Introduction to EDR Underpinnings

Overview

EDR software has become the norm for organizations looking to strike a balance between effective detection and remediation. Unlike the more granular and finely tuned solutions (sysmon with a true SIEM for example), EDRs are relatively undocumented at the level required to detect the never-ending onslaught of evasion techniques that pop-up in private chats among researchers and threat actors alike. The truth is that most of the most successful methods are just that, for the simple reason they are undocumented. EDR vendors operate with the same level of discretion to avoid rampant bypassing of their tooling. The goal of this document is not to unveil the most coveted and ground-breaking techniques employed by EDR developers for detection, but to serve as a collection of concepts that are difficult to find and compile on their own. Much of what is covered here will be done with brevity with the purpose of excluding details irrelevant to the pursuit of APEX detection creation.

Section 1: Assembly and CPU Primer

Understanding assembly in-depth is not necessary for the purpose of this document. Having context and reference when necessary is vital however, to visualizing how some EDR detection capabilities and malware functionalities play out in real time.

Abstraction Layers

Code can be viewed in three simple layers of abstraction:

High-Level Language

Programming languages vary in how they are interpreted versus compiled. Python, Java, and C# are examples of interpreted languages that would sit one abstraction layer higher than compiled languages. C will be used as the example of mid-level (hybrid of high and low level granularity) human readable language throughout.

Example:

int c;

printf(“Hello.\n”);

exit(0);

Machine Code

The C code above is translated by the compiler into code that can be understood by the CPU, hence the name ‘Machine Code.’ These are hexadecimal digits known as opcodes. The code below on its own will not produce the intended result of printing “Hello.” Registers and their stored values are being used here to move previously defined data around.

Example:

