MEMORY-RESIDENT MALWARE AND FILELESS ATTACK DETECTION

WINDOWS INTERNALS, MEMORY FORENSICS, EDR, AND NETWORK DETECTION ENGINEERING

WINDOWS INTERNALS
Deep dive into the architecture and execution model

MALWARE & EXECUTION SURFACES
PE/COFF structures, loaders, and managed execution

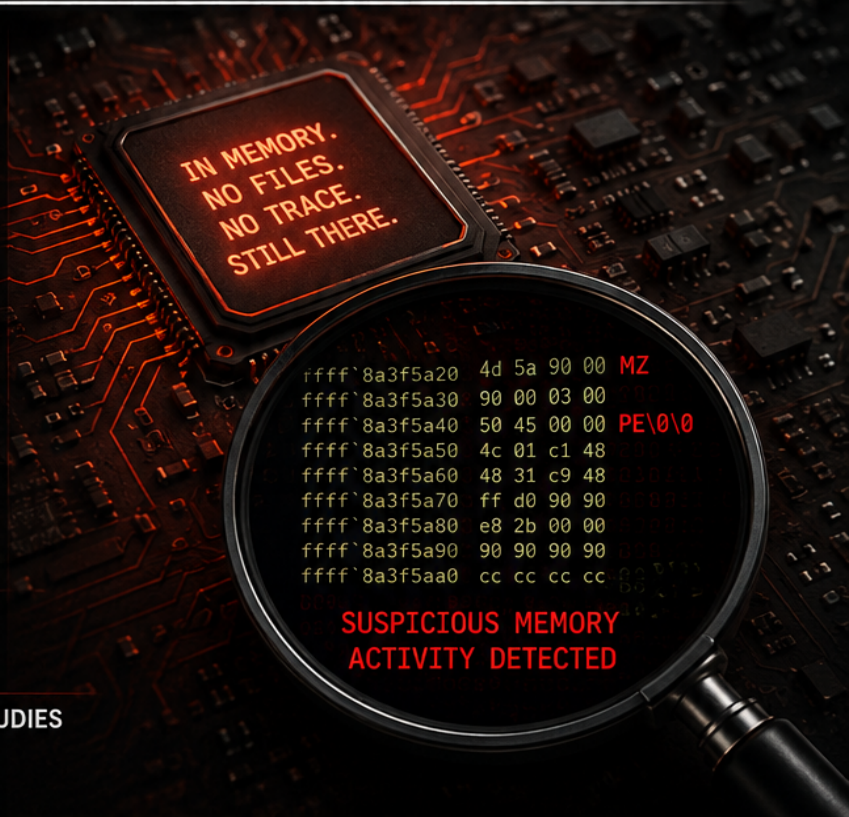
TELEMETRY & EDR ARCHITECTURE
ETW, Sysmon, Windows Events, and modern EDR pipelines

MEMORY FORENSICS
Acquire, analyze, and extract high-value artifacts

NETWORK DETECTION ENGINEERING
Zeek, Suricata, TLS, DNS, and detection at scale

DETECTION METHODOLOGY
From ATT&CK mapping to Sigma rules and analytics

HANDS-ON LABS & CASE STUDIES
Reproducible labs, reference implementations, and real-world investigations



ALL OFFENSIVE DEMONSTRATIONS USE BENIGN SIMULATORS, SYNTHETIC ARTIFACTS, INERT FIXTURES, OR DELIBERATELY NON-DEPLOYABLE CONSTRUCTS.
NO OPERATIONAL MALWARE OR WEAPONIZED CONTENT IS PROVIDED.

CAMERON LANGFORD

Memory-Resident Malware and Fileless Attack Detection

Windows Internals, Memory Forensics, EDR, and Network Detection Engineering

Steve Publications

This book is available at
https://leanpub.com/memory-residentmalwareandfilelessattackdetection

This version was published on 2026-08-26

Contents

Windows Internals, Memory Forensics, EDR, and Network Detection Engineering

This book teaches security engineers, SOC analysts, incident responders, threat hunters, detection engineers, malware analysts, DFIR practitioners, reverse engineers, Windows systems engineers, and advanced cybersecurity researchers how to understand, detect, investigate, contain, and validate defenses against memory-resident and fileless threats on Windows. It covers Windows internals relevant to detection, PE/COFF structures, managed execution surfaces, telemetry infrastructure, threat taxonomy, memory forensics, EDR architecture, network detection engineering, reproducible lab construction, detection methodology, reference implementations, forensic case studies, and production operations. All offensive demonstrations use benign simulators, synthetic artifacts, inert fixtures, or deliberately non-deployable constructs; no operational malware, credential stealers, persistence implants, EDR bypasses, stealth loaders, shellcode launchers, or weaponized injection frameworks are provided.

Introduction: The Memory-Only Battlefield

Why "Fileless" Is a Misleading Term

The word fileless has become shorthand in security discussions for a broad class of attacks that seem to operate without leaving traditional artifacts on disk. In practice, very few real-world techniques are truly fileless. A malicious document still exists somewhere. A PowerShell script may be downloaded from a web server and run entirely in memory, but the server hosting it is writing files. A process-injected payload lives only in RAM while running, yet the injector that placed it there was almost certainly loaded from disk at some point.

The term persists because it captures something real: attackers increasingly rely on techniques that avoid dropping a recognizable executable file into a location where an antivirus scanner or endpoint protection product can inspect it statically. They run code inside legitimate processes, they execute scripts directly from memory, they abuse built-in Windows tools and frameworks, and they stage payloads in ways that make disk-centric detection unreliable.

For defenders, the more useful framing is memory-resident threats: techniques whose critical execution phase occurs primarily in RAM rather than as a standalone executable file on persistent storage. This framing matters because it directs attention to the right signals. If you are looking only for suspicious files, you will miss activity that never materializes as one. If you understand how Windows manages processes, memory, modules, and execution contexts, you can recognize anomalous behavior regardless of whether anything landed on disk.

The modern Windows operating system is designed around flexibility and extensibility. It provides scripting engines like PowerShell, automation frameworks like WMI, COM infrastructure for inter-process communication, dynamic linking mechanisms that load libraries at runtime, memory allocation

APIs that allow processes to request executable memory regions, and a rich set of APIs for interacting with other processes. All of these features are legitimate and essential for normal operations. The same features also provide the building blocks for techniques that operate primarily in memory.

Defenders who understand the underlying mechanisms can distinguish normal behavior from suspicious behavior by looking at patterns: which process created which child, what arguments were passed, what modules were loaded, how memory permissions changed over time, what network connections were established, and whether any of these signals deviate from expected baselines. Memory-resident techniques are not magic; they exploit documented operating system behaviors that leave observable artifacts across multiple telemetry sources.

What This Book Teaches You to Do

This book is designed as a practical reference for security professionals who need to detect, investigate, and defend against memory-resident threats on Windows. It assumes you already understand basic networking concepts, have some familiarity with operating system fundamentals, and are comfortable reading logs, command output, and technical documentation. It does not assume expertise in Windows internals, reverse engineering, or advanced threat hunting, but it will take you to that level if you follow through.

