GuardCFCheckFunctionPointer`
- `GuardCFDispatchFunctionPointer`
- `GuardFlags`

Relevant PE fields:

- `IMAGE_DLLCHARACTERISTICS_GUARD_CF`
- `IMAGE_LOAD_CONFIG_DIRECTORY.GuardCFFunctionTable`
- `IMAGE_LOAD_CONFIG_DIRECTORY.GuardCFFunctionCount`

Each valid indirect call target is registered in a **CFG bitmap** maintained by the kernel.

26.4.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel CFG Enforcement Path

1. Kernel image is loaded
2. PE loader parses `Load Config Directory`
3. CFG function table is registered
4. A **CFG bitmap** is built
5. Each valid function entry is marked
6. On every indirect call, a CFG validation check is injected
7. If target is not marked valid:
    - Immediate bugcheck or fast fail
    - Hypervisor-assisted termination under HVCI

Hardware Execution Effect

- Indirect CALL/JMP is validated before instruction retirement
- Illegal targets never reach execution stage

26.4.5 Secure Kernel / VBS / HVCI Interaction

With **VBS + HVCI enabled**:

- CFG bitmaps are protected inside VTL1
- Kernel cannot tamper with its own CFG tables

- Any attempt to disable CFG triggers virtualization fault
- Driver code is validated before CFG registration

CFG becomes **hypervisor-anchored control-flow integrity**.

26.4.6 Performance Implications

CFG overhead in Windows 11 is:

- Less than 1–2% in kernel hot paths
- Zero impact on direct calls
- No TLB penalty
- No cache pollution
- No scheduling overhead

CFG validation is implemented as a **single-predicate branch fast-path**.

26.4.7 Real Practical Example (C / C++ / Assembly)

Kernel-Mode C Example Indirect Call Protected by CFG

```
typedef void (*PFN_DISPATCH)(void);

PFN_DISPATCH g_DispatchTable[4];

void CallDispatch(UINT index)
{
    g_DispatchTable[index]();
}
```

At compile time:

- Each valid function pointer is emitted into the CFG table
- The indirect call is automatically guarded by CFG

External x64 Assembly Indirect Call (CFG Checked)

```
; RCX holds function pointer
call rcx        ; This call is validated by CFG before execution
```

If `RCX` is not registered as a valid CFG target, the kernel will immediately terminate execution.

26.4.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check CFG State

```
!cfg
```

Inspect Kernel CFG Tables

```
dt nt!_MI_CFG_CONTEXT
```

Validate CFG Flags in Kernel Image

```
!dh nt
```

Disassemble Guarded Call

```
u nt!SomeIndirectCallSite
```

26.4.9 Exploitation Surface, Attack Vectors & Security Boundaries

CFG blocks:

- Function pointer overwrites
- Virtual table corruption
- Dispatch table hijacking
- ROP/JOP pivot chains

Modern bypass attempts require:

- CFG bitmap corruption
- Hypervisor escape
- Signed driver abuse
- CPU speculative bypass with info leaks

On Windows 11 with HVCI enabled, **CFG bypass requires a full hypervisor compromise**.

26.4.10 Professional Kernel Engineering Notes

- All kernel drivers must be built with CFG enabled
- Never use raw unchecked indirect calls
- CFG violations trigger immediate kernel termination
- Dispatch tables must be static or pre-registered
- CFG is mandatory for Windows 11 production drivers
- CFG + KASLR + SMEP + SMAP form the core kernel execution barrier

26.5 VBS (Virtualization-Based Security)

26.5.1 Precise Windows 11Specific Definition

Virtualization-Based Security (VBS) in Windows 11 is a **hardware-enforced security architecture** that uses the **Hyper-V Type-1 hypervisor** to create an **isolated execution domain (VTL1)** that is cryptographically and architecturally separated from the normal Windows kernel (VTL0).
VBS enforces security by:

- Virtualizing the kernel using the hypervisor
- Isolating security-critical code and data
- Preventing the Windows kernel itself from tampering with protected regions

VBS is a **mandatory foundation** for all modern Windows 11 kernel security systems, including:

- HVCI
- Credential Guard
- Secure Kernel
- Kernel-mode Code Integrity
- Kernel CFG

26.5.2 Exact Architectural Role Inside the Windows 11 Kernel

Windows 11 operates with **two primary Virtual Trust Levels (VTLs)**:

- **VTL0**: Normal Windows kernel (ntoskrnl.exe, HAL, drivers)
- **VTL1**: Secure Kernel (SK) isolated by Hyper-V

The architectural control hierarchy becomes:

$$\text{Hyper-V} \rightarrow \text{Secure Kernel (VTL1)} \rightarrow \text{Windows Kernel (VTL0)}$$

This design ensures:

- The Windows kernel is no longer the highest authority
- All security policies are validated outside of the compromised kernel
- Kernel exploitation no longer grants full system control

26.5.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Key real VBS-related kernel structures include:

- `_HV_PARTITION`
- `_HV_VTL_CONTROL`
- `_SK_SECURE_DATA`
- `_CI_POLICY_OBJECT`
- `_MI_SYSTEM_VTL_CONFIG`

Critical CPU structures used by VBS:

- Extended Page Tables (EPT)
- VMCS (Intel)
- VMCB (AMD)

Memory ownership is enforced by the hypervisor, not by page tables managed by the Windows kernel.

26.5.4 Execution Flow (Step-by-Step at C & Assembly Level)

Windows 11 Boot Sequence with VBS

1. UEFI Secure Boot validates bootloader
2. Hyper-V hypervisor is loaded before ntoskrnl.exe
3. Secure Kernel (VTL1) is initialized
4. Windows kernel (VTL0) is started under hypervisor supervision
5. Kernel access to protected memory is virtualized
6. All security decisions are redirected to VTL1

VM-Exit-Based Enforcement

- Any forbidden memory access from VTL0 triggers a VM-exit
- Hypervisor blocks execution or terminates the system

External x64 Assembly Privilege Isolation (Conceptual Execution Boundary)

```
; Attempting to access protected VTL1 memory from VTL0
mov rax, 0FFFF800000000000h
mov rbx, [rax]      ; Causes hypervisor intercept under VBS
```

This access never completes inside VTL0.

26.5.5 Secure Kernel / VBS / HVCI Interaction

VBS forms the **execution isolation substrate** for:

- **HVCI**: Code integrity enforced in VTL1
- **Credential Guard**: LSASS secrets stored in VTL1
- **Kernel CFG**: CFG state protected from kernel modification
- **PatchGuard**: State monitored from outside the kernel

Security policy execution is therefore:

$$\text{Security Decision} \notin \text{Windows Kernel}$$

26.5.6 Performance Implications

Modern Windows 11 VBS overhead:

- 1–5% CPU overhead in worst-case I/O paths
- No impact on standard user-mode applications
- No scheduler destabilization
- EPT hardware acceleration minimizes TLB penalties

On Intel VT-x and AMD SVM, VM-exit costs are amortized using:

- EPT caching
- VMFUNC
- Posted interrupts

26.5.7 Real Practical Example (C / C++ / Assembly)

Kernel Driver Load Under VBS + HVCI

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
                     PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(RegistryPath);
    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

With VBS enabled:

- Driver image is validated by Secure Kernel
- If not signed or policy-compliant, load is denied before DriverEntry executes

26.5.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Check VBS State

```
!vsm
```

Check Hypervisor State

```
!hypervisor
```

Check HVCI Policy

```
!ci
```

Check Secure Kernel Presence

```
lm m securekernel
```

26.5.9 Exploitation Surface, Attack Vectors & Security Boundaries

VBS eliminates entire historical kernel attack classes:

- PatchGuard bypasses
- Token stealing persistence
- Kernel hook implants
- Direct credential extraction

Remaining theoretical attack surfaces:

- Hypervisor vulnerabilities
- CPU microarchitectural flaws
- Signed vulnerable driver abuse (temporarily until blocked)

With full VBS enabled, **kernel exploitation no longer implies system compromise**.

26.5.10 Professional Kernel Engineering Notes

- VBS is mandatory for Windows 11 Secured-Core systems
- Kernel-mode drivers must assume hypervisor supervision
- Direct memory manipulation is no longer reliable
- Traditional kernel rootkits are fully obsolete under VBS
- Debugging under VBS requires hypervisor-aware tooling
- Production drivers must be compatible with HVCI + VBS by default

Part XIV

Performance Engineering & Optimization

Chapter 27

Kernel Performance Bottlenecks

27.1 Scheduler

27.1.1 Precise Windows 11Specific Definition

The **Windows 11 Kernel Scheduler** is the preemptive, priority-driven, hybrid-aware execution control system implemented inside `ntoskrnl.exe` that governs:

- Thread dispatch
- Quantum allocation
- Priority boosting and decay
- Core-type selection (P-Core vs E-Core)
- NUMA locality enforcement
- Power-performance trade-offs

In Windows 11, the scheduler is **hardware-topology-aware** and cooperates directly with **Intel Thread Director** and **AMD CPPC2** to dynamically place threads on heterogeneous cores.

27.1.2 Exact Architectural Role Inside the Windows 11 Kernel

The scheduler operates at the highest execution authority inside **VTL0** and directly interacts with:

- Dispatcher database
- PRCB (Processor Control Blocks)
- KTHREAD / ETHREAD state machines
- DPC and APC delivery
- Interrupt return paths

All context switching, IRQL transitions, and quantum expiration are enforced by the scheduler using **trap frames** and **dispatcher objects**.

27.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Primary verified scheduler structures:

- `_KPRCB`
- `_KTHREAD`
- `_ETHREAD`
- `_KPROCESS`
- `_KSCHEDULER_SUBNODE`

- _KNODE
- _GROUP_AFFINITY

Critical _KTHREAD fields:

- Priority
- BasePriority
- QuantumTarget
- State
- WaitReason
- Affinity

Critical _KPRCB fields:

- CurrentThread
- NextThread
- IdleThread
- ReadySummary
- DpcQueueDepth

27.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel Dispatch Path

1. Timer interrupt fires
2. Quantum expiration detected
3. IRQL raised to `DISPATCH_LEVEL`
4. Current thread state saved to `_KTHREAD`
5. Ready queues scanned
6. Next thread selected based on:
    - Priority class
    - Core type (P/E)
    - SMT topology
    - Cache locality
7. Context switch via `KiSwapContext`

External x64 Assembly Context Switch Core (Symbolic)

```
PUBLIC KiSwapContext
KiSwapContext PROC
    mov     [rcx+18h], rsp
    mov     rsp, [rdx+18h]
    mov     rax, cr8
    ret
KiSwapContext ENDP
END
```

27.1.5 Secure Kernel / VBS / HVCI Interaction

Under VBS:

- Scheduler executes in VTL0
- Secure Kernel monitors dispatch integrity
- HVCI enforces that all scheduled kernel code is verified
- VTL transitions are forbidden during normal context switching

VBS prevents:

- Malicious thread injection
- Kernel hook scheduling
- Dispatch table tampering

27.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Primary scheduler bottlenecks:

- Excessive cross-NUMA migrations
- High DPC queue depth
- Overuse of priority boosting
- SMT sibling contention
- Excessive context switching

Windows 11 mitigations:

- Core-type aware load balancing
- Quantum scaling per power state
- Cache-aware thread placement
- NUMA node preference enforcement

27.1.7 Real Practical Example (C / C++ / Assembly)

Kernel Driver Measuring Quantum Expiration

```
VOID TimerDpcRoutine(
    KDPC* Dpc,
    PVOID Context,
    PVOID SysArg1,
    PVOID SysArg2)
{
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(Context);
    UNREFERENCED_PARAMETER(SysArg1);
    UNREFERENCED_PARAMETER(SysArg2);

    LARGE_INTEGER tick = KeQueryInterruptTime();
}
```

This measures scheduler-driven timer resolution and dispatch latency.

