Live Memory Forensics without RAM Extraction

1st Anh Khoa Nguyen 2nd Dung Vo Van Tien 3rd Khuong Nguyen-An

***Abstract*—Memory forensics is a widely used technique in malware hunting. In this technique, researchers try to extract information from the Windows kernel space to learn more about the behavior of every program on the system. Traditionally, there are two ways to access these information: calling Windows API or memory analysis. However, both methods have their own limitations. Firstly, Windows often withholds some critical runtime information of the kernel. Therefore, we can miss some valuable data if we rely solely on functions from the Windows API. The other approach, memory analysis, most of the time involves processing a memory dump file, which is a complicated and time-consuming process. In this paper, we present a method to perform memory forensics on a Windows machine that avoids the use of Windows APIs or memory dump. This approach allows us to directly access system information in the kernel space without calling Windows APIs. It can also perform memory analysis on a currently running machine to detect suspicious behavior, which is usually only found when analyzing memory dumps during malware analysis. With a little extension, it can also act as a live WinDBG. The proposed method is implemented in an open-sourced framework called LPUS. This is a valuable tool for system administrators, security researchers, and malware analysts. It can be used to gain a deeper understanding of the Windows kernel and to detect malicious activities.**

***Index Terms*—pool scanning, pool tag scanning, Windows memory forensics, page table entry, live memory forensics**

I. Introduction

Malware are malicious software designed with the intention of causing damage to computer systems. Nowadays, as computer usage continues to rise, the risk of malware attacks is also growing. One of the most effective and popular methods to detect traces of malware on a system is memory forensics. It is a method to extract system information from a snapshot of a computer's memory (memory image). Memory forensics can give analysts valuable data about processes, threads, files, drivers, network connections, .etc at the time of the malware attack. A subset within the field of memory forensics is live memory forensics, which involves an investigator looking for evidence in real time on a running machine.

A. Motivation

Currently, most memory forensics tools rely heavily on system memory extraction. There are not many ways for investigators to perform live forensics on a computer system. Our objective is to develop a tool that enables live memory forensics to utilize the same techniques used on memory images.

Over the years, extensive research has been conducted on Windows kernel memory, resulting in a comprehensive understanding of its workings. Many forensics techniques use this knowledge for extracting information from a memory image. One such technique, called pool tag (quick) scanning, can be enhanced to enable real-time scanning on a running system.

Along with pool tag scanning, other techniques that are widely used on memory images could also be utilized for live forensics. By examining internal structures of the Windows kernel for memory paging like Page Table Entry or Page Frame Number Database, we can correctly identify various code injection techniques, which are used widely in malware to avoid detection.

B. Contributions

The contributions of this paper are:

- A method to perform pool tag scanning on a live machine.
- A method to detect code injection techniques on a live machine
- A live memory forensics tool capable of extracting system information and detecting potentially malicious behaviors.

II. Windows OS Background

In this section, we will clarify terms relevant to the Windows Operating System. These terms are important for readers to establish a fundamental background on Windows, which is essential for understanding concepts in Memory Forensics techniques for the Windows platform.

A. Access Tokens and Process Privileges

In the Windows operating system, *Access Tokens* [34] play a vital role in managing what actions a program can perform. These tokens are associated with specific users and are copied into the program's context . Access Tokens hold a list of permissions that dictate what the program can do. For instance, if a program lacks the `SeShutdownPrivilege` in its token, it won't have the privilege to shut down the system.

A program is equipped with only a portion of the privileges available to the user. If the program needs to activate a privilege that is initially inactive, it must first obtain access to the Access Token and change the privilege's status. However, this adjustment of the token is subject to a safeguard; the process must also possess the `TOKEN_ADJUST_PRIVILEGES` privilege. The adjustment of process' token are illustrated in Listing I.

B. Virtual address space

Address virtualization is a common memory management method used in many modern operating systems. In this method, a process does not access data in RAM using raw

```
LPCTSTR privilege;
TOKEN_PRIVILEGES tp;
LUID luid;

assert(LookupPrivilegeValue(
        NULL,
        privilege,
        &luid));

assert(AdjustTokenPrivileges(
       hToken,
       FALSE,
       &tp,
       sizeof(TOKEN_PRIVILEGES),
       (PTOKEN_PRIVILEGES) NULL,
       (PDWORD) NULL));
```

Listing I: Adjustment of Process Privileges

offset, and instead uses a layer of abstraction, called *virtual address* (or logical address). Each process running on the system will be given a continuous range of addresses, called a virtual address space. This address space will be divided into many *pages* for easy management and allocation. Each page in the virtual address space will be mapped to a corresponding memory region in the physical memory.

The processor handles the paging of physical memory [12], [27]. This involves using a set of pointer tables to define a specific virtual address. Along with the CR3 register, which points to the first table, the virtual address can be divided into components. These components are used to navigate through each nested table, allowing for the calculation of the corresponding physical address. The calculation from virtual address to physical address is called address translation and is illustrated in Figure 1.

Within the Windows operating system, processes can be categorized into two distinct types. First, there are user-mode processes, e.g., web browsers and notepad. Each of these user-mode processes operates within its own isolated virtual address space, which is further divided into two segments: the *user-space* and the *kernel-space*. On the other hand, we have kernel-mode processes, which encompass critical components of the operating system, such as the kernel and drivers. These kernel-mode processes share a common virtual address space. This shared address space of kernel-mode processes is then mapped to the system address space of all user-mode processes.

C. Windows kernel pool

The *kernel pool* in Windows is the heap section for kernel-mode processes. Windows reserves multiple pools for different allocation purposes. The two most general types of pools are paged pool and non-paged pool, both serve as general pools for object allocation. Paged pools allow the content to be paged out while data in the non-paged pool are guaranteed to be in physical memory at all times.