By the end of this book, you should be able to:

- Understand how Windows manages processes, threads, virtual memory, modules, handles, tokens, and security descriptors at a level sufficient for recognizing suspicious behavior patterns.
- Read and interpret PE/COFF headers, section attributes, import/export tables, and loader metadata to distinguish normal from anomalous module states across filesystem, loader, and memory views.
- Identify observable artifacts of common memory-resident techniques including process injection indicators, hollowing-like anomalies, reflective loading signals, suspicious inter-process access, token misuse patterns, and living-off-the-land behavior.
- Configure and consume Windows telemetry sources including ETW providers, Event Logs, PowerShell logging, Sysmon-style extended

telemetry, AMSI integration points, and Microsoft Defender interfaces to build effective detection coverage.
- Acquire and analyze Windows memory images using contemporary forensic frameworks to enumerate processes, inspect VAD trees, identify executable memory regions, detect permission anomalies, examine loaded modules, recover network artifacts, scan with YARA rules, and reconstruct timelines.
- Design, implement, test, and operate detection rules that target specific threat behaviors rather than relying solely on static indicators of compromise, using frameworks like Sigma for vendor-neutral rule authoring.
- Analyze network traffic to detect command-and-control patterns, DNS anomalies, TLS fingerprint signals, beacon-like periodicity, and protocol-level irregularities even when payload contents are encrypted.
- Build and maintain an isolated defensive research laboratory that supports safe telemetry collection, synthetic attack simulation, memory acquisition, packet capture, detection validation, and forensic analysis without risking production systems.
- Apply systematic detection engineering methodology including threat-informed hypothesis formulation, observable selection, telemetry requirements analysis, rule versioning, testing strategies, precision-recall trade-off management, coverage measurement, and purple-team validation.
- Work through realistic forensic investigations from initial alert through triage, evidence collection, hypothesis testing, detection refinement, containment recommendations, root cause analysis, and lessons learned.
- Deploy and operate reference tooling in production environments with appropriate security hardening, monitoring, retention policies, incident response procedures, and continuous improvement practices.

The book is organized as a progressive learning path that also functions as a desk reference. Early chapters establish foundational knowledge about Windows internals, PE structures, execution surfaces, and telemetry infrastructure. Middle chapters cover threat taxonomy, memory forensics, EDR architecture, and network detection engineering. Later chapters focus on practical implementation including lab construction, detection methodology, reference tooling, case studies, and production operations. Each chapter builds on previous material while remaining self-contained enough to consult independently when you encounter a specific problem.

Defensive Ethics, Safe Research, and Controlled Simulation

Studying memory-resident techniques requires examining how adversaries manipulate Windows mechanisms to execute code, evade detection, and maintain access. This examination must be conducted ethically and safely. The research model this book follows has three core principles: authorization, isolation, and defense-first orientation.

Authorization means you only analyze systems, networks, and data that you own or have explicit permission to examine. Running tests on production systems without approval can disrupt services, violate compliance requirements, and potentially cause legal consequences. Even in your own environment, consider the impact of aggressive testing: memory acquisition can consume significant resources, network packet capture may involve sensitive data, and some simulation techniques can trigger security controls that escalate alerts or block access.

Isolation means conducting research and testing in environments that cannot affect production systems or networks. A properly isolated lab uses virtualization with host-only or internal networking, snapshots for rapid reset, segregated storage for evidence and samples, time synchronization that does not interfere with production NTP configurations, and strict controls on how data moves between the lab and external systems. Chapter 9 provides detailed guidance on constructing such an environment.

Defense-first orientation means every technique is explained from the perspective of detection, investigation, and defense rather than deployment. When discussing how a particular memory-resident technique works architecturally, the focus is on what observable side effects it produces, what forensic traces remain, what telemetry sources expose it, and how defenders can recognize and respond to it. Where code examples are educationally valuable for understanding mechanisms or building defensive tools, they implement safe components like telemetry collectors, PE parsers, memory map inspectors operating on authorized test processes, event consumers, log normalizers, rule engines, correlation pipelines, forensic parsers, detection validators, IOC scanners, benign anomaly generators, and analysis utilities. Code that would materially facilitate malicious injection, stealth, credential theft, security-control disabling, or persistence is replaced with non-operational pseudocode,

structural explanations, prerecorded artifacts, or benign simulations.

This approach aligns with professional standards in cybersecurity research and incident response. It ensures the knowledge you gain from this book can be applied immediately to improve your organization's defensive capabilities without creating new risks through accidental misuse or inadequate safeguards.

The Reference Laboratory Architecture

Throughout this book, examples, demonstrations, and exercises assume a consistent reference laboratory environment. This is not an arbitrary choice: having a reproducible lab architecture allows you to follow along with practical examples, validate concepts hands-on, build your own detection rules against real telemetry, practice memory acquisition and analysis workflows, and develop muscle memory for investigation procedures.

The reference lab consists of the following components:

- A host machine running a supported hypervisor such as VMware Workstation Pro, VirtualBox, or Hyper-V with sufficient CPU cores, RAM, and storage to run multiple virtual machines concurrently without performance degradation.
- One or more Windows 10 or Windows 11 virtual machines configured as test endpoints with telemetry collection enabled including Sysmon, PowerShell script block logging, module logging, ETW providers, and optional Microsoft Defender for Endpoint integration. These VMs use host-only or internal networking to prevent any accidental communication with production networks or the internet unless explicitly needed for specific tests.
- A forensic workstation virtual machine running a Linux distribution such as REMnux or Kali Linux equipped with memory forensics tools including Volatility 3, YARA, and supporting utilities for analyzing memory dumps, extracting artifacts, and scanning for indicators.
- A network monitoring component consisting of either a dedicated Zeek/Suricata virtual machine or packet capture running on the hypervisor's virtual switch to record all traffic within the lab network for later analysis with Wireshark, Zeek logs, or Suricata alerts.

- A telemetry collection and storage system that may range from a simple centralized log aggregation setup using the Elastic Stack or Wazuh to a more sophisticated SIEM deployment depending on your resources and objectives. All event data is time-synchronized using an internal NTP server within the lab network.
- An evidence storage area with separate directories for memory dumps, packet captures, log exports, extracted artifacts, YARA rules, Sigma rules, configuration files, and documentation. Evidence is organized by date and incident identifier to support repeatable analysis workflows.
- Synthetic attack simulation capabilities using benign emulation frameworks such as Atomic Red Team that generate realistic telemetry without deploying actual malware or connecting to external command-and-control infrastructure. These tools allow you to validate detections against known techniques in a controlled manner.

The lab architecture is designed to be modular so you can start small and expand as your needs grow. A minimal setup might consist of a single Windows VM, packet capture on the virtual switch, and Volatility 3 running on your host machine. A more complete setup adds dedicated forensic and monitoring VMs, centralized logging, and automated detection testing pipelines. Chapter 9 walks through building this environment step by step with specific commands, configurations, and validation procedures.