27.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display Scheduler Summary

```
!sched
```

Inspect PRCB

```
dt nt!_KPRCB
```

Show Running Threads

```
!running
```

Show Ready Queues

```
!ready
```

27.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical scheduler attacks included:

- Priority inversion exploits
- APC delivery hijacking
- Thread state forgery
- Deferred ready list corruption

Windows 11 mitigations:

- PatchGuard enforcement
- VBS dispatch protection
- HVCI verification of kernel execution paths
- Secure dispatcher integrity monitoring

Scheduler exploitation now requires a hypervisor-class vulnerability.

27.1.10 Professional Kernel Engineering Notes

- Poor scheduler behavior is the primary cause of unexplained kernel latency
- Excessive context switching is more damaging than raw CPU usage
- High DPC queue depth always indicates a driver-level bottleneck
- Priority boosting misuse causes long-term fairness collapse
- Core parking and E-Core misplacement degrade real-time workloads
- Scheduler tuning must always consider cache and NUMA topology

27.2 Section 2 Memory

27.2.1 Precise Windows 11Specific Definition

The **Windows 11 Kernel Memory Subsystem** is the integrated execution layer inside `ntoskrnl.exe` responsible for:

- Virtual address translation
- Page fault handling
- Working set management
- Commit accounting
- Kernel pool allocation
- NUMA-aware physical memory distribution

In Windows 11, the memory manager is fully integrated with:

- VBS-enforced page isolation
- HVCI-protected code regions
- KASLR-randomized kernel layout
- Hardware-enforced page permissions via EPT/NPT

27.2.2 Exact Architectural Role Inside the Windows 11 Kernel

The memory manager operates as a central execution authority within **VTL0** and directly controls:

- Page tables (PML4E PDPTE PDE PTE)
- Working set trimming
- Page fault resolution
- Copy-on-write operations
- Kernel and user virtual address partitioning

It coordinates with:

- I/O Manager (paging I/O)
- Cache Manager
- Object Manager
- Secure Kernel (VTL1)

27.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Primary verified memory manager structures:

- `_EPROCESS`
- `_MM_SESSION_SPACE`
- `_MMVAD_SHORT`
- `_MMVAD_LONG`
- `_MMPTE`
- `_MMWSL`
- `_MI_SYSTEM_NODE`

Critical `_EPROCESS` memory-related fields:

- `VadRoot`
- `WorkingSetPrivateSize`
- `CommitCharge`
- `VirtualSize`
- `PeakVirtualSize`

Critical `_MMPTE` bitfields:

- `Valid`
- `Dirty`

- `Accessed`
- `NoExecute`
- `PageFrameNumber`

27.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Kernel Page Fault Path

1. CPU triggers `#PF` exception
2. IRQL raised to `DISPATCH_LEVEL`
3. `KiPageFault` invoked
4. Faulting virtual address decoded
5. Corresponding VAD node located
6. PTE validated or demand-zero page allocated
7. Physical page mapped
8. TLB invalidated
9. Execution resumes

External x64 Assembly Simplified Page Fault Stub

```
PUBLIC KiPageFault
KiPageFault PROC
    push    rbp
    mov     rbp, rsp
    mov     rax, cr2
```

```
    call    MmAccessFault
    pop     rbp
    ret
KiPageFault ENDP
END
```

27.2.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 security:

- Kernel code pages are enforced as `RX` by HVCI
- VTL1 Secure Kernel owns integrity verification
- Page table updates are validated through hypervisor mediation
- Shadow stacks are isolated from kernel pools

VBS prevents:

- Unauthorized executable mappings
- Direct kernel page table tampering
- Kernel page permission escalation

27.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Primary kernel memory bottlenecks:

- Excessive hard page faults
- TLB shootdowns across cores

- Cross-NUMA memory migrations
- Large kernel pool fragmentation
- Cache-line bouncing in shared kernel objects

Windows 11 optimizations:

- NUMA-local physical page allocation
- Large-page mapping for kernel code
- Dynamic working set trimming
- Cache-aware pool allocation

27.2.7 Real Practical Example (C / C++ / Assembly)

Kernel Driver: Measuring Page Fault Frequency

```
VOID SampleMemoryProbe(PVOID Address)
{
    __try
    {
        volatile UCHAR value = *(volatile UCHAR*)Address;
        UNREFERENCED_PARAMETER(value);
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }
}
```

This forces controlled fault probing at PASSIVE_LEVEL.

External x64 Assembly Memory Touch Loop

```
PUBLIC TouchBuffer
TouchBuffer PROC
    mov     rcx, rdx
LoopStart:
    mov     al, [rcx]
    inc     rcx
    dec     r8
    jne     LoopStart
    ret
TouchBuffer ENDP
END
```

27.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Show System Memory Summary

```
!memusage
```

Display Process VAD Tree

```
!vad 0 1
```

Dump a PTE

```
!pte fffff80000000000
```

Show Kernel Pools

```
!poolused
```

27.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Historical memory exploitation classes:

- Pool overflows
- Use-after-free
- Double mapping attacks
- Page table privilege escalation
- NX bit bypass attempts

Windows 11 defenses:

- SMEP / SMAP enforcement
- VBS-based page table validation
- Kernel pool cookies
- HVCI code page verification
- KASLR address randomization

Modern kernel memory exploitation requires a cross-boundary hypervisor flaw.

27.2.10 Professional Kernel Engineering Notes

- Memory pressure always manifests first as scheduler instability
- TLB shootdowns scale catastrophically with CPU core count
- Cross-node NUMA allocations destroy real-time performance

- Kernel pool fragmentation increases attack surface
- Executable kernel pages are the most security-critical assets
- Poor memory locality is a hidden root cause of I/O latency

27.3 Section 3 Locks

27.3.1 Precise Windows 11Specific Definition

In Windows 11, **kernel locks** are low-level synchronization primitives implemented inside `ntoskrnl.exe` to serialize access to shared kernel data structures across:

- Multiple logical processors
- Hybrid P-Core / E-Core scheduling domains
- Interrupt and deferred execution contexts

These locks operate strictly under IRQL constraints and are enforced using:

- Atomic CPU instructions
- Cache-coherent memory ordering
- Processor-local spin optimization

Windows 11 locks are performance-critical structures directly influencing scheduler latency, I/O throughput, and memory subsystem scalability.

27.3.2 Exact Architectural Role Inside the Windows 11 Kernel

Kernel locks provide execution integrity for:

- Scheduler run queues
- Dispatcher ready lists
- Object Manager handle tables
- Memory manager working sets
- Kernel pool allocation metadata
- I/O request queues

They operate across execution domains:

- PASSIVE_LEVEL
- APC_LEVEL
- DISPATCH_LEVEL
- DIRQL (interrupt spinlocks)

Lock contention directly throttles:

- Context switching throughput
- DPC dispatch latency
- Interrupt service latency
- NUMA locality

27.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Primary synchronization structures used by Windows 11:

- `_KSPIN_LOCK`
- `_EX_PUSH_LOCK`
- `_ERESOURCE`
- `_KQUEUE`
- `_KTHREAD`

Relevant `_KTHREAD` synchronization fields:

- `SpinLock`
- `WaitReason`
- `WaitIrql`
- `WaitTime`

Relevant `_ERESOURCE` fields:

- `OwnerTable`
- `ActiveCount`
- `SharedWaiters`
- `ExclusiveWaiters`

27.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

Spin Lock Acquisition at DISPATCH_LEVEL

1. IRQL raised to `DISPATCH_LEVEL`
2. CPU performs atomic `LOCK BTS` or `LOCK XCHG`
3. Cache line ownership obtained
4. Lock owner written into `_KSPIN_LOCK`
5. Critical section entered
6. Lock released with atomic store
7. IRQL restored

External x64 Assembly Minimal Spin Lock

```
PUBLIC AcquireSpinLock
AcquireSpinLock PROC
SpinWait:
    mov     rax, 1
    lock xchg qword ptr [rcx], rax
    test    rax, rax
    jnz     SpinWait
    ret
AcquireSpinLock ENDP

PUBLIC ReleaseSpinLock
ReleaseSpinLock PROC
    mov     qword ptr [rcx], 0
    ret
ReleaseSpinLock ENDP
END
```

27.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 VBS:

- Lock code regions are HVCI-protected as immutable
- Atomic instructions are validated under EPT/NPT rules
- Spin loops cannot modify executable pages
- Secure Kernel isolates certain scheduler locks in VTL1

Secure Kernel enforcement guarantees:

- No runtime patching of lock primitives
- No privilege escalation through lock corruption
- No executable permission abuse inside lock code

27.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Primary lock-related performance bottlenecks:

- Cache-line bouncing on contended locks
- False sharing of adjacent lock variables
- Cross-NUMA node lock acquisition
- Hybrid-core scheduling imbalance
- Long critical sections at DISPATCH_LEVEL

Windows 11 mitigation strategies:

- Per-CPU queue partitioning
- Lock-free fast paths
- NUMA-aware pool partitioning
- Adaptive spinning before blocking

27.3.7 Real Practical Example (C / C++ / Assembly)

KMDF Driver Using a Spin Lock

```
KSPIN_LOCK GlobalLock;

VOID DriverRoutine()
{
    KIRQL oldIrql;
    KeAcquireSpinLock(&GlobalLock, &oldIrql);

    // Critical kernel section
    SharedKernelCounter++;

    KeReleaseSpinLock(&GlobalLock, oldIrql);
}
```

External x64 Assembly Atomic Counter Update

```
PUBLIC AtomicIncrement
AtomicIncrement PROC
    mov     rax, 1
    lock xadd qword ptr [rcx], rax
    ret
AtomicIncrement ENDP
END
```

27.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Detect Lock Contention

```
!locks
```

Show Thread Wait States

```
!thread
```

Inspect Dispatcher Objects

```
!dispatcher
```

View DPC Queues

```
!dpcs
```

27.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

Lock-related exploitation classes:

- Race-condition privilege escalation
- TOCTOU kernel object abuse
- Double-lock deadlock exploitation
- Interrupt-time corruption of shared objects
- Scheduler manipulation via stalled dispatcher locks

Windows 11 defenses:

- Strict IRQL validation

- HVCI-enforced lock code immutability
- Secure Kernel scheduler supervision
- Lock-free fast paths in critical subsystems

Modern exploitation requires a synchronized race across hypervisor boundaries.

27.3.10 Professional Kernel Engineering Notes

- Every kernel lock is a scalability liability
- Dispatcher locks define absolute scheduling throughput
- Spin locks must never wrap pageable code
- Lock contention grows exponentially with core count
- False sharing destroys hybrid-core performance
- Excessive DISPATCH_LEVEL holding causes global latency spikes

Chapter 28

High-Performance Driver Design

28.1 Lock-Free Structures

28.1.1 Precise Windows 11Specific Definition

In Windows 11, **lock-free structures** are kernel-resident data structures that guarantee forward progress without using blocking synchronization primitives (spin locks, push locks, or ERESOURCE). They rely exclusively on:

- Atomic CPU instructions (CMPXCHG, XADD)
- Cache-coherent memory ordering
- IRQL-safe non-blocking algorithms

These structures are fundamental to Windows 11 scalability on high-core-count systems and hybrid P-Core/E-Core architectures, where traditional locks become a dominant bottleneck.

28.1.2 Exact Architectural Role Inside the Windows 11 Kernel

Lock-free mechanisms are used in performance-critical kernel paths including:

- Per-CPU scheduler queues
- Deferred Procedure Call (DPC) work queues
- I/O completion paths
- Kernel pool lookaside lists
- Reference counting of kernel objects

Their primary architectural function is to:

- Eliminate dispatcher lock contention
- Avoid IRQL escalation
- Prevent priority inversion
- Preserve NUMA locality

Lock-free algorithms directly reduce scheduling latency and interrupt delivery overhead.