An allocation in a pool is called a block. A block has two main sections, the header, and the data. The header is a structure named `_POOL_HEADER` containing information about the block itself, such as the size of the block, the previous block size, the index of the block in the memory page, and a tag value. The tag value is a required four-byte value that is given when Windows allocates a kernel space through APIs such as `ExAllocatePoolWithTag` [36]. This tag value, also called pool tag, is used by Windows operating system developers and kernel driver developers to reference the data they want to store. As a rule of thumb, developers usually define and use tags that are somewhat related to their drivers. A table of tag values and their related Windows structures is provided in Table I.

Structure	Structure Name	Pool Tag
Driver Object	_DRIVER_OBJECT	Driv
File Object	_FILE_OBJECT	File
Process	_EPROCESS	Proc
Thread	_ETHREAD	Thre
TCP endpoint		TcpE
TCP listener		TcpL
UDP endpoint		UdpA

TABLE I
LIST OF OBJECTS AND THEIR POOL TAG

D. Windows Kernel

The Windows kernel contains structures responsible for the management of the operating system. Not only structures, some variables are also defined globally and accessible through the *system space*. The kernel is distributed as an executable named `ntoskrnl.exe` and located inside the Windows operating system directory. Often times, researchers address the kernel by its internal name `ntkrnlmp`.

Below we list a few common Windows kernel structures that are commonly used for memory analysis.

1) _EPROCESS: On Windows, the data related to a running process is saved inside a structure called `_EPROCESS` [40]. This structure provides various information for the memory forensics process, such as process id, parent process id, a pointer to the list of threads belonging to the current process, etc.

We can also inspect the virtual memory space of each process using the data from `_EPROCESS`. One popular method used by various forensics tools is traversing the **Virtual Address Descriptor** (VAD) tree (`VadRoot` field in `_EPROCESS`) to enumerate all memory allocation requests that the process has made and from there list all the memory regions in the process.

`_EPROCESS` objects are stored inside the system address space and arranged into several doubly linked lists, which can be traversed by a kernel-mode driver to enumerate all processes running on the system. The *PsActiveProcessHead* and *KiProcessListHead* global variables store the addresses of these linked lists, allowing for easy access to the information.

2) _ETHREAD: `_ETHREAD` [41] is an internal structure of the Windows kernel to represent a thread. Researchers can use this structure to find many information about the corresponding thread, such as the create time and exit time of the thread, the process that created the thread, etc.

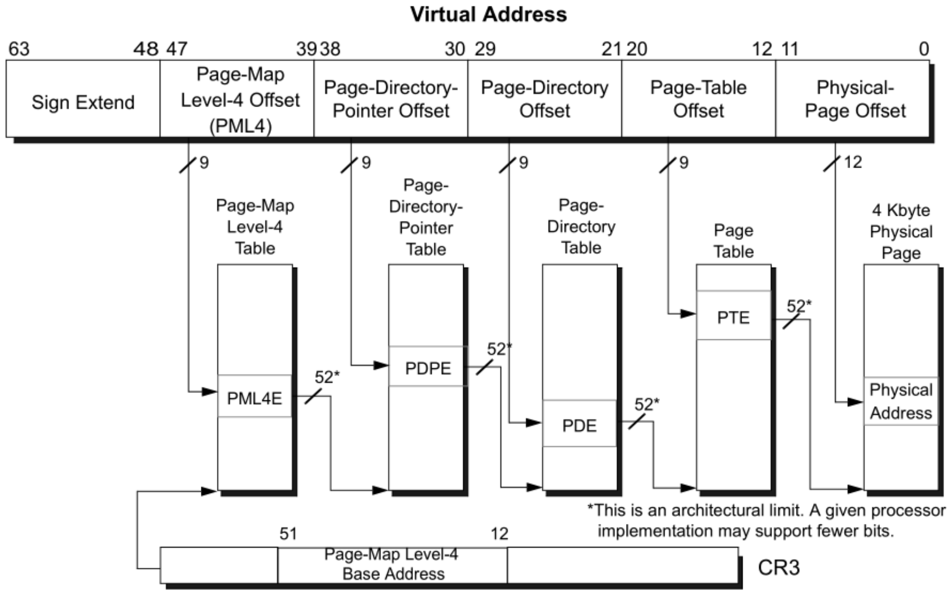

Figure 5-17. 4-Kbyte Page Translation—Long Mode

Fig. 1. Address Translation (taken from AMD64 Architecture Programmer's Manual Volume 2.)

All the `_ETHREAD` structures belonging to the same process are organized as a link list. We can traverse this link list using the field `ThreadListEntry`.

3) Virtual Address Descriptor: : Virtual Address Descriptor (VAD) is a structure in the Windows operating system used to manage memory allocations in user-mode processes. Internally, this structure is named *_MMVAD* [43]. Each process has its own set of VADs, which is organized as an AVL tree. As mentioned in section II-D1, the address of the root node of the VAD tree is stored in the ***VadRoot*** field of the `_EPROCESS` structure.

The VAD tree contains various valuable information for memory forensics. For example, the *Protection* flag in the *VadFlags* field, which describes the actions that a process can perform on the specified memory region, is widely used to detect multiple common hiding techniques in malware.

E. Kernel debug symbols

Although the Windows source code is not accessible to the public, Windows provides debug symbols for research use. These symbols as distributed under a special file format, called program database (or PDB). There is some information on how to extract data from PDB files but they are very limited [24], [33], [35].

A PDB file contains information such as the location of global variables, functions, and the layout of internal structures. Each PDB file is unique to one specific version of an executable. Most if not all PDB files for Windows components can be downloaded from Microsoft Symbol Server at https://msdl.microsoft.com/download/symbols.

When a binary is compiled with its PDB file, usually through using the MSVC (Microsoft Visual C++) compiler, the binary includes a 24-byte header that serves as an identifier for the corresponding PDB file. This header comprises of:

- A 4-byte magic value, typically set to "RSDS."
- A 16-byte Globally Unique Identifier (GUID), which uniquely identifies the PDB file in the database.
- A 4-byte version number, commonly referred to as "age".