All examples in this book specify which components of the reference lab they use so you can follow along regardless of your current deployment level. When a demonstration requires a particular tool or configuration, the chapter explains how to set it up if you have not already done so.

How to Use This Book

This book is written for readers who want both conceptual understanding and practical capability. It is dense with technical detail because memory-resident threat detection requires a deep understanding of underlying mechanisms. Here are some guidelines for getting the most out of it:

Read sequentially through at least Chapters 1 through 5 before jumping into advanced topics. These chapters establish the Windows internals knowledge, PE structure understanding, execution surface awareness, telemetry infrastructure familiarity, and threat taxonomy that later chapters assume.

Skipping this foundation will make the memory forensics, EDR engineering, and detection methodology sections significantly harder to follow.

Use the reference lab actively rather than passively reading about concepts. When a chapter describes how to inspect process memory maps, actually do it on your test VM. When it explains how to configure PowerShell logging, enable it and verify that events appear in the Event Log. When it discusses YARA scanning of memory dumps, acquire a dump from your lab and run the commands yourself. Hands-on practice reinforces understanding far more effectively than reading alone.

Treat code examples as learning tools rather than copy-paste templates. Every example specifies its supported Windows versions, required privileges, dependencies, build commands, expected inputs and outputs, and safe usage boundaries. Read these carefully before running anything. Modify examples to match your environment and experiment with variations to deepen your understanding of the underlying mechanisms.

Focus on detection logic rather than specific tool configurations. While this book uses particular tools like Volatility 3, Sysmon, Zeek, Sigma, and others for concrete examples, the principles apply regardless of which products you deploy in production. Understanding what signals matter and why matters more than memorizing configuration syntax for any single tool.

Keep a working notes file as you read. Record key concepts, interesting detection ideas, configuration commands you want to remember, questions that arise, and connections between topics. This book covers substantial material across many chapters; maintaining your own reference notes will help you retain information and create a personalized quick reference.

Do not run any demonstration code or simulation tests on production systems unless you have explicit authorization and have carefully evaluated the risks. Even benign simulations can trigger security controls, generate alerts that consume SOC resources, or interact unexpectedly with other software running on the system. Always test in your isolated lab first.

Use this book as both a learning path and a desk reference. When you encounter a specific problem in your work such as investigating suspicious PowerShell activity, analyzing a memory dump from an incident, tuning detection rules to reduce false positives, or designing network monitoring for encrypted traffic, return to the relevant chapter for detailed guidance rather than trying to recall everything from memory.

The book concludes with a comprehensive references section that lists all sources cited throughout the text. Each reference includes full publication details and stable URLs formatted as markdown links so you can verify facts, explore topics in more depth, or check for updates to rapidly changing technical information.

Chapter 1: Windows Internals for Detection Engineers

Understanding how Windows manages processes, memory, modules, and security is not optional for effective detection of memory-resident threats. Every technique that operates primarily in RAM does so by exploiting documented operating system behaviors. Defenders who understand these behaviors can recognize when they are being abused because they know what normal looks like. This chapter establishes the foundational Windows concepts every detection engineer needs.

User Mode, Kernel Mode, and Detection Visibility

Windows processors operate in different privilege levels that determine what code can access. Applications run in user mode with restricted privileges and isolated virtual address spaces. The kernel runs in kernel mode with full system access and a shared address space for core components including the executive, device drivers, and hardware abstraction layer [3].

The separation matters for detection because visibility differs dramatically between modes. User-mode code cannot directly read kernel memory or inspect other processes without going through documented APIs that enforce access controls. Kernel-mode code can see everything: all processes, all threads, all memory regions, all loaded modules, all handles, all network connections. This asymmetry means that kernel-mode monitoring typically provides more complete and tamper-resistant telemetry than user-mode monitoring alone.

Modern endpoint detection and response products often combine both approaches. User-mode components monitor API calls, consume ETW events, inspect process command lines, and analyze script content. Kernel-mode drivers subscribe to notification callbacks for process creation, thread creation, image loading, file system access, registry operations, and network activity. The kernel driver sees events before user-mode hooks can intercept

or suppress them, making it harder for malware to hide its activity from the sensor [1].

However, kernel-mode monitoring carries risks. A bug in a kernel driver can crash the entire system. Microsoft has increasingly restricted kernel-mode capabilities through mechanisms like PatchGuard, which prevents modification of critical kernel structures, and HVCI (Hypervisor-Protected Code Integrity), which validates that only signed code executes in kernel mode. These protections benefit defenders by making it harder for malware to tamper with security software, but they also constrain what monitoring techniques are possible on modern Windows builds.

For detection engineering, the practical implication is layered visibility. No single monitoring approach sees everything reliably. User-mode telemetry may miss activity that bypasses API hooks. Kernel-mode telemetry may be incomplete for operations that occur entirely within protected processes or use undocumented mechanisms. Effective detection architectures combine multiple signals from both modes and cross-correlate them to compensate for individual gaps.

Processes, Threads, and Execution Contexts

A process in Windows is an execution container with its own virtual address space, handle table, security context, and set of loaded modules. When you run a program, the operating system creates a process structure that tracks all resources associated with that execution instance. Each process has a unique process identifier (PID), though PIDs are recycled over time so they should never be used as the sole correlation key in detection logic.

Threads are the actual units of execution within a process. Every process has at least one thread, and many have multiple threads running concurrently. Threads share the process's address space and handles but maintain their own stack, register state, and scheduling context. A thread identifier (TID) is unique only within its parent process.

For detection purposes, processes and threads matter because memory-resident techniques often involve anomalous process relationships or unexpected thread behavior:

- Suspicious parent-child relationships: An attacker may spawn PowerShell from an unusual parent process like Word or Excel to execute a down-

loaded payload. The parent-child relationship itself is a strong signal that something abnormal occurred.

- Unexpected thread creation: Legitimate processes typically create threads in predictable patterns. A stable application like explorer.exe suddenly creating a new thread that allocates executable memory and establishes network connections warrants investigation.
- Process impersonation: Techniques like process injection place code inside legitimate processes so their activity blends with normal behavior. Detection requires understanding which processes normally perform which operations.

Windows provides several APIs for process and thread management including CreateProcess, CreateRemoteThread, OpenProcess, and NtCreateUserThread. Monitoring calls to these APIs through ETW or kernel callbacks provides visibility into process creation patterns, cross-process interactions, and potential injection activity. Sysmon Event ID 1 captures detailed process creation information including command line arguments, parent process details, hashes, and integrity levels [2].

The concept of execution context extends beyond individual processes and threads to include security tokens, job objects, and session boundaries. A process's token determines what resources it can access. Job objects group processes for resource management and enforcement of constraints like maximum memory usage or prevention of child process creation. Session boundaries separate interactive desktop sessions from service contexts. All of these factors influence how memory-resident techniques behave and what artifacts they leave behind.

Virtual Memory, Address Spaces, and Paging

Windows uses virtual memory to provide each process with the illusion of having its own large, contiguous address space independent of physical RAM. On 64-bit Windows, each process has a theoretical address space ranging from 0x0000000000000000 through 0x000FFFFFFFFFFFFF for user mode and the upper portion reserved for kernel mode [7].