28.1.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Windows 11 internally implements lock-free patterns using:

- `_SLIST_HEADER`
- `_GENERAL_LOOKASIDE`
- `_NPAGED_LOOKASIDE_LIST`

- `_DISPATCHER_HEADER`

Key `_SLIST_HEADER` fields:

- `Alignment`
- `Next`
- `Depth`
- `CpuId`

These fields are updated exclusively through interlocked operations.

28.1.4 Execution Flow (Step-by-Step at C & Assembly Level)

Lock-Free Push Operation

1. Load current `SLIST_HEADER`
2. Set new element `Next` pointer
3. Perform atomic `CMPXCHG16B`
4. Retry on failure

Lock-Free Pop Operation

1. Load `SLIST_HEADER`
2. Extract head pointer
3. CAS header to next pointer
4. Retry on contention

External x64 Assembly Atomic Compare and Swap

```
PUBLIC AtomicCAS64
AtomicCAS64 PROC
; RCX = address, RDX = expected, R8 = new
    mov     rax, rdx
    lock cmpxchg qword ptr [rcx], r8
    ret
AtomicCAS64 ENDP
END
```

28.1.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 VBS:

- All atomic primitives execute under hypervisor-enforced memory protections
- SLIST headers in kernel pools are HVCI validated
- Write-what-where exploitation via CAS failures is blocked by EPT enforcement

The Secure Kernel isolates:

- Scheduler SLISTs
- DPC queues
- Inter-processor interrupt lists

This prevents manipulation of lock-free metadata across VTL boundaries.

28.1.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Lock-free structures optimize:

- Cache-line locality (no spinning)
- Reduced coherency traffic
- No IRQL inflation
- Zero-priority inversion
- Superior scaling on 64+ logical cores

However, poorly designed lock-free algorithms can cause:

- CAS retry storms
- Livelock under extreme contention
- Cross-NUMA cache thrashing

Windows 11 mitigates this via per-CPU partitioned SLISTs.

28.1.7 Real Practical Example (C / C++ / Assembly)

Lock-Free Kernel Stack Using SLIST

```
SLIST_HEADER StackHead;

VOID PushItem(PSLIST_ENTRY Entry)
{
    InterlockedPushEntrySList(&StackHead, Entry);
}
```

```
PSLIST_ENTRY PopItem()
{
    return InterlockedPopEntrySList(&StackHead);
}
```

External x64 Assembly Lock-Free Increment

```
PUBLIC AtomicIncrement64
AtomicIncrement64 PROC
    mov     rax, 1
    lock xadd qword ptr [rcx], rax
    ret
AtomicIncrement64 ENDP
END
```

28.1.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Inspect SLIST Structures

```
dt nt!_SLIST_HEADER
```

View Lookaside Lists

```
!lookaside
```

Detect Contended Atomic Operations

```
!perfinfo
```

Thread Execution States

```
!thread
```

28.1.9 Exploitation Surface, Attack Vectors & Security Boundaries

Lock-free abuse vectors:

- ABA problem exploitation
- Pointer reuse attacks
- Stale CAS success via heap grooming
- Cross-CPU SLIST manipulation

Windows 11 protections:

- Pointer encoding
- Pool cookie validation
- HVCI write-protection
- Secure Kernel scheduler isolation

Modern kernel exploitation against SLIST structures requires defeating hypervisor memory protections.

28.1.10 Professional Kernel Engineering Notes

- Lock-free is mandatory above 32 logical processors
- CAS retries must be bounded
- Never mix pageable memory with atomic paths
- Always design for NUMA locality
- Prefer per-CPU queues over global SLISTs
- Lock-free does not mean wait-free under all contention levels

28.2 Cache-Friendly Layout

28.2.1 Precise Windows 11Specific Definition

A **cache-friendly layout** in Windows 11 kernel drivers is a memory organization strategy that aligns data structures and access patterns to the physical CPU cache hierarchy (L1/L2/L3) in order to:

- Minimize cache-line evictions
- Avoid false sharing between CPU cores
- Reduce memory latency in hot execution paths
- Preserve deterministic timing in high-IRQL contexts

This design approach is mandatory for performance-critical Windows 11 kernel subsystems operating under high interrupt rates, high I/O throughput, or high core counts.

28.2.2 Exact Architectural Role Inside the Windows 11 Kernel

Cache-friendly layout directly affects the performance of:

- Scheduler run queues
- DPC and timer queues
- I/O request packet (IRP) processing paths
- Kernel pool allocators
- Object reference counting

Its architectural role is to:

- Reduce cross-core cache coherency traffic
- Optimize hardware prefetch behavior
- Maintain predictable execution latency at DISPATCH_LEVEL and above
- Prevent starvation on hybrid P-Core/E-Core systems

Windows 11 kernel hot paths are explicitly organized to fit inside a limited number of cache lines to preserve microarchitectural efficiency.

28.2.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Cache-sensitive kernel structures include:

- `_KPRCB` Per-CPU control block
- `_KTHREAD` Thread control block
- `_KDPC` Deferred procedure call object
- `_DISPATCHER_HEADER`
- `_GENERAL_LOOKASIDE`

Example cache-hot fields inside `_KTHREAD`:

- `State`
- `Priority`
- `WaitReason`

- `KernelStack`
- `ApcState`

These fields are deliberately placed within the same cache lines to prevent repeated cache refetch.

28.2.4 Execution Flow (Step-by-Step at C & Assembly Level)

Cache-Friendly Access Flow

1. Thread enters kernel via interrupt or syscall
2. `_KTHREAD` hot fields are accessed first
3. Scheduler consults local `_KPRCB` (per-CPU cache residency)
4. DPC queue head accessed from local L1 cache
5. Cold paths deferred to second-level cache accesses

External x64 Assembly Cache-Line Aligned Load

```
PUBLIC LoadAligned64
LoadAligned64 PROC
; RCX = aligned address
    mov     rax, qword ptr [rcx]
    ret
LoadAligned64 ENDP
END
```

This instruction avoids split cache-line reads when the structure is 64-byte aligned.

28.2.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- Cache-friendly kernel structures remain fully protected by EPT
- Secure Kernel hot paths reside in isolated VTL1 memory
- Cache timing side-channel leakage is mitigated by hypervisor scheduling barriers

HVCI further enforces:

- Read-only enforcement on code hot paths
- Prevention of layout corruption through write-protection

Cache optimization therefore remains strictly performance-focused and not an attack vector.

28.2.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

Cache-friendly layout directly influences:

- L1 cache hit ratio
- DPC execution latency
- Scheduler fairness across cores
- Reduced TLB pressure through compact structure access
- NUMA-local memory access efficiency

Failure to apply cache-aware layout causes:

- Remote-core cache invalidations

- Unbounded interrupt latency
- Increased power consumption
- Scheduler jitter on hybrid systems

Windows 11 internally uses per-CPU alignment to preserve cache locality across P-Core clusters.

28.2.7 Real Practical Example (C / C++ / Assembly)

Cache-Aligned Kernel Structure

```
__declspec(align(64))
typedef struct _FAST_PACKET
{
    volatile LONG  State;
    ULONG          Length;
    PVOID          Buffer;
    ULONG          Flags;
} FAST_PACKET;
```

External x64 Assembly Aligned Store

```
PUBLIC StoreAligned64
StoreAligned64 PROC
; RCX = aligned address, RDX = value
    mov     qword ptr [rcx], rdx
    ret
StoreAligned64 ENDP
END
```

This guarantees single-cache-line stores with no split-line write penalties.

28.2.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Examine Structure Layout

```
dt nt!_KTHREAD
```

Inspect Cache-Line Alignment

```
!poolused 2
```

View Per-CPU Structures

```
!prcb
```

Measure DPC Latency

```
!dpcs
```

28.2.9 Exploitation Surface, Attack Vectors & Security Boundaries

Cache layout can influence:

- Speculative execution side channels
- Cross-core cache timing measurements
- Branch target inference attacks

Windows 11 mitigates these via:

- Kernel VA shadowing
- IBRS and STIBP
- Hypervisor-enforced core scheduling isolation

Direct cache-line probing of Secure Kernel memory is architecturally blocked.

28.2.10 Professional Kernel Engineering Notes

- Always align hot kernel structures to 64 bytes
- Never mix hot and cold fields in the same cache line
- Avoid false sharing between DPC and worker threads
- Use per-CPU partitioning to preserve NUMA locality
- Never perform cache-inefficient memory walks at DISPATCH_LEVEL
- Cache efficiency directly impacts interrupt latency and scheduler precision

28.3 NUMA Optimization

28.3.1 Precise Windows 11Specific Definition

NUMA optimization in Windows 11 kernel drivers is the explicit alignment of memory allocation, CPU execution, interrupt routing, and data access patterns to the **local NUMA node** of the executing logical processor in order to:

- Eliminate remote memory access latency
- Reduce cross-node interconnect traffic (UPI / Infinity Fabric)
- Preserve deterministic execution timing at elevated IRQL
- Prevent scheduler-induced memory locality collapse

In Windows 11, NUMA optimization is mandatory for **high-throughput drivers, storage drivers, network drivers, virtualization drivers, and DPC-heavy subsystems**.

28.3.2 Exact Architectural Role Inside the Windows 11 Kernel

NUMA optimization directly governs the performance behavior of:

- **Scheduler group assignment** (Processor Groups + NUMA nodes)
- **Interrupt steering** (MSI-X vector to node-local CPU)
- **Memory manager node-aware pool allocation**
- **Per-NUMA lookaside lists**
- **DPC and ISR locality**

Architecturally, Windows 11 guarantees that:

- Each NUMA node has a dedicated physical memory region
- Each processor group maps to one or more NUMA nodes
- Kernel pools support node-local allocation
- Scheduler attempts to preserve memory and execution locality

NUMA-unaware drivers collapse scalability on systems with more than one memory controller.

28.3.3 Internal Kernel Data Structures (REAL STRUCTS & FIELDS)

Key NUMA-related kernel structures include:

- `_KNODE`
- `_KPRCB`

- `_KPROCESSOR_NODE`
- `_GENERAL_LOOKASIDE`
- `_MI_NODE_INFORMATION`

Critical NUMA fields include:

- `_KPRCB::Node`
- `_KNODE::FreeCount`
- `_KNODE::ProcessorMask`
- `_MI_NODE_INFORMATION::ChannelCount`

These structures enforce memory locality, cache prefetch efficiency, and interrupt routing precision.

28.3.4 Execution Flow (Step-by-Step at C & Assembly Level)

NUMA-Aware Execution Path

1. Interrupt arrives on a node-local CPU
2. ISR executes using node-local stack and PRCB
3. DPC is queued to the same node
4. Driver allocates memory from node-local pool
5. CPU accesses memory through the local memory controller
6. No UPI/IF fabric traversal occurs

External x64 Assembly Node-Local Memory Touch

```
PUBLIC TouchLocalNumaMemory
TouchLocalNumaMemory PROC
; RCX = pointer to node-local buffer
    mov     rax, qword ptr [rcx]
    add     rax, 1
    mov     qword ptr [rcx], rax
    ret
TouchLocalNumaMemory ENDP
END
```

This ensures first-touch locality on the executing node.

28.3.5 Secure Kernel / VBS / HVCI Interaction

Under Windows 11 with VBS enabled:

- Secure Kernel memory is permanently confined to dedicated NUMA-resident VTL1 regions
- Hypervisor enforces NUMA separation via EPT translation domains
- HVCI prevents unauthorized modification of node-mapped kernel code
- Secure memory is never mapped across NUMA boundaries

NUMA locality is therefore enforced both by the Windows kernel and by the hypervisor.

28.3.6 Performance Implications (Cache, TLB, NUMA, Scheduling)

NUMA-aware drivers directly improve:

- Memory latency (local vs remote DRAM)

- Cache residency stability
- TLB locality
- Interrupt service determinism
- DPC throughput
- Storage and network throughput scaling

NUMA violations cause:

- Fabric saturation
- Remote memory stalls
- Cross-node cache invalidations
- Elevated DPC latency
- Scheduler thrashing across nodes

On Windows 11 hybrid systems, NUMA locality is preserved per performance core cluster.

