Memory Forensics

While traditional digital forensics often focuses on the post-mortem acquisition and analysis of non-volatile storage, Memory Forensics provides a critical window into the machine's state at a specific point in time. Through a technique known as Live Acquisition, an investigator can capture a dump of the device's RAM. This snapshot preserves highly volatile data that would otherwise be lost upon power-down, including:

- Active processes and their associated threads.
- Current network connections and listening ports.
- Browser history and unencrypted web activity.
- Registry keys and system event logs resident in memory.
- Loaded kernel drivers and files stored in the memory cache.

Random Access Memory (RAM). Random Access Memory serves as the primary volatile storage component of a digital system. It is fundamentally categorized into two types based on their architecture and application:

(1) *DRAM (Dynamic RAM)*: This is the most common form of system memory. It stores each bit of data within a separate capacitor and transistor pair. Because capacitors leak charge, DRAM requires constant refreshing to maintain data integrity. It is characterized by high density and relatively low cost, with modern DDR (Double Data Rate) modules operating between 1600 MHz and 5000 MHz. By transferring data on both the rising and falling edges of the clock cycle, DDR doubles the effective throughput.

(2) *SRAM (Static RAM)*: Utilizing a flip-flop circuit typically composed of six MOSFET transistors per bit, SRAM does not require refreshing. This architecture allows for significantly faster access times than DRAM, though at a higher cost and lower density. Consequently, SRAM is primarily utilized for CPU caches (L1, L2, L3) where speed is the absolute priority.

The Operating System manages memory by mapping Physical Memory to a Virtual Memory space via the Memory Management Unit (MMU). This abstraction allows the system to address more memory than is physically available by utilizing Swap Space - a designated area on the hard drive used to supplement RAM. To manage this efficiently and prevent fragmentation, memory is divided into small, fixed-size units called pages. The mapping between virtual and physical addresses is maintained through Page Tables. While the system can "page out" most data to the disk to free up RAM, certain critical components, such as kernel drivers and core OS structures, reside in Non-paged Memory. This memory is "pinned" to the physical RAM and can never be moved to the swap file, ensuring that essential system instructions are always immediately available to the CPU.

1.1 Windows Memory Organization

In a Windows environment, memory is strictly partitioned to maintain system stability and security. Each process operates within its own virtual address space, typically divided into User

Land and Kernel Land. Kernel Land is a protected region of memory reserved for the operating system. It houses critical data structures, including device drivers, memory paging tables, and the kernel itself. Conversely, User Land contains the components specific to an active process:

- *Program Image*: The executable code and static data loaded into memory.
- *Heap*: A region of memory used for dynamic allocation during runtime.
- *Stack*: Used for local variables and function call management; the stack typically grows towards lower memory addresses.
- *PEB & TEB*: The Process Environment Block (PEB) and Thread Environment Block (TEB) are essential data structures in Windows. They store metadata such as loaded modules (DLLs), environment variables, and heap pointers required for the process and its threads to function.

While the Program Image is traditionally located at higher addresses than the stack and heap, modern security features like Address Space Layout Randomization (ASLR) can shift or reverse these positions to prevent exploitation. It is important to note that Linux manages these structures differently: it lacks the PEB/TEB architecture, utilizing the vDSO (virtual Dynamic Shared Object) for system call efficiency and substituting Windows' DLLs with `.so` (shared object) files.

1.2 Memory Artifacts and Investigation

A memory forensics investigation focuses on extracting specific artifacts that reconstruct the activity of a system. These artifacts provide a granular view of the machine's state that disk forensics might miss:

- *Handles*: These are opaque tokens that represent a process's access to system resources. They indicate the presence of cached files, registry hives, and network sockets currently held open by a process.
- *Network Artifacts*: Memory captures reveal active network connections, listening ports, and historical connection attempts that may no longer be visible in live logs.
- *Processes and Threads*: These represent the actual execution flow. While a process is a container for resources, threads are the individual units of execution scheduled by the CPU.
- *Code Injection*: Malware often attempts to evade detection by injecting executable code into legitimate process memory. Forensic analysts can identify these anomalies by applying specific rule sets, such as YARA rules, to scan for malicious patterns or unauthorized memory permissions (e.g., `PAGE_EXECUTE_READWRITE`).

The Volatility Framework

Volatility is the industry-standard, open-source framework for incident response and memory forensics. It is a command-line interface (CLI) tool written in Python, compatible with Windows, Linux, and macOS. The framework underwent a significant architecture shift to improve performance and accuracy:

- *Volatility 2 (Legacy)*: This version relies on Profiles to interpret memory dumps. A profile is a static set of definitions describing the data structures and offsets of a specific OS version.

Identifying the correct profile can be a slow, manual process, and an incorrect profile often leads to corrupted or missing data.

- *Volatility 3 (Modern)*: Rewritten for Python 3, this version introduces a Symbol-based approach. Instead of static profiles, it uses intermediate symbol files derived from kernel debugging information. This shift makes the framework significantly faster, more effective, and capable of automatically identifying the operative system without manual profile guessing.