The GUID is the unique identifier used to locate the PDB file associated with a specific binary. For example, when attempting to download the PDB for a particular Windows kernel module, such as `ntoskrnl.exe`, you would request the PDB file for its kernel name, `ntkrnlmp.pdb`, along with the GUID and age values found within the binary.

III. Malware techniques

In this section, we introduce common malware technique that are used to evade detection. These could be techniques that work on user-space or kernel-space. Usually, these are not easy to discover unless the EDR system is capable of monitoring all system calls.

A. Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) refers to a set of malware techniques that involve the direct modification of kernel objects within the Windows operating system [10], [25], [30]. One of the most common DKOM techniques is the modification of the process list, which aims to remove a specific process from the list. This manipulation allows the malware to hide itself, as it won't be visible through standard API calls used for listing processes.

Other DKOM techniques involve altering control bits within certain Windows kernel global variables. By changing these bits from false to true or vice versa, the malware can potentially enable or disable specific Windows features.

It's worth noting that DKOM has become less common in recent times. Several factors contribute to this decline, including the increased difficulty of making such modifications due to advancements in Windows security mechanisms and the extensive efforts required to find suitable locations for these modifications. However, DKOM techniques remain relevant in certain highly sophisticated and elevated attacks carried out by advanced malware groups.

B. Syscalls Hooking

Operating systems manage various operations through a series of system calls, often referred to as syscalls. These syscalls are the interfaces that applications use to request services from the operating system. In Windows, these syscalls are typically implemented as Windows APIs and are managed within the Windows kernel.

Syscalls hooking is a technique where hooks are installed on these syscall functions for various purposes. Malware often uses syscall hooks to prevent itself from being detected or removed, while anti-virus software uses them to monitor system API invocations and filter out potentially malicious behavior.

In Windows, syscalls are managed through a table called the System Service Dispatch Table (SSDT). A driver had the ability to access this table and modify its entries to redirect basic operations to customized functions, typically for hooking purposes. However, with the introduction of Kernel Patch Protection, also known as PatchGuard, direct modifications to the SSDT are no longer possible. PatchGuard is a security feature that aims to prevent unauthorized modifications to the kernel and maintain the integrity of the operating system. This makes it more challenging for both legitimate software and malware to tamper with system functions and APIs.

The complexity of the Windows Operating System suggests that there may be other advanced methods for syscall hooking that go beyond the straightforward modification of the SSDT. These advanced techniques often require a deep understanding of multiple components of the Windows operating system. While we won't delve into the specifics of these techniques in this discussion, it's important for readers to be aware that such advanced methods for syscall hooking remain relevant and are actively explored (and exploited) in modern Windows systems.

C. Code injection

Code injection is a popular evasion technique used in malware. In this technique, the malware will try to write a piece of malicious code in the virtual address space of a legitimate process on the system, then force this process to execute said code. Nowadays, malware authors are developing more and more methods to perform code injection. In this section, we will explore some code injection techniques that are used by malware.

1) Remote code injection: This is the simplest code injection technique. The goal of this technique is to inject a piece of machine code, which is called "shellcode", into a specified process. This technique is carried out in 3 steps:

- The malware creates a new memory region inside another process's address space with PAGE_EXECUTE_READWRITE permission.
- The malware inserts its shellcode into the newly created memory region.
- The malware forces the process to execute the injected code (for example, using the *CreateRemoteThread* function).

2) DLL injection and Reflective DLL injection: In this technique, malware store the malicious code in a DLL file and then force a legitimate process on the system to load this harmful DLL into its address space. There are many methods that malware can use to achieve that. For example, they can use the Remote code injection technique to force the process to load a DLL file from secondary memory via the *LoadLibrary* function. Malware can also perform this technique by leveraging some functions provided by Windows such as *SetWindowsHookEx*.

One of the most effective and popular techniques for code injection using DLLs is *Reflective DLL injection*. The main difference between Reflective DLL injection and other classic DLL injection techniques is that instead of using available functions from the Windows API to load a DLL stored on disk, this new technique implements its own loader to be able to load the DLL directly from RAM. This approach offers notable advantages as it reduces reliance on the Windows API, eliminates the need to store malicious code as a file, and therefore enhances the ability of malware to evade detection by anti-virus and forensics tools.

3) Process Hollowing: The idea of the *Process Hollowing* technique involves initializing a new process, then removing all of its executable code and replacing it with malicious code. After the process is modified, the resulting process appears to be a legitimate process from the outside, but in reality, most of its executable code is malicious.

The Process Hollowing technique is usually implemented following four main steps:

- Initialize a new process in the suspended state.
- Unmap the memory region containing the executable code of the legitimate process. Malware often uses functions such as *NtUnmapViewOfSection* or *ZwUnmapViewOfSection* to achieve this.
- Create a memory region with PAGE_EXECUTE_READWRITE permission in the suspended process to write the malware's executable code.
- Change the instruction pointer address of the current process to the address of the memory region where the malicious code is stored, then resume the process using the *ResumeThread* function.

IV. Volatile Memory Forensics and Techniques

In this section, we present the concept of Volatile Memory Forensics and outline some common techniques associated with it. We also introduce the contemporary notion of Live Forensics. The combination of Live Forensics with Memory Forensics is an actively discussed topic which is also relevant to our research.

A. Volatile Memory Forensics

Volatile Memory Forensics refers to the practice of capturing and examining the physical memory of a computing system in response to an incident. Extensive research has been conducted to extract critical information from this memory. Techniques for capturing physical memory were initially developed to maximize data preservation. Subsequently, researchers have analyze the raw binary data from these memory extractions by leveraging the way the Windows kernel manages physical memory. These techniques primarily focus on identifying traces of processes, threads, files, and internet connections. Moreover, given the tactics employed by malware, specialized techniques have been devised to detect signs of specific malware behaviors.