28.3.7 Real Practical Example (C / C++ / Assembly)

NUMA-Aware Kernel Allocation

```
PVOID AllocateNodeLocalBuffer(SIZE_T Size)
{
    USHORT node = KeGetCurrentNodeNumber();
    return ExAllocatePool2(POOL_FLAG_NON_PAGED, Size, 'ANUM');
}
```

NUMA-Aware Lookaside List

```
GENERAL_LOOKASIDE numaLookaside;
KeInitializeGeneralLookaside(
    &numaLookaside,
    ExAllocatePoolWithTag,
    ExFreePool,
    0,
    sizeof(MY_PACKET),
    'MNUL',
    0
);
```

External x64 Assembly NUMA-Local Store

```
PUBLIC StoreNodeLocal
StoreNodeLocal PROC
; RCX = pointer, RDX = value
    mov     qword ptr [rcx], rdx
    ret
StoreNodeLocal ENDP
END
```

28.3.8 Kernel Debugging & Inspection (REAL WinDbg / KD Commands)

Display NUMA Nodes

```
!numa
```

Display Per-CPU Node Binding

```
!prcb
```

Inspect Memory Node Mapping

```
!memnode
```

Check Interrupt Vector Routing

```
!interrupts
```

28.3.9 Exploitation Surface, Attack Vectors & Security Boundaries

NUMA misconfiguration affects:

- Cache-timing side channels
- Cross-node memory probing
- Hypervisor scheduling fingerprinting

Windows 11 mitigates these via:

- Hypervisor-enforced memory domain isolation
- Restricted cross-node speculative execution
- Secure Kernel NUMA domain enforcement

NUMA-based side-channel attacks are structurally limited in VBS-enabled environments.

28.3.10 Professional Kernel Engineering Notes

- Always allocate memory on the executing NUMA node
- Never migrate DPC-heavy workloads across NUMA domains
- Never share hot data across nodes without explicit partitioning
- Align MSI-X vectors with NUMA-local CPUs
- Avoid global pools on multi-socket systems
- NUMA violations destroy both latency predictability and throughput scaling
- Proper NUMA design is mandatory for enterprise-grade Windows 11 drivers

Conclusion

This book has presented Windows 11 kernel engineering as it exists in reality: a **hypervisor-governed, security-enforced, hardware-mediated execution environment**. The classical model of an omnipotent kernel no longer applies. Windows 11 operates under continuous supervision by the Hyper-V hypervisor and Secure Kernel (VTL1), with execution authority, memory access, and code integrity permanently constrained by hardware-backed trust mechanisms.
Throughout this work, every subsystem was analyzed under four uncompromising principles:

- **Execution correctness at the lowest level**
- **Real enforcement by hardware and virtualization**
- **Security as a first-class architectural constraint**
- **Performance as a directly measurable physical outcome**

The modern Windows kernel is no longer defined purely by software design. It is defined by a three-layer authority model:

- User Mode (Ring 3) for untrusted execution
- Kernel Mode (Ring 0) for controlled privileged execution

- Secure Kernel (VTL1) and Hypervisor for final enforcement

This layering fundamentally alters how:

- Memory is mapped and validated
- DMA is authorized and isolated
- Code execution is verified and constrained
- Interrupts, APCs, and DPCs interact with security policy

Windows 11 kernel engineering is therefore no longer an exercise in unrestricted control. It is an engineering discipline centered on **operating correctly within permanent supervision**. Driver developers, security engineers, and systems architects must now think in terms of:

- Page-table ownership under hypervisor control
- IOMMU-enforced DMA domains
- HVCI-enforced code execution
- VBS-mediated trust boundaries

This book deliberately avoided abstraction-heavy descriptions and API-level explanations. Instead, it focused on:

- Native system calls and real execution paths
- Kernel memory pools and their security properties
- Scheduler behavior on hybrid P-Core/E-Core systems
- NUMA locality and cache coherency effects

- IRQL, APC, and DPC ordering constraints

Modern exploitation and defense were addressed as two sides of the same architectural reality. The same enforcement mechanisms that defend the system also define the remaining attack surfaces. Understanding:

- Use-after-free
- Race conditions
- Pool corruption
- Control-flow subversion

requires the same level of architectural precision as implementing their mitigations.
The appendices consolidated the core operational references required for real-world work:

- Kernel structures
- NTAPI indexing
- Calling conventions
- Bug check codes
- IRQL rules
- Secure boot measurement paths
- DMA and MDL behavior

These are not academic details. They are **daily operational constraints for real kernel engineers**.

Windows 11 represents a permanent turning point in operating system design. Security is no longer layered on top of the kernel. It is structurally embedded below it. Performance is no longer purely a function of code efficiency. It is a negotiation between scheduler topology, memory locality, and hardware enforcement.

This book was written for engineers who accept this reality and operate within it deliberately, precisely, and correctly.

The future of Windows kernel engineering will not reward unrestricted access. It will reward **architectural understanding, security alignment, and execution discipline**. This work is intended to serve as a stable technical reference within that future.

Appendices Critical for Engineers

Appendix A Windows 11 Kernel Structures Reference

A.1 Engineering Scope

This appendix provides a **Windows 11specific engineering reference** for the most critical kernel data structures observable in modern production builds of `ntoskrnl.exe`. All structures listed here are:

- Symbol-verifiable through WinDbg public symbols
- Actively used by the Windows 11 kernel
- Architecturally relevant for performance, security, and exploitation research
- Valid only for Windows 11 (no legacy Windows mixing)

A.2 Processor and Scheduling Structures

A.2.1 _KPRCB Kernel Processor Control Block

Architectural Role: Per-logical-processor execution state, interrupt routing, and scheduling anchor.

```
_KPRCB
{
  ULONG MxCsr;
  UCHAR Number;
  UCHAR InterruptRequest;
  UCHAR IdleHalt;
  UCHAR CurrentIrql;
  PKTHREAD CurrentThread;
  PKTHREAD NextThread;
  PKTHREAD IdleThread;
  KAFFINITY Affinity;
  USHORT Node;
  ULONG ReadySummary;
}
```

Subsystems Using _KPRCB:

- Scheduler
- Interrupt Dispatcher
- DPC Manager
- NUMA Node Routing

A.2.2 _KTHREAD Kernel Thread Object

Architectural Role: Represents a schedulable execution entity inside the kernel.

```
_KTHREAD
{
  DISPATCHER_HEADER Header;
  LIST_ENTRY MutantListHead;
  PVOID InitialStack;
```

```
  PVOID StackLimit;
  PVOID KernelStack;
  KAFFINITY Affinity;
  KPRIORITY Priority;
  KPRIORITY BasePriority;
}
```

A.3 Process Structures

A.3.1 _EPROCESS Executive Process Object

Architectural Role: High-level executive representation of a process.

```
_EPROCESS
{
  KPROCESS Pcb;
  EX_PUSH_LOCK ProcessLock;
  LARGE_INTEGER CreateTime;
  LARGE_INTEGER ExitTime;
  PPEB Peb;
  HANDLE UniqueProcessId;
  LIST_ENTRY ActiveProcessLinks;
  SIZE_T VirtualSize;
}
```

A.3.2 _KPROCESS Kernel Process Object

Architectural Role: Low-level scheduler process control block.

```
_KPROCESS
{
  DISPATCHER_HEADER Header;
  LIST_ENTRY ProfileListHead;
  ULONG_PTR DirectoryTableBase;
```

```
  KAFFINITY Affinity;
  ULONG ActiveProcessors;
}
```

A.4 Memory Manager Structures

A.4.1 _MMVAD Virtual Address Descriptor

Architectural Role: Represents a virtual memory region inside a process address space.

```
_MMVAD
{
  ULONG_PTR StartingVpn;
  ULONG_PTR EndingVpn;
  ULONG Flags;
  PVOID Subsection;
}
```

A.4.2 _MMPTE Page Table Entry Abstraction

Architectural Role: Software abstraction of hardware page table entries.

```
_MMPTE
{
  ULONG64 Valid : 1;
  ULONG64 Write : 1;
  ULONG64 Owner : 1;
  ULONG64 WriteThrough : 1;
  ULONG64 CacheDisable : 1;
  ULONG64 Accessed : 1;
  ULONG64 Dirty : 1;
  ULONG64 LargePage : 1;
  ULONG64 Global : 1;
  ULONG64 PageFrameNumber : 36;
```

```
}
```

A.4.3 _MI_NODE_INFORMATION NUMA Memory Node

Architectural Role: Tracks per-node physical memory state.

```
_MI_NODE_INFORMATION
{
  ULONG ChannelCount;
  ULONG LargePageCount;
  ULONG FreeCount;
}
```

A.5 I/O Manager Structures

A.5.1 _DEVICE_OBJECT

Architectural Role: Represents a device instance inside the I/O subsystem.

```
_DEVICE_OBJECT
{
  SHORT Type;
  USHORT Size;
  LONG ReferenceCount;
  PDRIVER_OBJECT DriverObject;
  PDEVICE_OBJECT AttachedDevice;
  PVOID DeviceExtension;
  ULONG Flags;
}
```

A.5.2 _DRIVER_OBJECT

Architectural Role: Represents a loaded kernel-mode driver.

```
_DRIVER_OBJECT
{
  SHORT Type;
  USHORT Size;
  PDEVICE_OBJECT DeviceObject;
  PVOID DriverStart;
  ULONG DriverSize;
  PVOID DriverExtension;
  UNICODE_STRING DriverName;
}
```

A.5.3 _IRP I/O Request Packet

Architectural Role: Represents an I/O transaction in flight.

```
_IRP
{
  SHORT Type;
  USHORT Size;
  PMDL MdlAddress;
  ULONG Flags;
  LIST_ENTRY ThreadListEntry;
  IO_STATUS_BLOCK IoStatus;
  PIO_STACK_LOCATION CurrentStackLocation;
}
```

A.6 Object Manager Structures

A.6.1 _OBJECT_TYPE

```
_OBJECT_TYPE
{
  LIST_ENTRY TypeList;
  UNICODE_STRING Name;
```

```
  ULONG TotalNumberOfObjects;
  ULONG TotalNumberOfHandles;
}
```

A.6.2 _HANDLE_TABLE

```
_HANDLE_TABLE
{
  ULONG_PTR TableCode;
  ULONG HandleCount;
  EX_PUSH_LOCK HandleLock;
}
```

A.7 Interrupt and Deferred Execution

A.7.1 _KINTERRUPT

```
_KINTERRUPT
{
  ULONG Vector;
  ULONG Irql;
  PKSERVICE_ROUTINE ServiceRoutine;
  PVOID ServiceContext;
  KAFFINITY Affinity;
}
```

A.7.2 _KDPC

```
_KDPC
{
  UCHAR Type;
  UCHAR Importance;
  USHORT Number;
  LIST_ENTRY DpcListEntry;
```

```
  PKDEFERRED_ROUTINE DeferredRoutine;
  PVOID DeferredContext;
}
```

A.8 Secure Kernel and Hypervisor Structures

A.8.1 Secure Kernel Memory Regions (VTL1)

```
VTL1 Secure Memory Regions
{
  Isolated Page Tables
  Secure Kernel Stack
  Secure Kernel Code
}
```

A.8.2 Hypervisor Root Partition Control

```
_HV_PARTITION
{
  PartitionId
  VpIndex
  EptPointer
}
```

A.9 WinDbg Structure Verification Commands

```
dt nt!_KPRCB
dt nt!_EPROCESS
dt nt!_KTHREAD
dt nt!_MMPTE
dt nt!_KDPC
dt nt!_IRP
!process 0 1
```

```
!thread
!memusage
!handle
!interrupts
```

A.10 Professional Kernel Engineering Notes

- Never hardcode structure offsets across Windows builds
- Always validate offsets dynamically using symbols
- Secure Kernel (VTL1) memory is invisible to normal kernel debugging
- NUMA locality must be verified using `!numa` and `!prcb`
- Hypervisor memory exists outside the normal NT virtual map

Appendix B NTAPI Function Index

B.1 Engineering Definition

The Native API (NTAPI) is the **lowest public boundary between user mode (Ring 3) and the Windows 11 kernel (Ring 0)**. All NTAPI functions are implemented internally inside:

- `ntoskrnl.exe`
- `HAL.dll`

And exposed to user mode primarily through:

- `ntdll.dll`

These interfaces form the **true syscall contract** of Windows 11.