55

88 EC

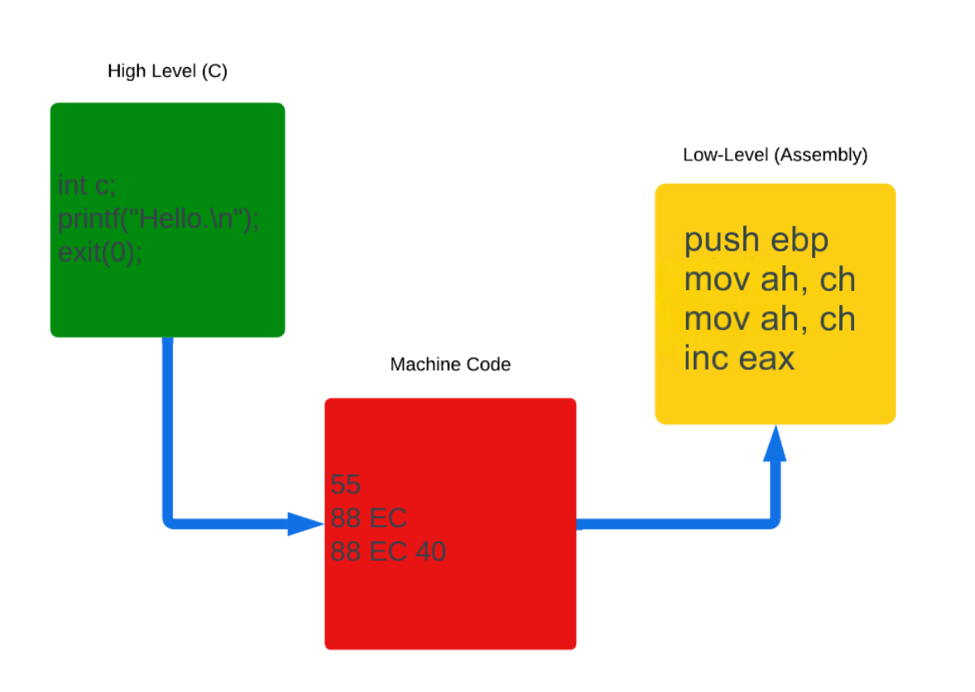
88 EC 40

Low-Level Language

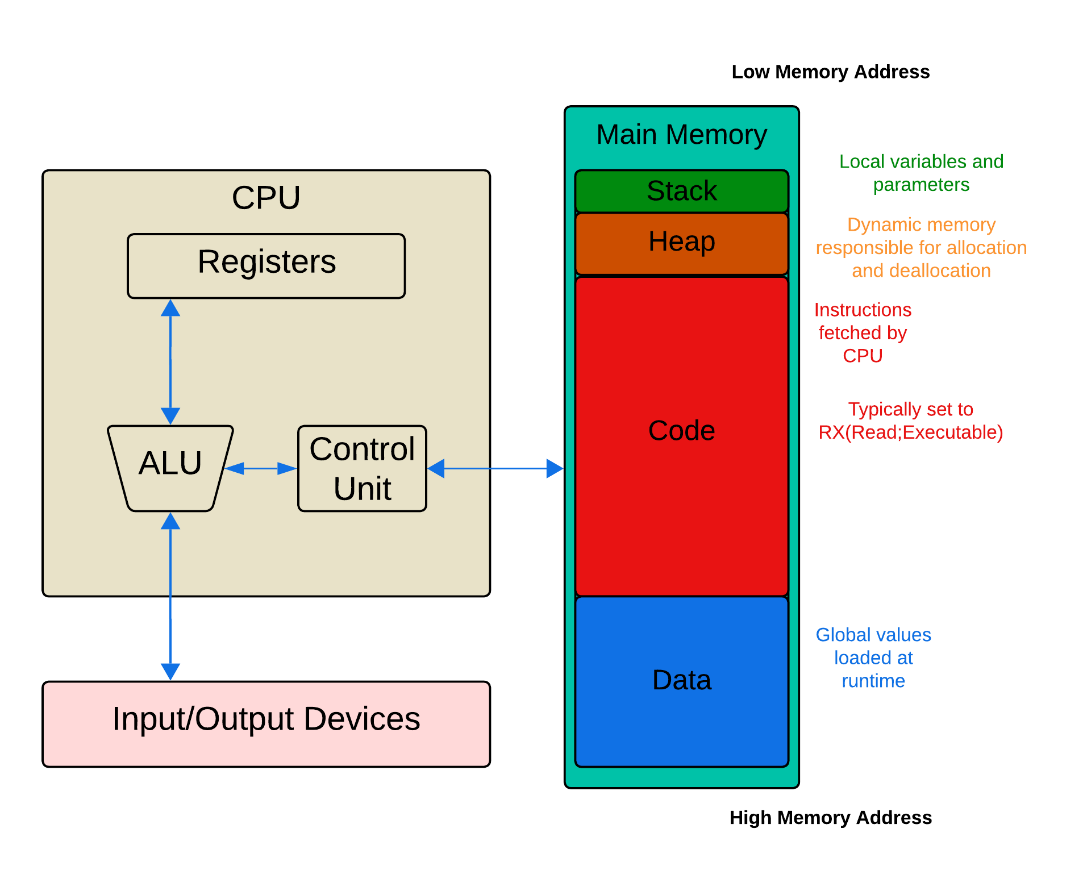
Assembly is the lowest level human readable language. This is the translated machine code you would find in a disassembler. The above opcodes are presented below in assembly.

Example:

push ebp  
mov ah, ch  
mov ah, ch  
inc eax

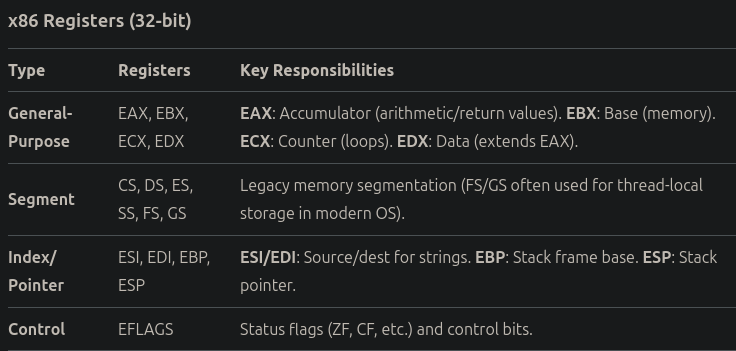


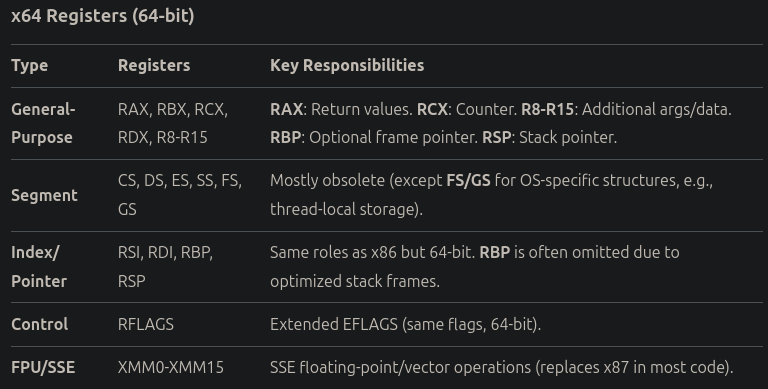
Despite x64 processors being the standard in 2025, due to the differences in how x86 registers function as well as x86 systems still being used (although numbers are dwindling), malware developers will often choose to deliver a binary based on its architecture. Below is a basic overview of the CPU, process memory, and the interoperative workflow between the two.



Registers

Registers are simple: they hold values. OP codes instruct the processor what registers to use and what should be done to them. X86 and x64 architectures differ, and it understanding some of the quirks is important if you are unlucky enough to be burdened with needing to look at a debugger/dissasembler.





Key considerations

#### x86

* Stack Args: Function arguments pushed right-to-left (cdecl/stdcall).
* Return Values: 32-bit in EAX, 64-bit in EDX:EAX.
* Floating-Point: Uses x87 stack (ST0-ST7) unless SSE is explicitly enabled.

#### x64

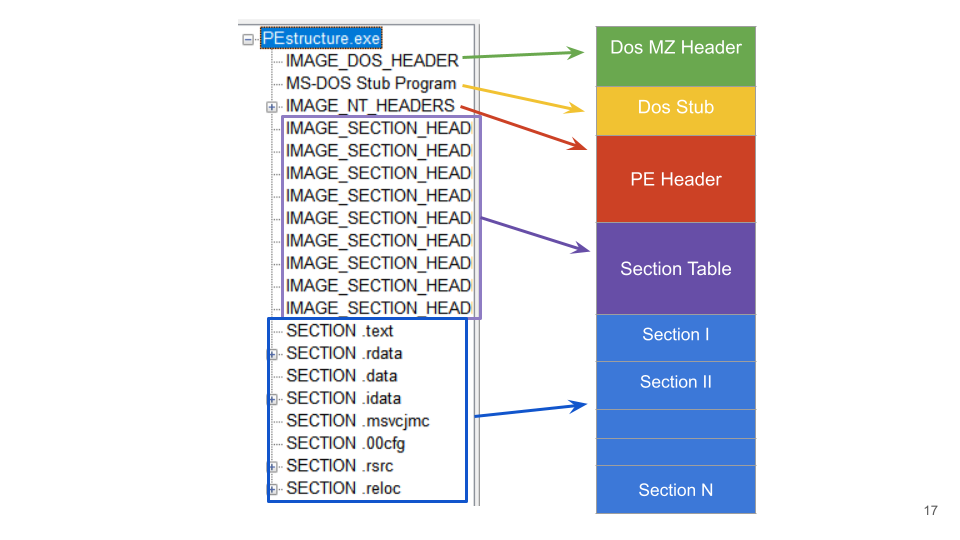
* Register Args: First 4 args in RCX, RDX, R8, R9 (integers), XMM0-XMM3 (floats).
* Stack Alignment: RSP must be 16-byte aligned before call.
* Red Zone: 128 bytes below RSP are safe from interrupts.
* Floating-Point: XMM0-XMM1 hold return values (SSE2 is mandatory in x64).

#### Universal Rules

* Zero-Extension: Writing to 32-bit regs (e.g., EAX) in x64 zero-extends to RAX.
* RIP-Relative Addressing: Common in x64 (e.g., mov rax, [rip+0x1234]).