Virtual memory works by mapping virtual addresses to physical memory pages through page tables maintained by the memory management unit. When a process accesses a virtual address, the hardware translates it to a physical

address using these tables. If no physical page is currently mapped, a page fault occurs and the operating system either loads the required data from disk or allocates a new zeroed page.

For detection engineers, virtual memory concepts matter because memory-resident techniques manipulate address spaces in observable ways:

- Memory allocation patterns: Processes allocate memory using VirtualAlloc or NtAllocateVirtualMemory. Normal applications typically allocate pages with PAGE_READWRITE protection for data and rely on mapped executable sections from loaded modules for code. Allocating regions with PAGE_EXECUTE_READWRITE (RWX) protection is uncommon in legitimate software and often indicates injected code or shellcode [13].
- Memory mapping: Executable images are mapped into address space as file-backed sections with specific permissions for each section (.text is read-execute, .data is read-write, etc.). Private executable memory that does not correspond to any loaded module warrants scrutiny.
- Permission changes: VirtualProtect or NtProtectVirtualMemory can change page protection attributes. A sequence where a process allocates RW memory, writes data to it, then changes protection to RX or RWX is characteristic of many injection techniques.

The Virtual Address Descriptor (VAD) tree is the internal Windows data structure that tracks all committed and reserved regions in a process's address space. Each VAD node represents a contiguous range of virtual addresses with associated attributes including commit status, protection level, whether it is file-backed or private, and which file if mapped. Memory forensic analysis of VAD trees reveals the complete memory layout of a process at the time of acquisition, including regions that may not appear in standard API-based enumerations [4].

Understanding paging also matters for evidence preservation. When physical memory pressure increases, Windows moves inactive pages to the paging file on disk. This means some memory contents from a compromised system may persist on disk even after reboot, though encrypted paging files and overwriting make recovery unreliable. For live response investigations, acquiring memory before significant paging occurs provides the most complete picture.

Heaps, Stacks, and Data Layout

Within a process's address space, different regions serve different purposes. The heap is used for dynamic memory allocation through functions like HeapAlloc, malloc, or new. The stack holds local variables, function arguments, return addresses, and call chain information for each thread. Loaded modules occupy their own mapped regions. Environment blocks and process parameters have dedicated areas.

For detection and forensic analysis:

- Stacks contain call chains that reveal how code reached its current execution point. Examining a suspicious thread's stack can show whether it was created normally or through injection techniques like CreateRemoteThread or APC queueing. Stack walking is a standard technique in both debugging and incident response.
- Heaps may contain configuration data, network strings, command-and-control URLs, encoded payloads, or other artifacts that malware stores during execution. String extraction from heap memory is a common forensic technique for recovering indicators.
- Data layout patterns can be anomalous. A process whose heap contains executable code rather than just data structures may indicate heap spraying techniques used to support return-oriented programming attacks or other exploitation methods.

Windows provides multiple heaps per process: a default process heap and potentially additional heaps allocated by the application or loaded libraries. Each heap maintains its own metadata for tracking allocated blocks. Heap metadata corruption is sometimes used as an attack technique, though modern Windows versions include heap hardening features like randomized allocation patterns and guard pages that make exploitation more difficult.

Stack size defaults to one megabyte per thread on most Windows configurations, though this can be changed at thread creation time. Stack overflow conditions and stack-based buffer overflows were historically common exploitation vectors; modern compilers and operating system protections including stack canaries, DEP (Data Execution Prevention), and ASLR (Address Space Layout Randomization) significantly reduce their reliability.

PEB, TEB, and Process Metadata

The Process Environment Block (PEB) is a user-mode data structure that the operating system creates for each process. It contains information about the process including its loaded modules list, command-line arguments, environment variables, current directory, image base address, debugging state, session ID, and various flags [8]. The PEB resides in the process's own address space and can be accessed from user mode without kernel calls.

The Thread Environment Block (TEB) is a per-thread structure that contains thread-specific information including a pointer to the PEB, TLS (Thread Local Storage) slots, last error code, exception handling state, and stack boundaries [4]. Each thread's TEB is accessible via segment registers: GS on x64 systems and FS on x86 systems.

For detection engineers, PEB and TEB matter for several reasons:

- Module enumeration: The PEB contains a linked list of loaded modules that applications use to find DLL addresses at runtime. Comparing this list against what the loader reports or what appears in memory can reveal discrepancies indicative of manual mapping or reflective loading techniques.
- Debugging detection: The BeingDebugged flag in the PEB allows malware to detect whether it is running under a debugger and modify its behavior accordingly. Security tools that use debugging APIs may inadvertently set this flag, which sophisticated malware checks before executing payloads.
- Command-line visibility: The full command line stored in the PEB's process parameters may contain arguments that were logged during process creation but later cleared from memory by an attacker attempting to hide evidence.
- Environment manipulation: Attackers sometimes modify environment variables through the PEB to affect DLL search paths or application behavior without touching the registry.

The PEB structure is considered semi-documented because Microsoft does not guarantee its layout across versions, though it has remained relatively stable. Security tools and forensic frameworks read PEB structures directly from memory during analysis because they provide rich context about process state that complements kernel-mode information.

Handles, Objects, Tokens, and Access Rights

Windows uses handles as references to kernel objects. A handle is a numeric identifier within a process's handle table that maps to an actual kernel object such as a file, registry key, mutex, event, semaphore, process, thread, or token. When a process calls an API like CreateFile or OpenProcess, the kernel creates or opens the requested object and returns a handle that the process uses for subsequent operations [19].

Handles matter for detection because they reveal what resources a process has accessed:

- Cross-process handles: A process opening a handle to another process with PROCESS_VM_WRITE or PROCESS_VM_OPERATION rights is preparing to modify that process's memory, which is characteristic of injection techniques. Sysmon Event ID 10 logs process access events including the target process and requested access mask [8].
- LSASS access: Opening lsass.exe is necessary for credential dumping techniques like Mimikatz. Windows Security Event ID 4673 or kernel object auditing can log when processes obtain handles to sensitive system processes.
- Hidden objects: Some advanced techniques manipulate handle tables to hide objects from enumeration, though this typically requires kernel-mode code and is detectable through memory forensics.

Objects in Windows are managed by the Object Manager, a kernel component that maintains global and per-process object directories. Each object has a security descriptor that controls access through discretionary access control lists (DACLs) and system access control lists (SACLs). The Security Reference Monitor performs access checks whenever code attempts to use an object handle [10].

Access tokens accompany every process and thread and represent the security context under which execution occurs. A token contains the user's SID, group SIDs, privileges (the Se*Privilege rights like SeDebugPrivilege or SeImpersonatePrivilege), integrity level, and metadata about elevation state and logon type [4]. Tokens determine what objects a process can access and what operations it can perform.