B.2 Architectural Role Inside Windows 11

NTAPI functions provide:

- Direct syscall entry to kernel services
- Execution routing into kernel subsystems:
  - Object Manager
  - Memory Manager
  - I/O Manager
  - Process Manager
  - Scheduler
  - Security Reference Monitor
- Secure transition via `syscall/sysret`
- Enforcement of SMEP, SMAP, CFG, and HVCI at the boundary

B.3 NTAPI Execution Flow (User Kernel)

1. User code calls `ntdll.dll` stub
2. Stub loads syscall index into `RAX`
3. Arguments passed via Windows x64 ABI
4. `syscall` instruction executed
5. CPU switches to Ring 0
6. Kernel dispatches via `KiSystemCall64`

7. Target service executes inside `ntoskrnl.exe`

8. `sysret` returns to Ring 3

B.4 Core NTAPI Function Index (Windows 11)

B.4.1 Process and Thread Management

```
NtCreateUserProcess
NtTerminateProcess
NtSuspendProcess
NtResumeProcess
NtCreateThreadEx
NtTerminateThread
NtDelayExecution
NtQueryInformationProcess
NtQueryInformationThread
```

B.4.2 Virtual Memory Management

```
NtAllocateVirtualMemory
NtFreeVirtualMemory
NtProtectVirtualMemory
NtReadVirtualMemory
NtWriteVirtualMemory
NtQueryVirtualMemory
NtMapViewOfSection
NtUnmapViewOfSection
```

B.4.3 Section Objects and Image Mapping

```
NtCreateSection
NtOpenSection
NtExtendSection
```

```
NtQuerySection
NtMapViewOfSection
```

B.4.4 Object Manager and Handles

```
NtCreateObject
NtOpenObject
NtDuplicateObject
NtQueryObject
NtClose
```

B.4.5 I/O Manager and File Operations

```
NtCreateFile
NtReadFile
NtWriteFile
NtDeviceIoControlFile
NtQueryInformationFile
NtSetInformationFile
```

B.4.6 Security and Tokens

```
NtOpenProcessToken
NtDuplicateToken
NtQueryInformationToken
NtAdjustPrivilegesToken
NtAccessCheck
```

B.4.7 Synchronization Primitives

```
NtCreateEvent
NtSetEvent
NtResetEvent
NtWaitForSingleObject
```

```
NtWaitForMultipleObjects
NtCreateMutant
NtReleaseMutant
```

B.4.8 ALPC and IPC

```
NtAlpcCreatePort
NtAlpcConnectPort
NtAlpcSendWaitReceivePort
NtAlpcAcceptConnectPort
```

B.5 Syscall ABI (Windows 11 x64)

```
RAX = Syscall Number
RCX = Arg1
RDX = Arg2
R8  = Arg3
R9  = Arg4
Shadow Space = 32 bytes
Stack Alignment = 16 bytes
```

B.6 External Assembly Syscall Stub (Windows 11 x64)

```asm
; File: nt_syscall.asm
; Windows 11 x64 syscall stub (Intel syntax)

global NtQueryInformationProcess

NtQueryInformationProcess:
    mov     r10, rcx
    mov     eax, 0x19        ; Example Syscall ID (varies by build)
    syscall
    ret
```

B.7 Kernel Debugging & Verification

```
x nt!Nt*
x ntdll!Nt*
u nt!KiSystemCall64
!syscalls
```

B.8 Exploitation Surface & Security Boundaries

- NTAPI is the primary attack surface for:
  - Privilege escalation
  - Token manipulation
  - Handle table corruption
  - Section object abuse
  - ALPC exploitation
- SMEP prevents direct shellcode execution in kernel
- SMAP prevents kernel from accessing user data without explicit override
- CFG restricts indirect branch abuse
- HVCI blocks unsigned kernel memory execution

B.9 Professional Kernel Engineering Notes

- Syscall numbers change between Windows 11 builds
- Never hardcode syscall IDs in production tools
- Always resolve through symbol lookup or dynamic extraction

- Direct syscalls bypass Win32 API validation layers
- All NTAPI calls execute under strict IRQL and memory access constraints

Appendix C WinDbg Command Reference

C.1 Engineering Definition

WinDbg is the **authoritative kernel-mode debugging environment for Windows 11**. It provides **direct symbolic, structural, memory, execution, and security-level visibility into `ntoskrnl.exe`, HAL, drivers, and the Secure Kernel**. All kernel crash forensics, runtime verification, scheduler analysis, memory inspection, and exploit research rely on WinDbg.

C.2 Architectural Role Inside Windows 11

WinDbg interacts with Windows 11 through:

- KD transport (USB, NET, COM, VM channel)
- Live kernel debugging
- Full memory crash dumps
- Kernel mini-dumps
- Hyper-V synthetic debug channels

It attaches to:

- `ntoskrnl.exe`
- `hal.dll`

- Boot drivers
- Runtime KMDF/WDM drivers
- Secure Kernel (VTL1) via hypervisor mediation

C.3 Core Execution Flow in Kernel Debugging

1. CPU generates exception, bugcheck, or breakpoint
2. Control transfers to `nt!KiDispatchException`
3. If KD attached, kernel halts scheduling
4. Debug state exported to WinDbg
5. Engineer inspects memory, registers, IRQL, stack, and threads

C.4 Fundamental WinDbg Control Commands

```
g         ; Continue execution
t         ; Single-step into
p         ; Single-step over
bp        ; Set breakpoint
bl        ; List breakpoints
bc        ; Clear breakpoint
bd        ; Disable breakpoint
be        ; Enable breakpoint
qd        ; Quit and detach
```

C.5 Register and Execution State Inspection

```
r         ; Display all registers
r rip     ; Display instruction pointer
```

```
r rsp    ; Display stack pointer
r cr3    ; Display page table base
u rip    ; Disassemble at RIP
ub rip   ; Disassemble backward
```

C.6 Stack and Call Chain Analysis

```
k        ; Stack trace
kp       ; Stack trace with parameters
kv       ; Verbose stack trace
.kn      ; Display stack with frame numbers
```

C.7 Thread and Process Inspection

```
!thread         ; Current thread
!process 0 1    ; List all processes
!process 0 7    ; Full process dump
!running        ; Running processors
```

C.8 Scheduler and CPU Topology (Windows 11 Hybrid CPUs)

```
!cpuinfo
!prcb
!ready
!runqueue
!irql
```

C.9 Memory Management and Paging

```
!vm
!memusage
!pte
```

```
!vtop
!address
```

C.10 Pool and Heap Debugging

```
!pool
!poolused
!poolfind
!heap
```

C.11 Object Manager and Handle Tables

```
!object
!handle
!objecttypes
```

C.12 I/O Manager and Driver Inspection

```
lm
lm vm drivername
!drvobj drivername 7
!devobj
!irp
```

C.13 Crash Dump and Bugcheck Analysis

```
!analyze -v
.bugcheck
.exr -1
.cxr -1
```

C.14 Secure Kernel, VBS, and HVCI Inspection

```
!securekernel
!hv
!hypervisor
!vtl
```

C.15 System Call and Execution Boundary Inspection

```
u nt!KiSystemCall64
x nt!Nt*
!syscalls
```

C.16 Lock, IRQL, APC, and DPC Inspection

```
!locks
!dpcs
!apc
!irql
```

C.17 Performance and Latency Analysis

```
!timer
!clock
!profile
```

C.18 Exploitation Surface & Security Boundaries

WinDbg enables direct forensic validation of:

- Use-after-free
- Pool overflow

- Token theft
- Handle table corruption
- VAD manipulation
- SMEP/SMAP bypass attempts
- CFG enforcement failures

C.19 Professional Kernel Engineering Notes

- Always match WinDbg version with Windows 11 build
- Symbol mismatches invalidate structure decoding
- IRQL violations are the most common root cause of deadlocks
- Kernel debugging automatically disables full system optimizations
- Live debugging of HVCI requires Hyper-V mediated sessions
- Never trust user-mode stack frames in kernel exploit analysis

Appendix D Windows 11 Calling Convention (Assembly)

D.1 Precise Windows 11Specific Definition

The Windows 11 x86-64 calling convention defines the **exact binary interface contract** between:

- User-mode code
- Kernel-mode code

- Drivers
- Native NTAPI
- Secure Kernel transitions

It specifies:

- Register-based argument passing
- Stack shadow space
- Stack alignment
- Volatile and non-volatile register sets
- Return value handling
- Call frame layout for structured unwinding

This ABI is **mandatory and immutable** across all Windows 11 kernel and user-mode execution paths.

D.2 Exact Architectural Role Inside Windows 11 Kernel

The calling convention is enforced at:

- User $\rightarrow$ Kernel transitions (`syscall/sysret`)
- Driver dispatch routines
- KMDF/WDM callback invocation
- Interrupt and exception trap frames

- Hypervisor-mediated Secure Kernel entry

Any violation results in:

- Stack corruption
- Register leakage
- Unwind failure
- Immediate system crash (BugCheck)

D.3 Core Register Classification

Integer Argument Registers

Argument	Register
Arg1	RCX
Arg2	RDX
Arg3	R8
Arg4	R9

Floating-Point Argument Registers

Argument	Register
Arg1	XMM0
Arg2	XMM1
Arg3	XMM2
Arg4	XMM3

Return Value Registers

- Integer return: `RAX`
- Floating return: `XMM0`

D.4 Volatile vs Non-Volatile Registers

Volatile (Caller-Saved)

```
RAX, RCX, RDX, R8, R9, R10, R11
XMM0XMM5
```

Non-Volatile (Callee-Saved)

```
RBX, RBP, RSI, RDI, R12R15
XMM6XMM15
```

Kernel-mode code **must preserve non-volatile registers explicitly**.