Portable Executable Format

Windows NT uses the Portable Executable (PE) format for its compiled executables. The .exe and .dll extensions are functionally identical, although in normal usage, a .dll will be utilized by another executable. The property that defines this difference is IMAGE\_FILE\_DLL ( 0 or 1 as true/false values). Dynamic Link Libraries (DLL) cannot be run on their own, and require loading via a secondary process. This is abused sometimes by using a relatively bare bones loader that calls to and loads a DLL import which is the second stage and true malicious binary. Below is a diagram of the PE format and its sections/data structures:



There are many steps required to take an executable image (PE format is the focus here) from disk and run it within memory, however the vital API calls (and of most interest) are as follows:

1. VirtualAlloc()
   * Allocates memory for process space with permissions
     + RX for most applications; RWX is a potential and classic indicator of something awry
2. CreateFileW()
   * Opens handle to the file for reference
   * OR CreateProcessA()/CreateProcessW()
3. GetFileSize()
   * Needed for proper memory allocation
4. ReadFile()
   * Reads contents of file into allocated memory
5. AddressOfEntryPoint()
   * Entry point is where the executable code begins
6. Relocation
   * ASLR is a security feature enabled by default on modern versions of Windows NT that forces a randomized layout of process memory for a loaded image
     + If disabled, the preferred (hardcoded) ImageBase (location in memory to load) is defaulted to
   * This is a vital concept to keep in mind when inspecting binaries and translating what is seen from a disassembler (static) to a debugger (dynamic), in which addressing will not line up with what is seen in either tool
   * VirtualAlloc > Image\_Base\_Relocation
   * OR LdrpProcessRelocations
7. memcpy()
   * OR manually done with ReadProcessMemory > WriteProcessMemory
   * OR LdrpMapViewOfDllSection
   * This is known as section ‘mapping’
8. Dll Resolution
   * LdrLoadDll
   * LoadLibraryA()/LoadLibraryW()
   * GetProcAddress()
     + Gets addresses of functions
     + Used to patch Import Address Table(IAT)
       - This must be done as ASLR will not allow for 1:1 continuity from the Image’s IAT as defined in its PE data structure
9. VirtualProtect()/VirtualProtect(EX)
10. Sets permissions for sections (Read/ReadWrite/ReadWriteExecutable(R/RW/RWX))
11. NtCreateThreadEx
    * + Thread is created within process
12. NTResumeThread
    * + Thread is executed and jumps to entry point
13. RtlUserThreadStart
    * + Jump to user code

Section 2: Windows EDR Support

Anti-Malware Scan Interface (AMSI)

AMSI is a module that works with AV/EDR software to provide in-memory scanning and reputation checking as workflow step to feed directly into tooling.