Token-related detection signals include:

- Privilege escalation: A process gaining new privileges through token manipulation or impersonation may indicate lateral movement preparation.
- Token theft: Techniques that duplicate or steal tokens from high-privilege processes allow attackers to execute code under different security contexts.
- Anonymous tokens: Processes running with unexpected token types or missing expected group memberships warrant investigation.

Integrity Levels, Sessions, and Services

Mandatory Integrity Control (MIC) adds a second axis of access control beyond traditional user/group permissions. Every process, thread, and securable object has an integrity level: low, medium, high, or system [2]. MIC restricts lower-integrity processes from modifying higher-integrity objects regardless of discretionary permissions. This protects system files and critical processes from tampering by compromised user applications.

For detection, integrity levels provide context for evaluating suspicious behavior:

- Low-integrity anomalies: Processes running at low integrity (typically Internet Explorer in protected mode or AppContainer sandboxed applications) should not be able to modify most system resources. A low-integrity process somehow accessing high-integrity objects indicates either a configuration error or exploitation.
- Unexpected elevation: A standard user process suddenly running at high or system integrity without documented justification may indicate privilege escalation.
- Integrity mismatches: Child processes normally inherit their parent's integrity level. Significant deviations from this pattern warrant examination.

Windows sessions separate interactive desktop environments from service contexts. Session 0 traditionally hosted services and was isolated from interactive user sessions starting with Windows Vista to reduce the attack surface of system services. Remote Desktop connections create additional sessions numbered sequentially. Session boundaries affect which processes

can interact through GUI mechanisms and influence how some persistence techniques operate.

Windows services are long-running processes that start automatically or on demand, typically running under dedicated service accounts rather than interactive user contexts. Services run in Session 0 and have different security constraints than interactive processes. Many memory-resident techniques target services because they run with elevated privileges, persist across reboots, and often have broader network access than user applications.

Monitoring service-related activity through Event Logs (Service Control Manager events), Sysmon configuration for service installation detection, and registry monitoring for service key modifications provides visibility into persistence attempts that abuse the service infrastructure.

The Registry and Configuration State

The Windows Registry is a hierarchical database that stores system and application configuration settings, security policies, device parameters, user preferences, and much more. It consists of hives: SYSTEM, SOFTWARE, SAM, SECURITY, and NTUSER.DAT files for each user profile. Each hive contains keys organized in a tree structure with values that hold actual data [19].

For detection engineers, the registry matters because:

- Persistence mechanisms: Attackers frequently use registry autorun keys (Run, RunOnce under CurrentVersion), service keys, and other locations to ensure their code executes on system startup. Sysmon Event IDs 12, 13, and 14 log registry object creation, value setting, and renaming [8].
- Configuration manipulation: Modifying security policies, disabling logging, changing proxy settings, or adjusting PowerShell execution policies all occur through registry modifications.
- Lateral movement artifacts: Some lateral movement techniques leave evidence in registry keys related to scheduled tasks, COM hijacking entries, or DLL search path modifications.
- Forensic timeline: Registry hives maintain transaction logs and USN journal references that support timeline reconstruction during investigations.

Key registry locations relevant to memory-resident threat detection include:

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run and RunOnce for user-level persistence
- HKLM\SYSTEM\CurrentControlSet\Services for service-based persistence
- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\taskcache for scheduled task configuration
- HKLM\SOFTWARE\Policies\Microsoft\Windows\System for security policy settings including DLL search order and UAC configuration
- HKCU\Software\Classes\CLSID and related keys for COM hijacking opportunities

Registry monitoring through Sysmon, native Windows auditing with SACL configuration, or ETW providers provides real-time visibility into configuration changes that may indicate active compromise. However, the high volume of legitimate registry activity requires careful filtering to avoid alert fatigue while maintaining coverage for suspicious modifications.

System Calls and API Layers: A Defensive View

Windows applications interact with the operating system through layered APIs. The Win32 API (implemented in kernel32.dll, user32.dll, advapi32.dll, and others) provides high-level functions that applications call for file operations, process management, networking, security, and more. These functions ultimately invoke the Native API (implemented in ntdll.dll), which contains system service stubs that transition from user mode to kernel mode [4].

The system call mechanism works as follows: when an application calls a function like CreateFile, the Win32 API implementation may perform some validation and parameter preparation, then call the corresponding NtCreateFile in ntdll.dll. The NtCreateFile stub loads the system service number into a register and executes a syscall instruction that transfers control to the kernel dispatcher. The dispatcher looks up the service routine in the System Service Descriptor Table (SSDT) and executes it in kernel mode [1].

For detection engineering, understanding API layers matters because:

- Monitoring points: Different monitoring approaches hook at different layers. User-mode API hooks intercept calls in Win32 DLLs. ETW providers emit events from within system components as operations

occur. Kernel callbacks trigger when the executive processes requests. Each layer provides different visibility and has different evasion risks.
- Direct syscalls: Some advanced techniques bypass monitored Win32 APIs by invoking Native API functions or syscalls directly, potentially evading user-mode hooks that only monitor higher-level functions. Defenders using kernel-mode monitoring are less affected by this because the kernel sees all system service invocations regardless of how they were initiated.
- API patterns: Legitimate applications use APIs in predictable sequences. A process calling OpenProcess with PROCESS_ALL_ACCESS, then VirtualAllocEx, then WriteProcessMemory, then CreateRemoteThread follows a pattern characteristic of classic DLL injection [5]. Behavioral detection looks for these sequences rather than flagging individual API calls in isolation.

Windows provides extensive telemetry about system call activity through ETW providers like Microsoft-Windows-Kernel-Process and Microsoft-Windows-Kernel-General. These providers emit events for process creation, thread creation, memory allocation, file operations, and other activities that reflect underlying system call patterns. Configuring these providers appropriately requires administrative privileges and understanding of the performance implications of high-volume event generation.

The defensive takeaway is that no single monitoring layer provides complete visibility. User-mode hooks can be bypassed. ETW events can be disabled or tampered with by attackers with sufficient privileges. Kernel callbacks can be unregistered by rootkits operating at kernel level. Effective detection architectures use multiple layers and cross-correlate signals to compensate for gaps in any individual approach.

Chapter Summary

This chapter established the Windows internals foundation necessary for understanding memory-resident threat detection. Key concepts include:

- User mode versus kernel mode determines visibility and tamper resistance of monitoring approaches.
- Processes, threads, and their relationships provide strong detection signals when anomalous patterns appear.

- Virtual memory management, VAD trees, and page protection attributes reveal how processes use address space and whether suspicious allocations occurred.
- Heaps, stacks, and data layout contain forensic artifacts including call chains, configuration strings, and injected code.
- PEB and TEB structures provide rich process and thread metadata useful for analysis and discrepancy detection.
- Handles, objects, tokens, and access rights determine what resources processes can use and reveal cross-process interactions characteristic of injection techniques.
- Integrity levels add a second security axis that helps evaluate whether process behavior matches expected privilege levels.
- The Registry stores configuration state and is frequently abused for persistence mechanisms.
- System call layers and API patterns provide multiple monitoring points, each with different strengths and evasion risks.