D.5 Stack Frame Rules

- Stack grows downward
- **32-byte shadow space is mandatory for every call**
- Stack must be **16-byte aligned at call instruction**
- Return address always pushed by `call`

D.6 Standard Stack Layout

```
|----------------------------|
| Caller Stack Parameters    |  <-- Above RSP at entry
|----------------------------|
| Shadow Space (32 bytes)    |  <-- Mandatory
|----------------------------|
| Return Address             |
|----------------------------|
| Local Variables            |
|----------------------------|  <-- RSP after prologue
```

D.7 External Assembly Function Example (User-Mode)

Assembly File: `add64.asm`

```
PUBLIC add64

.code
add64 PROC
    ; RCX = a, RDX = b
    mov rax, rcx
    add rax, rdx
    ret
add64 ENDP
END
```

C++ Prototype

```
extern "C" long long add64(long long a, long long b);
```

Usage

```
long long result = add64(10, 20);
```

D.8 Kernel-Mode External Assembly Example

Assembly File: `km_add.asm`

```
PUBLIC KmAdd

.code
KmAdd PROC
    ; Preserve non-volatile registers if used
    mov rax, rcx
    add rax, rdx
```

```
    ret
KmAdd ENDP
END
```

Kernel C Prototype

```
extern "C" LONGLONG KmAdd(LONGLONG A, LONGLONG B);
```

D.9 Execution Flow at Assembly Level

1. Caller places first four parameters in RCX, RDX, R8, R9
2. Caller allocates 32-byte shadow space
3. `call` pushes return address
4. Callee executes
5. Return value placed in RAX
6. `ret` restores control

D.10 Secure Kernel, VBS, and HVCI Interaction

- Secure Kernel transitions rely on **validated ABI frames**
- Shadow stacks (CET) enforce call/return integrity
- HVCI validates call targets at runtime
- Any ABI corruption is immediately trapped by hypervisor

D.11 Performance Implications

- Register-based arguments minimize memory traffic
- Shadow space eliminates dynamic stack probing
- Proper alignment avoids pipeline stalls
- Non-volatile register misuse causes severe performance degradation

D.12 Kernel Debugging & Inspection

```
r
k
kv
u rip
.dt _KTRAP_FRAME
```

D.13 Exploitation Surface & Security Boundaries

Incorrect calling convention usage enables:

- Stack pivot attacks
- Return address overwrite
- SMEP/SMAP bypass via corrupted frames
- ROP chain injection via misaligned frames
- CFG violations

D.14 Professional Kernel Engineering Notes

- Inline x64 assembly is **not supported** in MSVC
- All x64 assembly must be compiled as **external objects**
- Shadow space is mandatory even if unused
- Violating non-volatile preservation guarantees crash the kernel
- ABI is identical between User Mode and Kernel Mode
- CET shadow stacks further enforce ABI correctness in Windows 11

Appendix E Kernel Bug Check Codes

E.1 Precise Windows 11Specific Definition

A **Kernel Bug Check** is the final, non-recoverable safety mechanism of the Windows 11 kernel. It is triggered when the kernel detects a violation that compromises:

- Memory integrity
- Execution control
- Interrupt state consistency
- Security boundary enforcement
- Scheduler correctness

The Bug Check immediately:

- Raises IRQL to `HIGH_LEVEL`

- Freezes all logical processors
- Writes the crash dump via `KeDumpMachineState`
- Transfers control to the Windows crash handler

E.2 Exact Architectural Role Inside Windows 11 Kernel

Bug Checks act as the **last hardware-backed correctness barrier** protecting:

- Secure Kernel (VTL1)
- Hyper-V root partition
- Kernel memory pools
- Page table integrity
- Control Flow Guard enforcement

A Bug Check guarantees that once kernel integrity is broken, **no execution continues**.

E.3 Internal Kernel Data Structures

Bug Check Dispatcher

```
VOID KeBugCheckEx(
    ULONG BugCheckCode,
    ULONG_PTR Parameter1,
    ULONG_PTR Parameter2,
    ULONG_PTR Parameter3,
    ULONG_PTR Parameter4
);
```

Crash Parameter Storage

```
typedef struct _KBUGCHECK_DATA {
    ULONG BugCheckCode;
    ULONG_PTR Parameters[4];
} KBUGCHECK_DATA;
```

Trap Snapshot

```
typedef struct _KTRAP_FRAME {
    ULONG64 Rip;
    ULONG64 Rsp;
    ULONG64 Rflags;
    ULONG64 Cr3;
    ULONG64 ErrorCode;
} KTRAP_FRAME;
```

E.4 Execution Flow (Step-by-Step)

1. Kernel detects fatal violation
2. `KeBugCheckEx` invoked
3. Current processor state saved
4. Other CPUs halted via IPI
5. Dump subsystem initialized
6. Dump written to disk
7. System reset

E.5 Secure Kernel, VBS, and HVCI Interaction

- Secure Kernel validates dump integrity
- HVCI prevents compromised drivers from faking dump headers
- Hypervisor halts second-level execution contexts
- Any VTL violation escalates to immediate Bug Check

E.6 Performance Implications

- No performance cost during normal operation
- Dump generation consumes full I/O bandwidth
- Multiprocessor synchronization during Bug Check is O(N)

E.7 Real Practical Example (Manual Trigger in Driver)

Kernel C Example

```
#include <ntddk.h>

VOID TriggerBugCheck()
{
    KeBugCheckEx(
        MANUALLY_INITIATED_CRASH,
        0x11111111,
        0x22222222,
        0x33333333,
        0x44444444
    );
}
```

E.8 Kernel Debugging & Inspection

```
!analyze -v
.r
.kp
.trap
!thread
!process 0 1
```

E.9 Exploitation Surface & Security Boundaries

- Overwriting crash parameters enables false forensic trails
- Malicious dump filtering can hide exploit state
- Hypervisor-enforced dump integrity blocks tampering
- SMEP/SMAP violations escalate directly to Bug Check

E.10 Professional Kernel Engineering Notes

- Bug Checks must never be suppressed
- KeBugCheckEx is irreversible
- All drivers must treat Bug Check as a non-returning function
- Dump analysis is the primary root-cause tool for kernel failures

E.11 Critical Windows 11 Bug Check Codes

Code	Meaning
0x0000000A	IRQL_NOT_LESS_OR_EQUAL
0x0000001E	KMODE_EXCEPTION_NOT_HANDLED
0x0000003B	SYSTEM_SERVICE_EXCEPTION
0x00000050	PAGE_FAULT_IN_NONPAGED_AREA
0x0000007E	SYSTEM_THREAD_EXCEPTION_NOT_HANDLED
0x0000009F	DRIVER_POWER_STATE_FAILURE
0x000000C4	DRIVER_VERIFIER_DETECTED_VIOLATION
0x000000D1	DRIVER_IRQL_NOT_LESS_OR_EQUAL
0x00000109	CRITICAL_STRUCTURE_CORRUPTION
0x00000139	KERNEL_SECURITY_CHECK_FAILURE
0x00000154	UNEXPECTED_STORE_EXCEPTION
0x000001C7	FAST_FAT_FILE_SYSTEM
0x000001C8	KERNEL_MODE_HEAP_CORRUPTION
0x000001D5	HYPERVISOR_ERROR

E.12 Architecture-Specific Bug Check Classes

- Page Table Corruption
- VAD Tree Corruption
- Scheduler Queue Corruption
- Pool Header Corruption
- Secure Kernel Boundary Violations

E.13 Driver Engineering Pitfalls

- Invalid IRQL transitions
- Use-after-free in NonPagedPool
- Stack overruns in ISR/DPC
- Incorrect DMA buffer mapping
- Lock-free structure misuse

E.14 Final Engineering Summary

Bug Check codes are the **cryptographic fingerprints of kernel failure**. Correct interpretation requires:

- Precise IRQL awareness
- Full dump analysis
- Stack trace reconstruction
- Pool validation
- Secure Kernel state verification

No modern Windows 11 kernel forensic investigation is complete without deep Bug Check analysis.

Appendix F Kernel Memory Flags

F.1 Precise Windows 11Specific Definition

Kernel Memory Flags are bit-level attributes attached to virtual memory regions and pool allocations that define:

- Access permissions
- Caching behavior
- Executability
- Paging eligibility
- Security enforcement boundaries

In Windows 11, these flags are enforced simultaneously by:

- The Memory Manager (MM)
- The Secure Kernel (VTL1)
- Hypervisor Second-Level Address Translation (SLAT)
- SMEP, SMAP, and NX hardware features

F.2 Exact Architectural Role Inside Windows 11 Kernel

Kernel memory flags are consumed by:

- Page table entry (PTE) generation
- Virtual Address Descriptor (VAD) permissions

- Pool allocator state machines
- Secure Kernel integrity enforcement
- DMA remapping (IOMMU)

They dictate whether memory:

- May execute instructions
- May be accessed from user mode
- May be paged out
- May be cached

F.3 Internal Kernel Data Structures

Page Table Entry (x64)

```
typedef struct _MMPTE_HARDWARE {
    ULONG64 Valid : 1;
    ULONG64 Write : 1;
    ULONG64 Owner : 1;
    ULONG64 WriteThrough : 1;
    ULONG64 CacheDisable : 1;
    ULONG64 Accessed : 1;
    ULONG64 Dirty : 1;
    ULONG64 LargePage : 1;
    ULONG64 Global : 1;
    ULONG64 CopyOnWrite : 1;
    ULONG64 Prototype : 1;
    ULONG64 NoExecute : 1;
    ULONG64 PageFrameNumber : 36;
} MMPTE_HARDWARE;
```

Pool Descriptor Flags

```
typedef struct _POOL_DESCRIPTOR {
    ULONG PoolType;
    ULONG PoolIndex;
    ULONG RunningAllocs;
    ULONG RunningDeAllocs;
    ULONG PoolTag;
} POOL_DESCRIPTOR;
```

F.4 Execution Flow (Step-by-Step)

1. Driver requests memory via `ExAllocatePool2` or `MmAllocatePagesForMdl`
2. Memory Manager selects NUMA node and pool
3. PTE flags generated from allocation type
4. Hypervisor validates second-level permissions
5. SMEP/SMAP/NX hardware gates final access

F.5 Secure Kernel, VBS, and HVCI Interaction

- Executable pool requires Secure Kernel approval
- Writable + executable mappings are blocked by HVCI
- DMA buffers are validated by IOMMU against pool flags
- Kernel code pages are mapped as immutable inside VTL1

F.6 Performance Implications

- CacheDisabled memory disables L1/L2/L3 caching
- WriteThrough increases memory bus traffic
- NonPagedPool increases physical memory pressure
- LargePage flag reduces TLB pressure

F.7 Real Practical Example (NonPaged Pool Allocation)

```
#include <ntddk.h>

PVOID AllocateSecureBuffer(SIZE_T Size)
{
    return ExAllocatePool2(
        POOL_FLAG_NON_PAGED |
        POOL_FLAG_UNINITIALIZED,
        Size,
        'tseT'
    );
}
```

F.8 Kernel Debugging & Inspection

```
!poolfind tseT
!pte fffff803`12345000
!vad fffff803`12345000
!memusage
```

F.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Misconfigured NX allows shellcode execution
- CacheDisable misuse enables side-channel attacks
- RWX mappings immediately violate HVCI
- Improper MDL mapping enables DMA corruption

F.10 Professional Kernel Engineering Notes

- Never allocate executable pool unless absolutely required
- Always prefer NonPagedPoolNx
- DMA buffers must always be CacheDisabled
- Never mix user-accessible and kernel-owned mappings

F.11 Primary Kernel Memory Flags (Windows 11)

Flag	Meaning
`POOL_FLAG_NON_PAGED`	Cannot be paged out
`POOL_FLAG_PAGED`	Pageable kernel memory
`POOL_FLAG_NON_PAGED_EXECUTE`	Executable kernel memory
`POOL_FLAG_CACHE_ALIGNED`	Cache-line alignment enforced
`POOL_FLAG_UNINITIALIZED`	Allocation not zeroed
`POOL_FLAG_SECURE`	Protected by Secure Kernel

F.12 Page Table Flag Bits (x64 Hardware)

Bit	Function
0	Present
1	Writeable
2	User Accessible
3	Write Through
4	Cache Disable
7	Large Page
8	Global
63	No Execute (NX)

F.13 Driver Engineering Pitfalls

- Executing from paged memory at IRQL > APC
- Writing to CacheDisabled memory without fencing
- Mapping user buffers without SMAP guards
- Reusing freed NonPagedPool blocks

F.14 Final Engineering Summary

Kernel memory flags are a **hardware-enforced security contract** between:

- The CPU
- The Hypervisor
- The Secure Kernel

- The Windows Memory Manager

Incorrect flag usage immediately leads to:

- Bug Checks
- HVCI violations
- Silent memory corruption
- DMA-based exploitation

Correct flag discipline is mandatory for all Windows 11 kernel engineers.