Currently, the state-of-the-art tool for conducting Volatile Memory Forensics is Volatility3 [20], developed by the Volatility Foundation. Prior to that, Volatility [19], which originated as a modest project by Aaron Walters that combined research in Memory Forensics into a single, versatile tool, was widely utilized. Rekall [23], a fork of Volatility initiated by Google's Mike Cohen, gained some attention but was eventually discontinued.

B. Live Forensics

Live forensics is a specialized field of forensics that revolves around analyzing a computer system while it is actively running, or in a "live" state. This analysis is typically carried out by utilizing Windows APIs to interact with the system. Through extensive use of these APIs, analysts can obtain a substantial amount of information about the system. This information encompasses various aspects, including but not limited to processes, threads, files, registry entries, and network packets. The data obtained during this analysis can be collected from both user-space and kernel-space, with kernel-space potentially offering a more comprehensive set of information.

Over the years, numerous tools for Live Forensics have been developed and widely adopted by researchers for incident response. Table II provides a summary of some of these tools and their associated information collection capabilities.

Name	Inspection
Wireshark [21]	Network
Autoruns [44]	Start-up items
Process Explorer [45]	Processes and Threads
PE-sieve [16]	Injected code
Process Monitor [46]	System's Activities
Process Hacker [32]	System's Activities
System Informer [52]	System's Activities

TABLE II
Common Live Forensics tools for incident response

Live Forensics has expanded beyond the exclusive use of Windows APIs. With the maturation of Memory Forensics techniques, these methods have found their way into the realm of Live Forensics. The incorporation of Memory Forensics techniques has demonstrated the capability to yield more comprehensive details compared to traditional use of Windows APIs. This methodology typically involves an initial step of extracting physical memory. Subsequently, Memory Forensics techniques are applied to this memory extraction, unveiling information through thorough indexing and searching.

These motivating factors have spurred the development of several tools that integrate both Live Forensics and Memory Forensics. Tools such as [17], [22], [23], [49] serve as examples, where they conduct memory extractions and subsequently perform analysis on the extracted files.

For the sake of brevity, we will refer to Live Forensics using Memory Forensics techniques as ***Live Memory Forensics***.

C. Malware Detection using Memory Forensics

Memory Forensics is a crucial process conducted when a security incident occurs to gain insight into what occurred. Often, a sample of the malware responsible for the incident resides within the memory extraction, enabling further analysis. However, pinpointing the location of these samples can be a challenging task, as many malware techniques are employed to conceal their presence. Fortunately, advancements in Memory Forensics have simplified the process of locating such samples, especially after an incident has taken place.

While these powerful tools have proven effective for post-incident analysis, they are not commonly used for real-time malware detection due to the high overhead associated with Live Memory Forensics, as well as the lack of full automation. However, in theory, by applying Memory Forensics to identify irregularities, we should be capable of detecting malware, provided their techniques are evident through Memory Forensics methods. Works like [26], [28] have demonstrated that this technical aspect is worth exploring further in the future.

D. Locating the kernel base address

An essential aspect of Memory Forensics is the identification of the kernel base address. The ability to locate this address is crucial for analysts to navigate and inspect the system. Historically, it was relatively straightforward to find critical variables by conducting a brute-force search, especially for structures of KDBG or KPCR. KDBG, for instance, contains pointers to the kernel process list and loaded libraries list, while KPCR holds the CR3 register value used for address translation. These lists are kernel global variables, and their addresses are typically offset from the kernel base address. However, in modern Windows versions, locating these variables has become increasingly challenging or even impossible.

In Volatility3, the developers utilize a unique and undocumented technique referred to as the ”pdb signature scanner” [1] to discover potential kernel base addresses and employ heuristics to effectively filter out false values. In essence, this method involves a brute-force search of the entire memory for the PDB header of the kernel executable.

> Volatility provides a comprehensive compilation of important structure definitions and offsets of global variables. The Rekall project, which has been discontinued, as well as Volatility3, a rewrite of Volatility, obtain these offsets and structure definitions directly from PDB files.

E. Pool Tag Scanning

While using the kernel base as a reference point for effectively navigating the kernel systems is a common method for analyzing a physical memory file, it may sometimes prove insufficient. This inadequacy arises from the fact that certain values are not explicitly referenced or are *freed-but-not-overwritten*. To address this limitation, [47] introduced the concept of Pool Tag Scanning, a technique that can discover (potentially) all objects within the kernel pool.

As detailed in Section II-C, all objects allocated by the kernel are stored within the kernel pools. These objects are often allocated using the function `ExAllocatePoolWithTag` and are conveniently tagged with a 4-byte value. By searching for these specific 4-byte tags and implementing heuristics to eliminate false positives, there is a high likelihood of discovering all objects within the kernel pool. This approach is valuable for revealing critical information during memory forensics.

Nevertheless, performing a brute-force search throughout the entire physical memory image can be highly inefficient, particularly with modern computers typically equipped with a minimum of 8GB of RAM, and often having 16GB or more. Fortunately, Windows reserves a dedicated and more confined region for the kernel pool. It has been proven that searching for objects within this reserved region is significantly faster while maintaining a high level of precision. This entire process, including the details of how to search within this kernel pool region, is comprehensively documented in [48] and the technique is also named Pool Tag **Quick** Scanning.