The next chapter examines PE/COFF structures in detail because understanding how Windows represents executable images on disk and in memory is essential for recognizing when that representation has been manipulated by an attacker.

Chapter 2: PE/COFF Structures and Executable Image Behavior

Windows executables, dynamic-link libraries, and device drivers all use the Portable Executable (PE) format, which is based on the Common Object File Format (COFF). Understanding PE structures matters for detection because memory-resident techniques often manipulate how images are loaded, mapped, or represented in memory. Defenders who can distinguish normal from anomalous image states across filesystem, loader metadata, and actual memory contents gain visibility into techniques that would otherwise be invisible to file-centric monitoring alone.

PE File Format: Headers, Sections, and Metadata

A PE file consists of several structural elements arranged sequentially [1]:

- MS-DOS Header (IMAGE_DOS_HEADER): A 64-byte header starting with the MZ signature (0x5A4D) for backward compatibility with MS-DOS. It contains an e_lfanew field that points to the start of the PE header.
- DOS Stub: Optional real-mode code that typically displays "This program cannot be run in DOS mode" when executed on MS-DOS. Many PE files include this; some packers or custom loaders omit or modify it.
- PE Signature: Four bytes containing "PE\0\0" (0x00004550) that identifies the file as a valid PE image.
- COFF File Header (IMAGE_FILE_HEADER): Specifies machine type (0x8664 for x64, 0x14C for i386), number of sections (limited to 96 by the Windows loader), time/date stamp, pointer to symbol table, number of symbols, size of optional header, and characteristics flags indicating whether the file is executable, a DLL, or has specific properties like being large-address-aware.
- Optional Header (IMAGE_OPTIONAL_HEADER): Despite its name, this header is required for PE image files (though not for object files). It uses magic numbers to identify PE32 (0x10B) for 32-bit or PE32+ (0x20B)

for 64-bit formats. Key fields include entry point address, base of code and data, image base for loading, section alignment and file alignment values, subsystem specification (GUI, console, native), DLL characteristics flags, size of image in memory, size of headers, and a series of data directory entries pointing to imports, exports, resources, relocations, debug information, TLS callbacks, load configuration, bound imports, delay-load descriptors, and COM descriptors.

- Section Headers: An array of 40-byte IMAGE_SECTION_HEADER structures, one per section. Each header contains the section name (typically .text for code, .data for initialized data, .rdata for read-only data, .rsrc for resources, .reloc for relocations, .tls for thread-local storage, .idata for imports, .edata for exports), virtual size, virtual address within the loaded image, size of raw data on disk, file pointer to raw data, and characteristics flags indicating whether the section is readable, writable, executable, contains code, or contains initialized data.
- Section Data: The actual content of each section aligned according to file alignment rules. When loaded into memory, sections are mapped at their virtual addresses with alignment determined by the section alignment value in the optional header.

For detection engineers, PE structure knowledge enables several analysis capabilities:

- Header validation: Corrupted or malformed headers may indicate packing, obfuscation, or manual construction of malicious images. Unusual characteristics flags like an executable file claiming to be both a DLL and an EXE warrant scrutiny.
- Section anomalies: Sections with suspicious permission combinations (both writable and executable), unusually large sizes relative to their content type, non-standard names, or section data that does not match expected characteristics may indicate packed payloads or embedded malicious code.
- Entry point analysis: The entry point specified in the optional header should fall within a code section (.text typically). An entry point in an unusual location or outside any section is suspicious and may indicate a custom loader or packed executable.

Imports, Exports, Relocations, and TLS Callbacks

Data directories in the PE optional header reference several critical tables that define how an image interacts with the system and other modules:

Import Table: Specifies which functions from which DLLs the image requires at load time. The table lists each imported module by name and provides function names or ordinal numbers for each required import [1]. For detection, import analysis reveals what capabilities an executable intends to use. A simple utility program importing VirtualAllocEx, WriteProcessMemory, CreateRemoteThread, and OpenProcess suggests process injection capability even without examining the code itself. Missing imports that should be present (like standard Win32 API functions in a GUI application) may indicate packing or manual resolution techniques.

Export Table: Present primarily in DLLs, this table lists functions that the module makes available to other programs. Legitimate system DLLs export many well-known functions. A suspicious DLL with no exports or only oddly named exports may be designed for injection rather than normal linking. Some malware modules export nothing because they are loaded reflectively and never registered with the loader.

Relocation Table: Contains fixups that the loader applies when an image cannot be loaded at its preferred base address due to conflicts. Each relocation entry specifies where in the image a pointer or address needs adjustment by the difference between preferred and actual load addresses [1]. Missing relocation tables force the loader to use ASLR-compatible alternatives or fail to load, which can affect whether DLL shadowing or side-loading techniques succeed.

TLS Callbacks: Thread Local Storage callbacks are functions specified in the TLS directory that the loader invokes before the entry point during process attachment and thread attachment events. These callbacks execute with full privileges of the hosting process before any normal initialization code runs [1]. For detection, TLS callbacks matter because they can be abused to execute code early in the loading process, potentially before security hooks are established or before AMSI scanning occurs. Unusual numbers of TLS callbacks or callbacks that perform significant operations rather than simple thread-local initialization warrant examination.

The Windows Loader: How Images Become Processes

When you create a process using CreateProcess or a similar API, the Windows loader performs a sequence of well-defined steps to transform a PE file on disk into an executing process in memory:

1. The loader reads the DOS header and verifies the MZ signature, then follows e_lfanew to locate the PE signature.
2. It validates the COFF header and optional header, checking that the machine type matches the system architecture and that required characteristics are present.
3. It reserves a region of virtual address space at the image's preferred base address (from the optional header) or an alternative location if the preferred address is occupied.
4. It maps each section from the file into the reserved address space with permissions matching the section characteristics (.text mapped as read-execute, .data as read-write, etc.).
5. It processes the import table by loading required DLLs recursively and resolving imported function addresses through their export tables or API set mappings.
6. It applies relocations if the image loaded at a different base address than preferred.
7. It initializes TLS structures and calls any registered TLS callbacks.
8. It sets up the PEB including the loader data structure with the linked list of loaded modules, process parameters with command-line arguments and environment variables, and other metadata.
9. It creates the initial thread and transfers execution to the entry point specified in the optional header.

For detection engineers, understanding this sequence matters because each step produces observable artifacts:

- File system access events as the loader reads the PE file and any dependent DLLs.
- Image load events logged by Sysmon Event ID 7 or ETW providers for each module mapped into the process [8].
- Registry access as the loader consults KnownDLLs, API set mappings, and side-by-side assembly information.

- Process creation events with command-line arguments, parent process information, and hash values captured by Sysmon Event ID 1 or Security Event ID 4688.

When an attacker bypasses the normal loader through techniques like reflective loading or manual mapping, several of these artifacts are missing or anomalous. A DLL that exists in memory but has no corresponding image load event, does not appear in the PEB's loaded module list, and maps to private rather than file-backed memory is a strong indicator of non-standard loading behavior [2].