Appendix G IRQL & Execution Rules

G.1 Precise Windows 11Specific Definition

IRQL (Interrupt Request Level) is the hardware-enforced execution priority scheme used by the Windows 11 kernel to:

- Serialize access to shared kernel resources
- Control interrupt preemption
- Enforce safe memory access rules
- Regulate APC, DPC, and thread dispatching

IRQL is not a software abstraction; it directly maps to **CPU interrupt masking semantics** on x86-64 and is further enforced by Hyper-V in VBS-enabled systems.

G.2 Exact Architectural Role Inside Windows 11 Kernel

IRQL governs:

- Whether thread scheduling is permitted
- Whether page faults are legal
- Whether spinlocks may be acquired
- Whether APCs or DPCs may execute
- Whether interrupts are masked at the LAPIC

IRQL transitions occur on:

- Interrupt entry/exit paths
- Dispatcher database locking
- Spinlock acquisition
- DPC and ISR execution

G.3 Internal Kernel Data Structures

KIRQL Type

```
typedef UCHAR KIRQL;
```

KTHREAD IRQL Field

```
typedef struct _KTHREAD {
    ...
    KIRQL CurrentIrql;
    ...
} KTHREAD;
```

KPCR IRQL Storage

```
typedef struct _KPCR {
    ...
    KIRQL Irql;
    ...
} KPCR;
```

G.4 Execution Flow (Step-by-Step at C & Assembly Level)

IRQL Raise Path

1. Driver calls `KeRaiseIrql`
2. CPU interrupt mask level updated
3. Scheduler preemption disabled
4. APC delivery blocked

IRQL Lower Path

1. Driver calls `KeLowerIrql`
2. CPU interrupt level restored
3. Pending DPCs may execute
4. Scheduler resumes normal operation

x86-64 Assembly (Conceptual Hardware Effect)

```
; IRQL effect (conceptual)
mov cr8, rax    ; CR8 controls task priority register (TPR)
```

G.5 Secure Kernel, VBS, and HVCI Interaction

- Hyper-V mirrors IRQL masking into VTL1 interrupt controls
- Secure Kernel enforces that pageable code never executes above APC_LEVEL
- HVCI validates ISR and DPC target code pages before dispatch
- DMA interrupts are filtered through IOMMU policies before ISR delivery

G.6 Performance Implications

- Prolonged execution above DISPATCH_LEVEL causes scheduler starvation
- Excessive spinlock holding at elevated IRQL causes global CPU stalls
- DPC storms at DIRQL increase interrupt latency
- Cache-line bouncing is amplified by spinlock contention at high IRQL

G.7 Real Practical Example (Safe IRQL Usage in Driver)

```
KIRQL oldIrql;

KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);

/* Critical section: spinlocks, nonpaged memory only */

KeLowerIrql(oldIrql);
```

Illegal Operation Example

```
/* BUGCHECK: Page fault at DISPATCH_LEVEL */
PVOID p = ExAllocatePool(PagedPool, 0x100);
```

G.8 Kernel Debugging & Inspection

```
!irql
!thread
!cpuinfo
!stacks 2
!dpcs
```

G.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Use-after-free triggered at DISPATCH_LEVEL bypasses normal recovery
- Spinlock corruption leads to permanent system deadlock
- Malformed ISR leads to immediate triple fault if IRQL stack is corrupted
- IRQL confusion bugs allow timing-based privilege escalation

G.10 Professional Kernel Engineering Notes

- Never access pageable memory above APC_LEVEL
- Never sleep or wait above PASSIVE_LEVEL
- Never call `Zw/Nt` syscalls above PASSIVE_LEVEL
- Never perform disk, registry, or file I/O above APC_LEVEL

G.11 Windows 11 IRQL Level Table

IRQL	Value	Purpose
PASSIVE_LEVEL	0	Normal thread execution
APC_LEVEL	1	APC delivery
DISPATCH_LEVEL	2	DPC execution, spinlocks
DIRQL	3–15	Hardware interrupt handling
CLOCK_LEVEL	Platform	System timer interrupt
IPI_LEVEL	Platform	Inter-processor interrupts
HIGH_LEVEL	31	System halt level

G.12 IRQL vs Allowed Operations

Operation	PASSIVE	APC	DISPATCH+
Memory Paging	Yes	No	No
Thread Sleep	Yes	No	No
Spinlock Use	No	Yes	Yes
File I/O	Yes	No	No
Zw/Nt Syscalls	Yes	No	No
DPC Execution	No	No	Yes

G.13 Common IRQL Bug Checks

- `IRQL_NOT_LESS_OR_EQUAL`
- `DRIVER_IRQL_NOT_LESS_OR_EQUAL`
- `DPC_WATCHDOG_VIOLATION`
- `ATTEMPTED_WRITE_TO_READONLY_MEMORY`

G.14 Final Engineering Summary

IRQL is the **hard real-time safety boundary** of the Windows 11 kernel:

- It enforces memory safety
- It regulates interrupt flow
- It protects scheduler integrity
- It prevents illegal blocking in atomic contexts

Violating IRQL rules results in:

- Immediate system crashes
- Deadlocks
- Unrecoverable memory corruption
- Hypervisor security violations

Strict IRQL discipline is mandatory for all production-grade Windows 11 kernel engineers.

Appendix H Secure Boot & TPM Tables

H.1 Precise Windows 11Specific Definition

Secure Boot in Windows 11 is a UEFI-enforced cryptographic verification chain that guarantees only trusted firmware, bootloaders, and kernel components are executed during system initialization. **TPM 2.0 (Trusted Platform Module)** is a discrete or firmware-based cryptographic processor used to measure, seal, attest, and protect platform integrity state. Windows 11 **mandates**:

- UEFI Secure Boot enabled
- TPM 2.0 present and active
- Measured boot enforced

Secure Boot validates execution. TPM records execution.

H.2 Exact Architectural Role Inside Windows 11 Kernel

Secure Boot and TPM participate in:

- Pre-kernel trust establishment
- Boot-time code integrity enforcement
- Hypervisor root-of-trust validation
- Secure Kernel (VTL1) initialization
- BitLocker key sealing
- HVCI policy enforcement

Windows 11 boot trust chain:

1. UEFI Firmware
2. Boot Manager (`bootmgfw.efi`)
3. Windows OS Loader (`winload.efi`)
4. Hypervisor (`hvix64.exe`)
5. Secure Kernel (SK)
6. NT Kernel (`ntoskrnl.exe`)

Each stage is cryptographically verified before execution.

H.3 Internal Kernel Data Structures

TPM PCR Representation

```
typedef struct _TPM_PCR {
    UINT32 PcrIndex;
    BYTE   Digest[32];
} TPM_PCR;
```

Secure Boot Policy Structure (Abstracted)

```
typedef struct _SECURE_BOOT_POLICY {
    BOOLEAN SecureBootEnabled;
    BOOLEAN TestSigningEnabled;
    BOOLEAN DebugPolicyEnabled;
} SECURE_BOOT_POLICY;
```

Kernel Secure Boot State

```
typedef struct _KERNEL_SECURE_STATE {
    BOOLEAN HvciEnabled;
    BOOLEAN VbsEnabled;
    BOOLEAN SecureKernelActive;
} KERNEL_SECURE_STATE;
```

H.4 Execution Flow (Step-by-Step at C & Assembly Level)

Boot-Time Measurement Flow

1. UEFI loads Boot Manager and extends TPM PCR[0]
2. Boot Manager loads OS Loader and extends PCR[4]
3. OS Loader validates kernel signature and extends PCR[7]
4. Hypervisor initializes Secure Kernel and finalizes PCRs

Conceptual x86-64 Measurement Trigger (Firmware Level)

```
; Conceptual measurement operation (firmware-managed)
mov rax, OFFSET BootComponentHash
call TpmExtendPcr
```

Kernel Query of Secure Boot State

```
BOOLEAN secure = 0;
ZwQuerySystemInformation(SystemSecureBootInformation,
                         &secure,
                         sizeof(secure),
                         NULL);
```

H.5 Secure Kernel, VBS, and HVCI Interaction

- Secure Boot guarantees initial trust
- TPM proves that trust via PCRs
- Hyper-V isolates Secure Kernel in VTL1
- HVCI uses Secure Kernel trust to validate all kernel code pages
- Kernel-mode drivers must be signed by a Microsoft-trusted root

Without Secure Boot + TPM:

- HVCI cannot be enforced
- Memory integrity is disabled
- Kernel trust is downgraded

H.6 Performance Implications

- Measured boot adds cryptographic hashing overhead during boot
- TPM I/O introduces latency during key sealing
- HVCI adds runtime page validation overhead
- DMA remapping incurs IOMMU translation penalties

However, there is **no steady-state performance impact** on:

- Scheduler operation
- APC/DPC dispatch
- Thread switching

H.7 Real Practical Example (TPM PCR Inspection)

TPM PCR Dump via WinDbg

```
!tpm
!pcrs
```

Kernel Query of VBS State

```
SYSTEM_VBS_INFORMATION vbs = {0};

ZwQuerySystemInformation(SystemVbsInformation,
                         &vbs,
                         sizeof(vbs),
                         NULL);
```

H.8 Kernel Debugging & Inspection

```
!secureboot
!hvinfo
!sysinfo secure
!sysinfo hv
!drvobj hvix64 2
```

H.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Bootkit attacks require Secure Boot disablement
- PCR replay attacks require physical TPM compromise
- DMA attacks blocked by Secure Kernel + IOMMU
- Kernel rootkits blocked by HVCI page signing enforcement

Secure Boot + TPM eliminate:

- Pre-boot persistent malware
- Unsigned bootloader injection
- Kernel code patching
- Hypervisor subversion

H.10 Professional Kernel Engineering Notes

- Never rely on runtime integrity without Secure Boot
- HVCI requires Secure Boot + TPM + Hypervisor

- Test-signing disables memory integrity
- Secure Kernel cannot be patched or debugged conventionally

H.11 TPM PCR Usage Table

PCR	Measured Component
0	Firmware Core
2	Option ROMs
4	Boot Manager
7	Secure Boot Policy
11	BitLocker State
12	Hypervisor Launch

H.12 Secure Boot Policy States

Policy	Enabled	Security Impact
Secure Boot	Yes	Enforces chain-of-trust
Test Signing	No	Prevents unsigned drivers
Debug Mode	No	Blocks kernel debugging bypass

H.13 Common Secure Boot Bug Checks

- `SECURE_BOOT_VIOLATION`
- `KERNEL_SECURITY_CHECK_FAILURE`
- `ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY`

H.14 Final Engineering Summary

Secure Boot and TPM form the **root-of-trust foundation** of Windows 11:

- Secure Boot enforces execution integrity
- TPM validates platform authenticity
- Secure Kernel isolates trust logic
- HVCI seals all kernel code execution

Without Secure Boot and TPM:

- Kernel trust collapses
- Hypervisor protections degrade
- Driver signing enforcement weakens
- System becomes permanently vulnerable to bootkits

All Windows 11 kernel security enforcement ultimately anchors in Secure Boot and TPM.