In essence, the virtual addresses that mark the beginning and end of the kernel pool are indicated within kernel global variables. Starting from Windows 10, these values have been relocated to a global variable named ”MiState” of struct `_MI_SYSTEM_INFORMATION` [42]. Listing II illustrates how each version of Windows stores the pool range.

```
// on Windows 11
struct _MI_SYSTEM_INFORMATION* MiState;
struct _MI_SYSTEM_NODE_NONPAGED_POOL* Pool
    = MiState->Hardware.SystemNodeNonPagedPool;
PVOID* PoolStart = Pool->NonPagedPoolFirstVa;
PVOID* PoolEnd = Pool->NonPagedPoolLastVa;

// on Windows 10
struct _MI_SYSTEM_INFORMATION* MiState;
struct _MI_SYSTEM_NODE_NONPAGED_POOL* Pool
    = MiState->Hardware.SystemNodeInformation;
PVOID* PoolStart = Pool->NonPagedPoolFirstVa;
PVOID* PoolEnd = Pool->NonPagedPoolLastVa;
```

Listing II: Windows 10 and 11 kernel pool range

F. Detecting code injection

1) Detection based on VAD: In most code injection techniques, the malware needs to initialize a memory region with all 3 permissions: read, write, and execute, in order to be able to write and execute the injected code. A normal process in the system rarely allocates a memory region with all 3 access permissions. Therefore, a common method used to detect injected memory regions is to search for any VAD with the Protection flag set to `PAGE_EXECUTE_READWRITE`.

The VAD-based code injection detection approach is widely used by detection tools. In fact, many state-of-the-art tool nowadays works by scanning the VAD tree to look for memory regions with both write and execute permission. For example, the plugin **malfind** [18] is a prime example of this approach, as it walks the VAD tree and notifies the analyst if it finds a VAD that has the `PAGE_EXECUTE_WRITECOPY` protection flag. This plugin is available for both Volatility and Rekall and has become a standard plugin that gets installed with Volatility by default.

Another approach to using VAD for detecting code injection is combining it with the Process Environment Block (PEB). In 2016, Monnappa created the plugin **HollowFind** [38] for Volatility. In addition to checking the protection flag in a VAD node, **HollowFind** compares the information inside PEB and VAD to detect whether malware has unmapped the process's original code. Another technique utilized by this plugin is examining parent-child relationships of running processes in the system. It identifies system processes that are started by the wrong parent process and uses it as a signature to detect malware. By combining all these methods, **HollowFind** can accurately pinpoint processes that have been affected by Process Hollowing with a very high degree of accuracy.

In 2016, Monnappa presented another plugin for Volatility, named **Psinfo** [39], in an attempt to combine all detection techniques from **malfind** and **HollowFind**. The goal of this plugin is to allow a security analyst to get process-related information and spot any process anomaly without having to run multiple plugins.

In 2017, Aleksandra Doniec introduced **PE-sieve** [16], an open-source tool with the ability to detect multiple types of code injection attacks. The tool is designed to analyze a running process and identify potential shellcode injection, Process Hollowing, and other types of malicious activity. It is used as a base engine for many other malware-detecting tools like HollowsHunter [14], mal_unpack [15], etc.. **PE-sieve** utilizes many different techniques, such as:

- Compare the image loaded to memory with the original

[1] https://github.com/volatilityfoundation/volatility3/blob/v2.5.0/volatility3/framework/symbols/windows/pdbutil.py#L319

program stored on disk.
- Use VAD to detect PAGE_EXECUTE_WRITECOPY regions.
- Use VAD and PEB to detect Process Hollowing.
- Detect PE image loaded in private memory.

One of the standout features of **PE-sieve** is its capability to work on a live machine. Most of the other detection tools at the time is either a plugin for Volatility or Rekall. Therefore, they are only used to analyze memory dumps. It means that to use these plugins, users have to first extract the RAM from their memory. The ability to perform live forensics makes **PE-sieve** much more accessible and easier to use.

G. Detection tools based on PTE and PFN database

Detection methods using VAD have certain limitations. VAD only contains the initial protection flag of a memory region, so malware can allocate the malicious buffer in a particular manner to avoid detection from forensics tools. Due to these drawbacks, researchers started to seek alternative techniques that do not heavily depend on VAD.

In 2019, Block et al. [9] proposed to use PTE and PFN database for code injection detection and developed the **PteMalfind** [8] plugin. As of 2023, this plugin supports Rekall, Volatility, and Volatility 3. The tool works by traversing the paging structures of a process to extract information from all the PTEs, then combining this information with the PFN database to detect any injected code. The tool can detect various code injection techniques such as Process Hollowing, Remote code injection, Atom bombing, etc.

V. Live Memory Forensics Without Extraction

Current Live Forensics based on Volatile Memory Forensics relies heavily on the extraction of physical memory. This introduces an additional step of extracting and determining the kernel base address, which can be inefficient due to brute-force searches. In this section, we introduce our novel approach to Live Forensics based on Volatile Memory Forensics, which operates exclusively within the system's memory and doesn't necessitate the extraction of RAM. Our method demonstrates the ability to perform various Memory Forensics techniques with minimal adjustments to the traditional file-based approach.

A. Overview

In our approach, we utilize a combination of a kernel driver and a user-space program. The kernel driver provides access to the kernel-space and is managed by the user-space program through a series of IOCTL[2] calls or similar communication methods.

Additionally, we leverage the PDB file of the kernel to obtain offsets for all global variables and structure definitions. The user-space program retrieves the PDB file associated with the running system.

[2]https://learn.microsoft.com/en-us/windows/win32/devio/device-input-and-output-control-ioctl-

Through the established communication channel between the kernel driver and the user-space program, we can construct intricate logic, as elaborated in the subsequent sections, allowing us to conduct live forensics directly in memory without the need for RAM extraction.

B. Accessing the kernel-space

Accessing the kernel-space is accomplished through the kernel driver, and there are various methods for loading this driver. It can be configured to load automatically during system boot or loaded on-demand when necessary. In either case, Windows requires that the kernel driver be defined within the registry at the following location: `HKLM\SYSTEM\CurrentControlSet\Services`.

If the driver is loaded on-demand, a user program with the `SeLoadDriverPrivilege` privilege is required to issue the `NtLoadDriver` command for loading the kernel driver.