Dynamic Linking and DLL Search Paths

Windows uses dynamic linking extensively: most executable images import functions from shared DLLs rather than including all code statically. When the loader needs to locate a DLL referenced in an import table, it searches specific directories in a defined order [2]:

For unpackaged applications with safe DLL search mode enabled (the default on modern Windows):

1. Directories listed in KnownDLLs registry key
2. The directory containing the executable image
3. The system directory (%SystemRoot%\system32)
4. The 16-bit system directory
5. The Windows directory (%SystemRoot%)
6. The current working directory
7. Directories listed in the PATH environment variable

Applications can modify this search order using SetDllDirectory, LoadLibraryEx with specific flags, or manifest-based redirection. Applications can also use LOAD_LIBRARY_SEARCH_SYSTEM32 to restrict searches to system directories only, mitigating side-loading risks.

For detection engineers, DLL search path behavior matters because:

- Side-loading attacks exploit the search order by placing a malicious DLL with the same name as a legitimate dependency in a directory that is

searched before the real DLL's location. When the victim application loads, it imports from the malicious DLL instead [3]. The malicious code executes under the identity of a legitimate, potentially signed application.

- Detection signals include: ImageLoad events showing DLLs loaded from unexpected paths, discrepancies between expected and actual DLL provenance, unsigned DLLs loaded into processes that typically use only signed modules, or DLLs whose hashes do not match known-good baselines for that filename [13].
- Legitimate software sometimes ships with private copies of DLLs in application directories. Distinguishing between intentional private DLL deployment and adversarial side-loading requires understanding the specific application's normal behavior and maintaining baseline knowledge of expected module locations.

The LOAD_LIBRARY_SEARCH_* flags introduced in later Windows versions provide safer alternatives to LoadLibrary for applications that need explicit control over DLL resolution. Security recommendations encourage developers to use fully qualified paths when loading DLLs dynamically and to enable safe search mode consistently, but legacy applications and poorly coded software continue to create side-loading opportunities.

Signed Versus Unsigned Modules and Authenticode

Authenticode is Microsoft's code signing framework for Windows executables and drivers. A signed PE file includes a digital signature stored in the Certificate Table data directory that cryptographically binds the publisher's identity to the file contents [1]. The signature covers all sections up to the last section's end, excluding the certificate table itself and checksum fields to allow signatures to be added or updated without invalidating existing signatures.

For detection engineers, code signing status provides important context:

- Signed modules from trusted publishers are more likely to be legitimate, though attackers can abuse legitimately signed binaries through living-off-the-land techniques or side-loading attacks that use the victim application's signature for cover.
- Unsigned modules loaded into processes that normally use only signed components warrant investigation. System services loading unsigned DLLs from unexpected locations is particularly suspicious.

- Invalid or expired signatures may indicate tampering, though they can also result from legitimate certificate management issues or test-signed software in development environments.
- Signature spoofing through techniques like signature stamping (copying a valid signature from one file to another) produces cryptographic mismatches that verification tools detect, but casual inspection of signature presence alone may be fooled.

Sysmon Event ID 7 includes Signature and SignatureStatus fields for loaded images when configured appropriately [8]. ETW providers like Microsoft-Windows-CodeIntegrity log code integrity policy enforcement events including unsigned driver loads and integrity violations. These telemetry sources support detection rules that flag unexpected unsigned module loading or signature verification failures.

Authenticode validation occurs at multiple points: during driver loading (where enforced signing is mandatory on modern Windows), during application installation through SmartScreen and application control policies, and optionally during execution when AppLocker, WDAC (Windows Defender Application Control), or similar solutions are deployed. Understanding where and how signature validation occurs helps defenders identify gaps where unsigned malicious code might execute without triggering policy enforcement.

Memory-Mapped Images and Their Normal Appearance

When a PE file is loaded by the Windows loader, it becomes a memory-mapped image with specific characteristics that distinguish it from private or anonymous memory regions:

- File-backed VAD entries: Each section appears in the process's VAD tree as a mapped view of the underlying file. The VAD node references the file object and specifies the mapping offset [4].
- Expected protection attributes: Section permissions match their characteristics flags. Code sections (.text) are mapped with PAGE_EXECUTE_READ or PAGE_EXECUTE_READWRITE during loading then typically changed to PAGE_EXECUTE_READ after relocation processing. Data sections (.data) are PAGE_READWRITE. Read-only data (.rdata) is PAGE_READONLY.

- Loader registration: The module appears in the PEB's Ldr->InLoadOrderModuleList, Ldr->InMemoryOrderModuleList, and Ldr->InInitializationOrderModuleList linked lists [15]. These three lists should be consistent for normally loaded modules.
- Path information: Full path to the source file is stored in the LDR_DATA_TABLE_ENTRY structure associated with the module.
- Consistent hashes: Computing a hash of the memory-mapped image and comparing it to the hash of the file on disk should produce matching results (barring legitimate modifications like patching or update mechanisms).

For detection, discrepancies from this normal appearance indicate potential manipulation:

- Private executable memory without corresponding file backing suggests injected code or shellcode rather than a legitimately loaded module.
- Modules appearing in only one of the three loader lists or missing entirely from loader metadata may have been manually mapped to avoid detection [2].
- Protection attributes that differ from section characteristics (like a .text section mapped as writable when it should be read-only after initialization) indicate potential patching or code modification.
- Hash mismatches between memory and disk versions of the same module suggest in-memory tampering, though legitimate patching mechanisms and some update frameworks also produce this pattern.

Memory forensic analysis tools like Volatility 3's windows.pslist, windows.dlllist, and windows.vadinfo plugins enumerate these structures from acquired memory images to identify anomalies that may indicate memory-resident threats operating outside normal loader behavior [4].

Side-Loading Indicators and Module Provenance Anomalies

DLL side-loading is a technique where an attacker places a malicious DLL alongside a legitimate executable, exploiting the DLL search order so that when the legitimate application loads its dependencies, it imports from the malicious

DLL instead of the expected one [3]. The malicious code executes under the identity of the legitimate application, potentially inheriting its trust level and evading detection mechanisms that rely on process name or signature alone.

Detection of side-loading requires understanding normal module provenance for each application:

- Path analysis: Each DLL should load from an expected location. System DLLs should come from %SystemRoot%\system32 or KnownDLLs locations. Application-specific DLLs should come from the application's installation directory or documented subdirectories. A DLL named "msvcp140.dll" loading from a user's Downloads folder rather than the Visual C++ redistributable installation path is suspicious [17].
- Signature correlation: If an application normally loads signed system DLLs, an unsigned module with the same filename loaded from an unexpected path indicates potential side-loading. Sysmon configuration that logs SignatureStatus for image load events supports this detection [8].
- Hash baselining: Maintaining known-good hashes for frequently loaded modules allows detection of replacements or substitutions. A legitimate DLL hash that suddenly differs across multiple systems may indicate a supply chain compromise or targeted side-loading deployment.
- Dependency analysis: Analyzing which modules each application actually imports versus which modules appear loaded can reveal discrepancies. If an application loads a DLL it does not import and has no documented dependency on, investigate why that module is present.
- Timing correlation: Side-loading often occurs during initial execution of the victim application. Correlating image load events with process creation events helps identify whether suspicious modules loaded as part of normal startup or were injected later.