* Integration
  + User-mode DLL loaded into various compatible windows processes
    - Scipt-host
    - Command interpreters
    - UAC
    - Office VBA Macros
  + Exposes APIs to be used by EDR software
    - AmsiScanBuffer()
    - AmsiScanString()
* AMSI Provider
  + Security products (Defender, CrowdStrike, etc.) register as providers via IAntimalwareProvider.
  + Implements Scan() to analyze buffers/scripts.
* AMSI Context
  + Session-based tracking (e.g., per PowerShell runspace).
  + Correlates scans to detect evasion (e.g., string splitting).
* Windows Defender (MsMpEng.exe)
  + Default AMSI provider; runs as a protected process (PPL).
* Key AMSI APIs (Attack Surface)
  + AmsiInitialize()
    - Initializes AMSI in the calling process.
  + AmsiOpenSession()
    - Creates a scan session (used for correlation).
  + AmsiScanBuffer()
    - Scans arbitrary memory buffers (primary target for bypasses).
  + AmsiScanString()
    - Scans strings (used by scripting engines like PowerShell).
  + AmsiResultIsMalware()
    - Checks if a scan result is flagged as malicious.
* AMSI Integration Points
  + PowerShell
    - Scripts/cmdlets are scanned before execution.
    - Hook point: System.Management.Automation.dll → AMSIClient.ScanContent().
    - .NET (CLR)
    - Dynamic assembly loads trigger AmsiScanBuffer().
    - Hook point: clr.dll → CAmsiAntimalware::Scan().
  + Office VBA/Macros
    - Scans VBA code via AmsiScanString().
  + WMI
    - Suspicious queries are intercepted.
* How AMSI Detects Malware
  + Signature-Based
    - Hashes, YARA-like rules for known patterns (e.g., "Invoke-Mimikatz").
  + Heuristics/Behavioral
    - Detects obfuscation (base64, XOR, string concatenation).
  + Contextual Analysis
    - Tracks API calls after scans (e.g., VirtualAlloc → WriteProcessMemory).
* AMSI Bypass Techniques
  + Memory Patching
    - Patch AmsiScanBuffer()
      * Overwrite function prologue (mov eax, 0x80070057) to force "clean" result.
      * Key APIs: VirtualProtect, WriteProcessMemory.
    - Patch AMSI DLL in Memory
      * Modify amsi.dll’s .text section to disable scans.
  + API Unhooking
    - Restore hooked AMSI functions
      * Overwrite jmp stubs in amsi.dll IAT with original code.
      * Tools: PowerShell Get-ProcAddress, P/Invoke to manipulate memory.
  + AMSI Context Manipulation
    - Forge Scan Results
      * Modify AMSI\_RESULT structs to return AMSI\_RESULT\_CLEAN.
    - Session Tampering
      * Corrupt HAMSISESSION to break scan correlation.
  + Direct Syscalls (Kernel-Level)
    - Call NtProtectVirtualMemory Directly
      * Bypass user-mode hooks via syscall instructions.
      * Tools: SysWhispers3, Hell’s Gate.
* Script-Specific Tricks
  + PowerShell
    - Force AMSI initialization failure (e.g., [Ref].Assembly.GetType() crash).
    - Use reflection to disable AMSI ($amsiContext = 0).
  + .NET
    - CLR Profiling API to intercept AMSI calls.
    - Dynamic Methods to JIT-unhook AMSI.
* Critical Knowledge for Bypass Development
  + AMSI’s Scan Triggers
    - When buffers/strings are scanned (e.g., PowerShell vs. C#).
  + AMSI Provider Architecture
    - How third-party AVs integrate (e.g., CrowdStrike’s csamsi.dll).
  + Process Injection & Hook Detection
  + AMSI’s reliance on user-mode hooks (easy to bypass).
  + Syscall Mechanics
    - How ntdll → kernel transitions work (for direct syscall bypasses).
  + AMSI Logging
    - ETW (Event Tracing for Windows) events for detection (e.g., Microsoft-Antimalware-Scan-Interface).
* **AMSI’s Default Execution Order (PowerShell Example)**
  + **Step 1: Script Submission**
    - **Trigger**: User runs a script (e.g., powershell.exe -c "Invoke-Mimikatz").
    - **Action**:
      * PowerShell host calls AmsiScanBuffer() **before execution**.
      * AMSI scans the **raw, uninterpreted script text**.
  + **Step 2: Initial Scan (Pre-Execution)**
    - **What’s Scanned**:
      * Literal strings (e.g., "Invoke-Mimikatz").
      * Syntax patterns (e.g., [Reflection.Assembly]::Load).
    - **Limitation**:
      * AMSI **cannot** evaluate:
        + **Runtime-generated strings** (e.g., "Inv" + "oke-Mimik" + "atz").
        + **Decoded data** (e.g., [System.Text.Encoding]::UTF8.GetString([Convert]::FromBase64String("..."))).
  + **Step 3: Deobfuscation & Execution**
    - **After AMSI Scan**:
      * PowerShell interprets the obfuscated script.
      * **Deobfuscation happens at runtime** (e.g., string concatenation, base64 decoding).
    - Key Insight:
      * AMSI **does not re-scan** runtime-resolved strings unless explicitly triggered.
      * The most frequent patterning in CrowdStrike bypass development:
        + Attacking AMSI and its integration with CS via csamsi.dll
        + Dynamic resolution of syscalls in memory through number