Appendix I DMA & MDL Structures

I.1 Precise Windows 11Specific Definition

Direct Memory Access (DMA) in Windows 11 is a hardware-assisted data transfer mechanism that allows peripheral devices to read and write system memory without continuous CPU intervention.
A **Memory Descriptor List (MDL)** is a kernel data structure that describes the physical memory layout backing a virtual buffer. MDLs are mandatory for:

- DMA operations
- Locked I/O buffers
- Zero-copy network and storage drivers
- Secure memory mapping between user and kernel space

Windows 11 enforces DMA isolation through:

- IOMMU (Intel VT-d / AMD-Vi)
- Hyper-V DMA remapping
- Secure Kernelenforced DMA protections

I.2 Exact Architectural Role Inside Windows 11 Kernel

DMA and MDLs directly participate in:

- Storage I/O path (NVMe, AHCI)
- Network data path (NDIS, TCP/IP offload)
- GPU memory transfer
- USB and PCIe device transfers

Architectural integration layers:

- I/O Manager
- HAL DMA Abstraction
- IOMMU Translation Engine

- Secure Kernel DMA Guard

Execution boundary:

- User buffers → MDL translation → DMA physical pages

I.3 Internal Kernel Data Structures

MDL Structure (Windows 11)

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

DMA Adapter Structure (Abstracted)

```
typedef struct _DMA_ADAPTER {
    USHORT Version;
    USHORT Size;
    PVOID  DmaOperations;
} DMA_ADAPTER, *PDMA_ADAPTER;
```

Scatter/Gather List

```
typedef struct _SCATTER_GATHER_ELEMENT {
    PHYSICAL_ADDRESS Address;
    ULONG Length;
```

```
    ULONG_PTR Reserved;
} SCATTER_GATHER_ELEMENT;

typedef struct _SCATTER_GATHER_LIST {
    ULONG NumberOfElements;
    SCATTER_GATHER_ELEMENT Elements[1];
} SCATTER_GATHER_LIST;
```

I.4 Execution Flow (Step-by-Step at C & Assembly Level)

DMA Read Operation Flow

1. Driver receives I/O request
2. User buffer is locked
3. MDL is built with `IoAllocateMdl`
4. Pages are probed with `MmProbeAndLockPages`
5. DMA mapping created via HAL
6. Device performs bus-master transfer
7. DMA completion interrupt fires
8. MDL is unlocked and released

MDL Creation Flow

```
PMDL mdl = IoAllocateMdl(Buffer, Length, FALSE, FALSE, NULL);
MmProbeAndLockPages(mdl, KernelMode, IoReadAccess);
```

Conceptual DMA Address Load (External Assembly)

```
; rdx = Physical DMA Address
mov rax, [rdx]
mov [DeviceDmaRegister], rax
```

I.5 Secure Kernel, VBS, and HVCI Interaction

Windows 11 enforces:

- DMA remapping through IOMMU
- DMA access blocked from Secure Kernel (VTL1) memory
- HVCI forbids unsigned DMA-capable drivers
- Kernel DMA Guard prevents pre-boot DMA injection

Secure Kernel validates:

- DMA-capable PCIe device authorization
- Hypervisor IOMMU page tables
- SMM DMA isolation

I.6 Performance Implications

- MDL locking introduces page-fault suppression cost
- IOMMU translation adds DMA address latency
- Scatter/gather list size affects cache locality
- Large contiguous DMA improves throughput

Optimal performance requires:

- Page-aligned buffers
- Minimal MDL chains
- NUMA-local DMA allocation

I.7 Real Practical Example

KMDF DMA Initialization

```
WDF_DMA_ENABLER_CONFIG dmaConfig;
WDF_DMA_ENABLER_CONFIG_INIT(&dmaConfig,
                            WdfDmaProfileScatterGather64,
                            MAX_TRANSFER_SIZE);

WdfDmaEnablerCreate(Device,
                    &dmaConfig,
                    WDF_NO_OBJECT_ATTRIBUTES,
                    &DmaEnabler);
```

Building a DMA Transaction

```
WdfDmaTransactionInitializeUsingRequest(
    DmaTransaction,
    Request,
    EvtProgramDma,
    Direction
);
```

I.8 Kernel Debugging & Inspection

```
!mdl <address>
```

```
!dma
!iommu
!devnode 0 1
!wdfkd.wdfdmaenabler <address>
```

I.9 Exploitation Surface, Attack Vectors & Security Boundaries

- DMA attacks via malicious PCIe devices
- Thunderbolt DMA injection (blocked by Kernel DMA Guard)
- MDL corruption leading to arbitrary physical memory access
- IOMMU misconfiguration enabling VTL bypass

Windows 11 blocks:

- Unauthorized DMA pre-boot
- Cross-VTL DMA writes
- Kernel page table corruption via DMA

I.10 Professional Kernel Engineering Notes

- Never assume DMA buffers are cache-coherent
- Always unlock MDLs after I/O completion
- Never expose physical addresses to user-mode
- Always use DMA Guardcompatible device profiles
- Never reuse MDLs across IRP lifetimes

I.11 MDL Flag Reference

Flag	Meaning
MDL_MAPPED_TO_SYSTEM_VA	Kernel mapping created
MDL_PAGES_LOCKED	Physical pages locked
MDL_SOURCE_IS_NONPAGED_POOL	Nonpaged source buffer
MDL_IO_PAGE_READ	Read DMA operation
MDL_IO_PAGE_WRITE	Write DMA operation

I.12 DMA Security Enforcement Table

Protection	Enforcement Layer
Kernel DMA Guard	Secure Kernel
IOMMU Remapping	Hypervisor
PCIe Authorization	ACPI + SK
Bus Master Control	HAL

I.13 Common DMA-Related Bug Checks

- `DRIVER_VERIFIER_DMA_VIOLATION`
- `IOMMU_FAULT`
- `SYSTEM_SERVICE_EXCEPTION`
- `DRIVER_IRQL_NOT_LESS_OR_EQUAL`

I.14 Final Engineering Summary

DMA and MDLs form the **physical memory transfer backbone** of Windows 11:

- MDLs translate virtual buffers to physical pages
- DMA engines move data without CPU overhead
- IOMMU enforces secure physical isolation
- Secure Kernel guarantees DMA trust boundaries

Any MDL misuse directly results in:

- Physical memory corruption
- Kernel compromise
- Secure Kernel violation
- Immediate system crash

Appendix J Windows 11 Build Differences

J.1 Precise Windows 11Specific Definition

Windows 11 Build Differences refer to kernel-level architectural, security, scheduler, memory manager, and virtualization changes introduced across Windows 11 releases (21H2, 22H2, 23H2, and post-23H2 cumulative builds). These differences directly affect:

- Kernel data structure layouts
- System call dispatch paths
- Hypervisor and Secure Kernel enforcement
- IOMMU, DMA, and memory isolation policies

All Windows 11 builds share the same fundamental NT architecture, but internal implementations evolve continuously.

J.2 Exact Architectural Role Inside Windows 11 Kernel

Build-level differences influence:

- **ntoskrnl.exe** internal symbol layouts
- **Secure Kernel** policy enforcement
- **Scheduler** heterogenous CPU scheduling behavior
- **Memory Manager** large-page, compression, and VAD handling
- **I/O Manager** DMA and storage stack optimizations

Each build modifies execution paths inside:

- System call entry
- Context switching
- Interrupt dispatch
- Page fault resolution

J.3 Internal Kernel Data Structure Variability

EPROCESS Field Drift Across builds, the following internal offsets may change:

- `DirectoryTableBase`
- `VadRoot`
- `SeAuditProcessCreationInfo`
- `ImageFileName`

KTHREAD Changes

- Scheduler quantum storage
- Hybrid CPU core classification
- Affinity bitmap expansion

Secure Kernel Data Extensions

- VTL1 policy tables
- HVCI verification descriptors
- DMA Guard authorization maps

J.4 Execution Flow Differences (Step-by-Step)

System Call Path Evolution

1. `syscall` instruction executes
2. Entry transitions from Ring 3 to Ring 0
3. Dispatch table validation (HVCI enforced)
4. Secure Kernel shadow validation (VTL1)
5. NT system service table resolution

Post-22H2 builds add:

- Extended CET shadow stack validation
- Additional hypervisor-mediated checks

Context Switch Enhancements

- Core-class-aware run queue selection
- NUMA-aware thread migration
- Secure Kernel context tagging

J.5 Secure Kernel, VBS, and HVCI Interaction

Build differences strongly affect:

- Secure Kernel page table layout
- Enforced HVCI policy strictness
- DMA Guard blocking thresholds

Evolution Across Builds

- 21H2: Initial Secure Kernel enforcement
- 22H2: Mandatory HVCI on many OEM devices
- 23H2+: Hypervisor-assisted kernel shadow validation expanded

J.6 Performance Implications

- Additional hypervisor transitions slightly increase syscall latency
- Improved scheduler logic reduces core migration overhead
- Enhanced memory compression reduces paging pressure
- Stronger DMA isolation adds minimal IOMMU translation cost

Overall trend: **Security increases with minimal but measurable kernel execution overhead.**

J.7 Real Practical Example

Kernel Build Detection Using NTAPI

```
RTL_OSVERSIONINFOW info = {0};
info.dwOSVersionInfoSize = sizeof(info);
RtlGetVersion(&info);
```

Build-Based Feature Gate

```
if (info.dwBuildNumber >= 22621) {
    EnableEnhancedHvciPolicy();
}
```

External Assembly (Syscall Entry Observation)

```
mov r10, rcx
mov eax, SyscallIndex
syscall
ret
```

J.8 Kernel Debugging & Inspection

```
vertarget
!version
dt nt!_EPROCESS
dt nt!_KTHREAD
!syscall
!securekernel
```

J.9 Exploitation Surface, Attack Vectors & Security Boundaries

- Kernel exploit offsets break across builds

- Secure Kernel blocks legacy kernel shellcode patterns
- DMA attacks fully blocked post-22H2
- PatchGuard expands integrity scope each build

Exploit reliability strictly decreases with newer builds due to:

- Increased randomization
- Stronger hypervisor mediation
- Enforced CET shadow stacks

J.10 Professional Kernel Engineering Notes

- Never hard-code kernel offsets across builds
- Always validate build number before structure parsing
- Secure Kernel behavior changes between cumulative updates
- HVCI enforcement is non-negotiable in modern deployments
- Debug symbol mismatches cause silent analysis corruption

J.11 Windows 11 Build Evolution Table

Build	Security Level	Kernel Impact
21H2	High	Initial VBS adoption
22H2	Very High	Mandatory HVCI on OEMs
23H2	Extreme	Extended Secure Kernel enforcement
Post-23H2	Maximum	Hypervisor shadow validation

J.12 Final Engineering Summary

Windows 11 kernel builds are not cosmetic updates. Each build:

- Alters Secure Kernel behavior
- Tightens DMA and memory isolation
- Hardens syscall dispatch
- Expands hypervisor control of kernel execution

Ignoring build differences in kernel engineering results in:

- Invalid structure parsing
- Broken debugging sessions
- Non-deterministic crashes
- Immediate security violations

All serious Windows 11 kernel work **must be build-aware at all times**.

References

- Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*. Volumes 14.
- AMD Corporation. *AMD64 Architecture Programmers Manual*. Volumes 15.
- Microsoft. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More*. 7th Edition.
- Microsoft. *Windows Internals, Part 2: I/O System, Storage, Networking, and Security*. 7th Edition.
- Microsoft. *Windows Driver Kit (WDK) Documentation*.
- Microsoft. *Hyper-V Architecture and Virtualization-Based Security Documentation*.
- Microsoft. *Windows Security Internals: Secure Kernel, VBS, and HVCI*.
- UEFI Forum. *Unified Extensible Firmware Interface (UEFI) Specification*.
- Trusted Computing Group. *TPM 2.0 Library Specification*.
- Russinovich, M., Solomon, D., Ionescu, A. *Windows via C/C++ System Programming*.
- Microsoft. *NT Native API Reference and Undocumented Structures*.

- Microsoft. *WinDbg and Kernel Debugging Reference*.
- Intel Corporation. *Virtualization Technology (VT-x, EPT) Technical Documentation*.
- AMD Corporation. *Secure Virtual Machine (SVM) and Nested Page Tables (NPT)*.
- Microsoft. *Memory Manager, VAD, and Page Table Implementation Notes*.
- Microsoft. *I/O Manager, Plug and Play, and Power Management Internals*.
- Microsoft. *Control Flow Guard (CFG), Kernel CFG, and XFG Internals*.
- Microsoft. *Kernel-Mode Code Integrity (KMCI) and HVCI Architecture*.
- Microsoft. *IRQL, APC, and DPC Execution Rules*.
- PCI-SIG. *PCI Express Base Specification*.
- Microsoft. *DMA Remapping, IOMMU, and Kernel DMA Protection*.