C. Acquiring the kernel base address

In our approach, determining the kernel base address is a relatively straightforward task. When the kernel driver is initiated (**DriverEntry**), it queries the current process by calling `IoGetCurrentProcess`. Due to the way Windows manages drivers, this process corresponds to the `_EPROCESS` structure for the system process. Following this, it traverses the process list backward to the head, which is typically only 1 to 2 steps away, as the system process is usually the first item in the list. Conveniently, the head of the process list corresponds to the address of the kernel's global variable, `PsActiveProcessHead`. By using the offset of `PsActiveProcessHead`, we can readily calculate the kernel base address.

The structure of `_EPROCESS` and the offset of `PsActiveProcessHead` are obtained from the PDB. In recent Windows versions, the list pointers (previous and next) within `_EPROCESS` remain consistent. We hard-code these values into the kernel-space, enabling us to calculate `PsActiveProcessHead`. Subsequently, we return this address to the user-space, where the kernel base address can be determined from the offset extracted from the PDB.

Alternatively, in an advanced approach, the user-space can provide the kernel with all the requisite values for calculating the kernel base address.

Listing III provides an example of how to obtain the kernel base address. We've tested this sample on Windows 7, 10, and 11, consistently achieving the correct result. Nevertheless, for added certainty, a full traversal until the head (where the process name is empty) is found is recommended.

It's worth noting that there may be alternative methods for finding the kernel base address, as suggested in [] (reference needed). While all of these proposed methods, including our own, may not be proven correct, through rigorous testing and validation, they have demonstrated their reliability and accuracy.

```
PVOID systemEprocess;

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT     DriverObject,
    _In_ PUNICODE_STRING    /* RegistryPath */
) {
    systemEprocess = IoGetCurrentProcess();
}

void calculate_kernel_base() {
    // eprocessLinkOffset + listBLinkOffset
    ULONG64 backPointer; /* from PDB */
    ULONG64 processHeadOffset; /* from PDB */

    PVOID processHead =
        (PVOID)(*(ULONG64*)((ULONG64)
        systemEprocess + backPointer));
    PVOID ntosbase =
        (PVOID)((ULONG64)
        processHead - processHeadOffset);
}
```

Listing III: Acquiring the kernel base address

D. Perform Pool Tag Scanning

Pool Tag Scanning can be executed in its variations, Pool Tag Quick Scanning, albeit with substantial modifications to ensure stability. Given that we are now working directly with kernel virtual memory, any illegal memory access can lead to a kernel crash. This is due to the possibility of encountering virtual ranges that is unmapped (lacking a physical backup page), as explained in [48], "sparse allocation of virtual address space, enabling the kernel to reserve a large range of addressees for a pool, but to only allocate physical pages when needed".

Following the steps of Pool Tag Quick Scanning, we can easily locate the non-paged pool range using the kernel base address and offsets from the PDB. Recall that these values are available in `MiState` for Windows 10 and later.

During the scanning process, we incorporate checks at the beginning of each page to verify its accessibility. If a potential object is discovered, and its size exceeds the boundaries of the page, it is rejected. To determine the accessibility of a page, we utilize the function `MmIsAddressValid`.

Subsequently, the kernel driver transmits the addresses of potential objects back to the user-space program. For each object, if the structure is well-documented, the user-space program can request the kernel to conduct successive memory reads to fully extract the object. It is highly probable that there will be no invalid memory access issues since all the objects discovered are in the non-paged pool, which remains in physical memory and is not subject to paging out.

E. Code injection detection method

Among the techniques for detecting code injection discussed above, we decide that using PTE and PFN database is the most effective method. This technique solves the issues of previous VAD-based tools. Moreover, the information about a page's protection flag could also be extracted from the PTE. Therefore, we can use PTE to scan for `PAGE_EXECUTE_READWRITE` pages instead of VAD.

Tool name	**Detection method**			**Able to work on a running machine?**
	VAD	VAD and PEB	PTE and PFN database	
Malfind	x			
HollowFind		x		
PE-seive	x			x
PteMalfind			x	
Our method			x	x

TABLE III
OUR APPROACH COMPARED TO OTHER DETECTION TOOLS

Another reason to choose the PTE and PFN database approach is that despite being an excellent technique, it is still relatively new and has not yet been widely adopted by many of the latest detection tools. As shown in Table III, most well-known tools still use VAD to identify code injection. Moreover, there currently aren't any live forensics tools that incorporate this method, primarily because it is difficult to access and extract data from the PTE and PFN database. These structures serve as an interface between Windows and the CPU, and obtaining information from such low-level structures on a running system is very challenging.

With the goal of detecting code injection using PTE and PFN database, our tool needs to perform the following tasks:

1) Find every `_EPROCESS` structures inside the memory. Each `_EPROCESS` structure represents a running process on the computer.
2) From each `_EPROCESS` structure, extract the address of every paging structure of the process.
3) Traverse the paging structures to locate the PTE(s).
4) Parse and extract the needed information from the PTE.
5) Traverse the PFN database to extract the needed information.
6) Combine the data from PTE and the PFN database to detect the memory regions where malware has injected their code.

```
NTSTATUS openPhysicalMem() {
    NTSTATUS ntStatus = STATUS_SUCCESS;
    RtlInitUnicodeString(
        &ObjectNameUs,
        L"\\Device\\PhysicalMemory");

    InitializeObjectAttributes(
        &ObjectAttributes,
        &ObjectNameUs,
        OBJ_CASE_INSENSITIVE,
        (HANDLE)NULL,
        (PSECURITY_DESCRIPTOR)NULL);

    ntStatus = ZwOpenSection(
        &physicalMemHandle,
        SECTION_ALL_ACCESS,
        &ObjectAttributes);
}
```

Listing IV: Accessing physical memory

F. Setup physical memory access

Since the addresses used in the paging structures are physical addresses, we need a component with the ability to read data from memory using physical addresses. This component will support the process of traversing paging structures such as PML4E, PTE, etc.