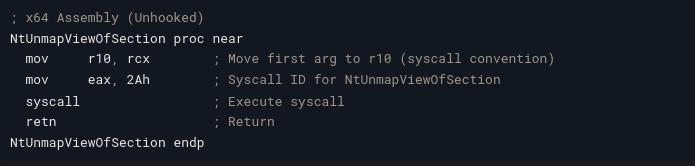
Seen in Hell’s Gate 2.0

* + - Capture AMSI Bayesian scores via ETW (in theory):
      * logman start AMSITrace -p Microsoft-Antimalware-Scan-Interface 0x8000 -o scan.etl
* Additional Reading
  + [AMSI Unchained](https://www.deepinstinct.com/pdf/antimalware-scan-interface-amsi-unchained)
  + [Using CLR Hosting to Evade AMSI](https://www.nttdata.com/global/en/-/media/nttdataglobal/1_files/services/cybersecurity/radar_magazine/2024/radar_supplement_july.pdf?rev=95736bd059584c05a31ef61eec8b83dd)

Section 3: Execution Flow Redirection

Malware developers operate by attempting to constantly trick and move EDR focus away from its key components and back to observably routine and benign functionality. Execution flow redirection (EFR) can be done many different ways, however like most of the techniques this document has shown (and will reveal further), modified variations of well-established approaches are the predominant components of successful evasion. EFR can be abstracted into two primary categories; global and local. Future versions of this document will overview other techniques. The current draft is restricted to the most vital three to understand.

* Global
  + Hooking
    - The best example of hooking can be found within an EDR’s monitoring component for API calls
    - Here is an example of what an unmodified API call might look like for the key component of process hollowing, NtUnmapViewOfSection

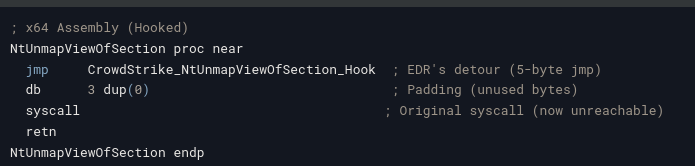


* + - When an image is executed and loaded into process memory, loaded Win32API libraries will be hooked by the EDR using a jmp instruction
      * This redirects execution to the EDR’s monitoring function, which will begin to perform actions that may include:
        + Bayesian calculations
        + Setting a watchpoint for the return value
        + Constructing a chronological call stack based on previous, current, and future calls

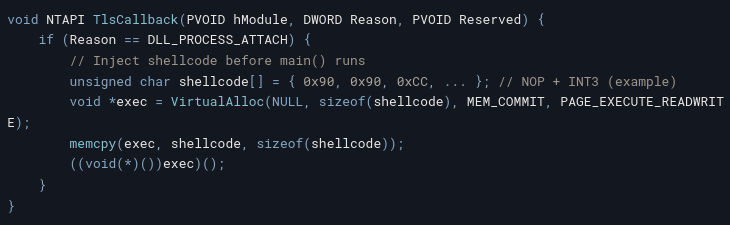
Note: callstacks are often not in a chronological order due to multi-threading

If single-threaded, the callstack will be in order

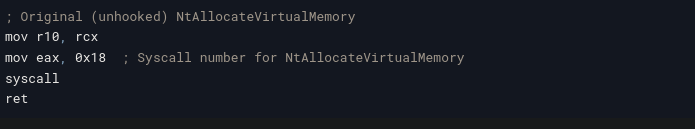
Getting an accurate callstack would require processor level dumping as the OS level does not provide enough logging or time-based calculations that can be obtained efficiently



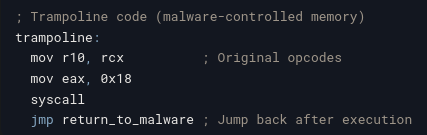
* + - Once execution of the monitoring function completes, the application may return to the original code and continue, terminate the application, or perform additional checks in other routines
* Local
  + TLS Callbacks
    - Thread Local Storage callbacks are used in applications for cleanup routines and initialization of data structures per thread (stability)
    - TLS callbacks are executed before normal code execution begins
      * This allows for many methods of abuse within malware from debugger/anti-analysis checks, to execution of entirely separate routines for early injection
    - Example of early injection in C++



* + Trampolining
    - Trampolines are counter-hooks to circumvent API monitoring routines through hooking, where the loaded target functions are overwritten with a jmp instruction and cannot be called without redirection
    - Trampolining involves the following steps (NtAllocateVirtualMemory as an example):
      * Locating the unmodified API in nt.dll



* + - * Allocating executable memory and copying instructions



* + - * Execution of trampoline and return to original code