Legitimate software deployment practices complicate side-loading detection. Some applications ship with private copies of commonly used DLLs to avoid version conflicts. Software updates may replace DLLs with new versions that have different hashes. Virtualization and sandboxing solutions inject their own monitoring DLLs into processes, creating expected anomalies that must be accounted for in detection rules. Effective detection requires maintaining accurate baselines for your specific environment rather than relying on generic assumptions about what is normal.

Chapter Summary

This chapter covered PE/COFF structures and executable image behavior at a level sufficient for recognizing when images have been manipulated or loaded through non-standard mechanisms. Key concepts include:

- PE file format consists of DOS header, PE signature, COFF headers, optional headers with data directories, section headers, and section data.
- Import tables reveal intended capabilities; unusual imports suggest injection or exploitation functionality.
- The Windows loader follows a defined sequence producing observable artifacts at each step; missing artifacts indicate non-standard loading.
- DLL search order determines where modules are found; side-loading exploits this by placing malicious DLLs in prioritized locations.
- Authenticode signatures provide trust indicators but can be abused through living-off-the-land techniques or spoofing attempts.
- Memory-mapped images have specific characteristics including file-backed VAD entries, expected protection attributes, loader registration, and path information; deviations indicate potential manipulation.
- Side-loading detection requires understanding normal module provenance for each application and monitoring for path anomalies, signature mismatches, hash discrepancies, and unexpected dependencies.

The next chapter examines managed execution environments, scripting frameworks, and automation surfaces that attackers frequently abuse for memory-resident techniques, focusing on observability, telemetry generation, and detection opportunities for each surface.

Chapter 3: Managed Execution, Scripting, and Automation Surfaces

The CLR, Managed Code, and Runtime Observability

PowerShell Execution: Policies, Logging, and Telemetry

Scripting Hosts: cscript, wscript, and Suspicious Invocation

WMI Architecture, Event Subscriptions, and Abuse Indicators

COM Objects, Activation Patterns, and Detection Signals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

RPC, Named Pipes, and Inter-Process Communication Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 4: Windows Telemetry Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Event Tracing for Windows (ETW): Providers, Sessions, and Events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

The Windows Event Log: Channels, Sources, and Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

PowerShell Script Block Logging and Module Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

AMSI: Architecture, Integration Points, and Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Microsoft Defender Antivirus Telemetry and Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Sysmon-Style Extended Telemetry and Custom Providers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Telemetry Gaps, Performance Costs, and Baseline Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 5: The Memory-Resident Threat Spectrum

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

A Working Taxonomy: From Scripted Attacks to Pure Memory Implants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Living-off-the-Land: Abuse of Legitimate Binaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Suspicious Process Creation and Parent-Child Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Anomalous Command Interpreters and Script Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Unexpected Module Loading and DLL Search Path Manipulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

In-Memory PE Artifacts and Reflective Loading Indicators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Process Injection Patterns: What Defenders Can Observe

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Hollowing-Like Anomalies and Image Replacement Signals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Module Stomping, Token Misuse, and Persistence Telemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 6: Memory Forensics on Windows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Memory Acquisition Principles and Evidence Integrity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Virtual Versus Physical Addresses: What Matters to Defenders

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Process Enumeration and Hidden/Anomalous Process Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Loaded Modules, Unlinked DLLs, and Image Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

VAD Trees and Memory Map Inspection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Executable Memory Regions and Permission Anomalies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Thread Stacks, Handles, and Inter-Process Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Strings, Configuration Data, and Embedded Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Network Artifacts in Memory and Connection State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

YARA Scanning, IOC Hunting, and Behavioral Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Timeline Reconstruction and Process Tree Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 7: EDR Engineering and Detection Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Endpoint Telemetry Architecture: Sensors and Data Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Kernel Versus User-Mode Visibility and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Event Normalization, Enrichment, and Entity Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Process Ancestry Graphs and Stateful Correlation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Behavioral Analytics: From IOCs to Attack Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Rule Engines, Sigma/YARA Concepts, and Detection Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Alert Scoring, Deduplication, Suppression, and Baselining

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Telemetry Integrity, Tamper Awareness, and Sensor Health

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Performance Overhead, Privacy, and Retention Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 8: Network Detection for Memory-Resident Threats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Windows Networking Fundamentals for Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Socket-to-Process Correlation and Connection Telemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

DNS Patterns: DGA, Tunneling, and Beacon Indicators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

HTTP/HTTPS Metadata: Headers, Hostnames, and Timing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

TLS Visible Artifacts: SNI, JA3, and Certificate Signals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

SMB, RDP, and Enterprise Protocol Anomalies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Beacon Detection: Periodicity, Jitter, and Statistical Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Zeek and Suricata Telemetry for Memory-Resident Threats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Endpoint-Network Correlation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 9: Building a Defensive Research Laboratory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Virtualization Choices and Isolation Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Network Segmentation and Controlled Connectivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Test Endpoints: Windows Versions, Configurations, and Telemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Forensic Workstations and Analysis Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Packet Capture Infrastructure and Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Synthetic Attack Telemetry and Benign Emulation Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Time Synchronization, Logging, and Evidence Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Snapshots, Reset Procedures, and Containment Safeguards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 10: Detection Engineering Methodology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Threat-Informed Detection and Hypothesis Formulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Observable Selection and Telemetry Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Behavioral Abstractions and ATT&CK Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Detection-as-Code: Versioning, Review, and Reproducibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Unit Testing, Integration Testing, and Replay Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Synthetic Telemetry Generation for Rule Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Precision, Recall, False Positive Budgeting, and Alert Fatigue

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Severity Scoring, Confidence Levels, and Triage Guidance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Coverage Measurement and Detection Gap Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 11: Reference Implementation — A Mini EDR Lab

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Repository Structure and Build Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Telemetry Collector: ETW Consumer and Log Normalizer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

PE Parser and Section Analyzer Utility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Memory Map Inspector for Authorized Processes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Event Correlation Engine and Stateful Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Rule Engine with Sigma-Style Pattern Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

IOC Scanner and YARA Integration Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Investigation Console and Evidence Export

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Chapter 12: Forensic Case Studies and Investigative Workflows

Case Study 1: Suspicious PowerShell Activity and Outbound Beaconing

Case Study 2: Anomalous DLL Loading and In-Memory PE Artifacts

Case Study 3: Unexpected Executable Memory and Process Injection Signals

Case Study 4: Correlated Endpoint and Network Indicators of Persistence

From Alert to Conclusion: A Repeatable Investigation Workflow

Chapter Summary

Chapter 13: Operations, Production Deployment, and Defense Architecture Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Deploying Detection Infrastructure into Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Security Hardening Recommendations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Incident Response Procedures for Memory-Resident Threats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

Coverage Validation Methodology

Architectural Limitations and Adversarial Trade-offs

Future Directions for Memory-Resident Threat Defense

Chapter Summary

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/memory-residentmalwareandfilelessattackdetection.