There are several methods to read data from memory using physical addresses. For example:

1) Using the `MmCopyMemory` function with `MM_COPY_MEMORY_PHYSICAL` flag
2) Using the `MmGetVirtualForPhysical` function to convert a physical address to a virtual address.
3) Mapping a section of the physical address to the kernel virtual address space using `ZwMapViewOfSection`.
4) Mapping a section of the physical address to the kernel virtual address space using `MmMapIoSpace`.

As a foundational approach, we recommend following option 3. Specifically, we propose mapping the desired physical page into the virtual address space using the `ZwMapViewOfSection`. Once this is done, we can read the data on that page using standard Windows APIs for memory access.

G. Detecting injected pages

We have implemented two scanning modes to detect injected pages: RWX scan and Private executable page scan. In RWX scanning mode, we search for pages in memory that have both write and execute privileges, similar to other VAD-based detecting techniques. As mentioned in section IV-F1, a page with both write and executable protection is rarely seen in a normal process in Windows. On the other hand, RWX pages are always needed in malware code injection techniques. Therefore, this is a very good signature to help us determine code injection malware. The second mode, Private executable page scan, is based on the idea that injected pages are always in private memory. Normally, private pages are used to store process data and aren't executable. Therefore, a private page with executable privilege is suspicious and has a high chance of containing malicious code.

VI. Implementation

To support our methodology, we've developed a tool called LPUS [5], [6]. This tool is capable of executing Pool Tag Quick Scanning, inspecting multiple kernel global variables, and performing several code injection detection methods.

LPUS consists of a kernel driver written in C [6] and a user-space program written in Rust [5]. Communication between these components is facilitated through file-based IOCTL.

The entire tool is encapsulated within a single binary file. Upon startup, it extracts the kernel driver stored within to a designated location and configures the registry settings for driver loading. Simultaneously, the user-space program determines the kernel version to download the appropriate PDB file and initiates the parsing process upon completion of the download.

After everything is set up, the tool can perform Live Memory Forensics without extraction of RAM.

LPUS was successfully executed on Windows 10H2. By simulating multiple code injection methods and achieving successful detection, we have validated that our proposed method is functioning as intended.

A. Limitations

Our approach comes with certain limitations and drawbacks that users should consider. Here, we highlight some significant drawbacks that may be encountered, and it's advisable to fall back to alternative methods when possible.

1) Internet Dependency: Our method relies on the availability of internet access for PDB file downloads during the analysis. This requirement can be problematic in environments where internet access is unavailable or intentionally isolated, such as during an ongoing incident response in a secure or isolated network setting. In such cases, it may not be feasible to rely on this method.

It's worth noting that major Windows versions may not always introduce substantial changes due to *delta updates* and *express updates* [7]. This observation may indeed provide an opportunity to store multiple PDB files of different major Windows versions. However, it's important to keep in mind that the extent of changes between major versions can still vary, and not all updates will have uniform impacts. Due to limited research on this assertion [11], [13], the effectiveness or potency of this claim remains uncertain.

2) Unsafe memory access: Our approach allows for extensive access to the kernel-space memory, as well as the ability to read and write in memory. However, it's important to emphasize that memory access is not safeguarded, and any erroneous or unauthorized memory operations have the potential to trigger errors, leading to system instability or even a kernel panic (exception) that forces a complete machine shutdown.

3) Insecure kernel access: Our method employs a communication mechanism between user and kernel through file-based IOCTL, which is accessible by any process. This presents a security vulnerability as a malicious process could potentially send IOCTL commands and read system data freely. To mitigate this risk, an authorization mechanism should be implemented to prevent unauthorized access and enhance security.

4) Deployment security: Deploying the solution necessitates that the kernel driver be signed using a Microsoft certificate. However, because of the extensive capabilities of this kernel driver, there is the potential for it to be exploited for malicious purposes. To prevent this undesirable outcome, the to-be-deployed kernel driver should undergo rigorous security measures to ensure its integrity and prevent misuse. Security hardening and stringent access controls are crucial to safeguard the system from potential threats.

5) Efficiency of Live Forensics: There are various concerns [1], [2], [4], [50] about the efficiency of Live Forensics, but it's important to note that some of these research may be outdated, as physical memory has undergone significant upgrades over time. Furthermore, Live Forensics had not been previously integrated with Memory Forensics or worked directly on memory as our proposed method does. As a result, claiming that our method's efficiency is higher or lower than current state-of-the-art forensic practices is unconfirmed, and this should be the subject of thorough and up-to-date research.

B. Extensions

Our method has opened up a new paradigm in Live Forensics, enabling Memory Forensics to be conducted directly in memory. The fundamental concept involves precise manipulation of the kernel using PDB files as guiding directives. Building on this foundation, we envision the potential for further extension into a more sophisticated and versatile piece of application. In this section, we present a list of applications that could benefit from enhancements made possible through our method.

1) Memory Extraction: Most memory extraction tools use a kernel driver and complete the extraction without any metadata. We can build on this limitation by providing the kernel base address, the CR3 register, and the kernel version. These information can be easily collected if following our method.

Kernel base address can be fetched as soon as the kernel driver starts, the kernel version can be fetched from the kernel executable when performing extraction. Only the CR3 register is hard to collect since these require access to global variables, while this requires the PDB files. If PDB files are available, then destructing the global variables. Otherwise this register value can be inspected later.

By providing this metadata, we enhance the comprehensiveness and value of memory extractions, making them more informative and useful for forensic analysis and security research.

2) Scripting Engine: The ability to attach to the kernel and inspect global variables, along with their internal structure values, is a valuable asset for security researchers and kernel driver developers. A similar program is WinDBG [37], developed by Microsoft, which also provides these capabilities. However, WinDBG typically requires users to access the target machine remotely from another machine for such inspections. In contrast, our method allows users to directly perform global variable inspections on the current machine.

To facilitate these capabilities, a scripting engine similar to the one found in WinDBG can be employed, enabling users to execute arbitrary commands and conduct in-depth kernel analysis. It's important to note that our method, while powerful for memory analysis and global variable inspections, does have limitations and cannot support full debugging functionality, including features like breakpoints and direct modifications of registers. Nonetheless, it provides valuable capabilities for research, analysis, and development tasks.

3) Anti-Cheat Engine: An Anti-Cheat Engine is a suite of software and techniques designed to detect and prevent cheating in games. Game cheaters often employ two primary techniques: static patching and dynamic patching. Static patching involves making significant modifications to the binary control flow to render certain checks irrelevant, while dynamic patching involves using methods akin to those used in malware to modify the process memory or inject code that alters the control flow or hooks into the game logic. Dynamic patching is more common as it typically requires less time investment.

One common method used in dynamic patching is game hooking, which is an alternate term in the game cracking community for process injection. As process injection can be identified through Memory Forensics, our method is certainly capable of discovering these game cheats in real-time, making it a valuable tool for detecting and mitigating cheating in (online) games.

4) Security Systems: The use of our method presents an opportunity to enhance various security systems, including Anti-Virus and Endpoint Detection and Response solutions. Integrating Live Memory Forensics into these systems is a reasonable approach in enhancing their capabilities, as it can provide real-time insights into system behavior and threats. However, one of the historical challenges in adopting Live Memory Forensics has been the overhead associated with RAM extraction, making it less common in these systems.

Our method, which eliminates the need for cumbersome memory extraction, can make it significantly easier to integrate Live Forensics into these security systems. This integration can lead to more effective and proactive threat detection, providing security solutions with a valuable tool to respond to sophisticated and evolving threats in real-time.

VII. Conclusion

This paper introduces a novel methodology for Live Forensics using Volatile Memory Forensics, often referred to as Live Memory Forensics. In contrast to the current state of Live Memory Forensics, which typically requires the extraction of physical memory and extensive brute-force searching to initiate the analysis, our method enables analysis to be conducted directly in memory without the need for RAM extraction.

The proposed method has been implemented in a prototype tool capable of performing two common tasks in Memory Forensics: Pool Tag Scanning and the detection of process injection. While this method holds great promise, there are possibilities for further enhancement. Due to time constraints, we can only briefly discuss these potential enhancements without presenting concrete evidence at this stage.

References

[1] Ahmed Alasiri. Comparative analysis of operational malware dynamic link library (dll) injection live response vs. memory image. 2012.

[2] Amer Aljaedi, Dale Lindskog, Pavol Zavarsky, Ron Ruhl, and Fares Almari. Comparative analysis of volatile memory forensics: live response vs. memory imaging. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1253–1258. IEEE, 2011.

[3] Andrea Allievi, Alex Ionescu, Mark E Russinovich, and David A Solomon. *Windows internals, part 2*. Microsoft Press, 2021.

[4] Muteb Alzaidi. The study of ssdt hook through comparative analysis between live response and memory image. 2012.

[5] Vo Van Tien Dung Anh Khoa Nguyen. Lpus. URL: https://github.com/nganhkhoa/lpus.

[6] Vo Van Tien Dung Anh Khoa Nguyen. Lpus driver. URL: https://github.com/nganhkhoa/lpus-driver.

[7] Mike Benson. Windows 10 quality updates explained and the end of delta updates, 2018. URL: https://techcommunity.microsoft.com/t5/windows-it-pro-blog/windows-10-quality-updates-explained-and-the-end-of-delta/ba-p/214426/.

[8] Frank Block. Ptemalfind. URL: https://github.com/f-block/volatility-plugins/blob/main/ptemalfind.py.

[9] Frank Block and Andreas Dewald. Windows memory forensics: Detecting (un) intentionally hidden injected code by examining page table entries. *Digital Investigation*, 29:S3–S12, 2019.

[10] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.

[11] Michael I Cohen. Characterization of the windows kernel version variability for accurate memory analysis. *Digital Investigation*, 12:S38–S49, 2015.

[12] A Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. *2006*, 2006.

[13] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, 2009.

[14] Aleksandra Doniec. Hollowhunter. URL: https://github.com/hasherezade/hollows_hunter.

[15] Aleksandra Doniec. mal_unpack. URL: https://github.com/hasherezade/mal_unpack.

[16] Aleksandra Doniec. Pe-sieve. URL: https://github.com/hasherezade/pe-sieve.

[17] Fireeye. Memoryze. URL: https://fireeye.market/apps/211368.

[18] Volatility Foundation. Malfind. URL: https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#malfind.

[19] Volatility Foundation. Volatility. URL: https://github.com/volatilityfoundation/volatility/.

[20] Volatility Foundation. Volatility3. URL: https://github.com/volatilityfoundation/volatility3/.

[21] Wireshark Foundation. Wireshark. URL: https://www.wireshark.org/.

[22] gleeda. memtriage. URL: https://github.com/gleeda/memtriage.

[23] Google. Rekall. URL: https://github.com/google/rekall/.

[24] Jim Hogg. What's inside a pdb file?, 2016. URL: https://devblogs.microsoft.com/cppblog/whats-inside-a-pdb-file/.

[25] Greg Hoglund. A *real* nt rootkit, patching the nt kernel. URL: http://phrack.org/issues/55/5.html/.

[26] Qiang Hua and Yang Zhang. Detecting malware and rootkit via memory forensics. In *2015 International Conference on Computer Science and Mechanical Automation (CSMA)*, pages 92–96. IEEE, 2015.

[27] Intel. Intel® 64 and ia-32 architectures software developer's manual.

[28] Ilker Kara. Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges. *Expert Systems with Applications*, 214:119133, 2023.

[29] Amit Klein and Itzik Kotler. Windows process injection in 2019. *Black Hat USA*, 2019, 2019.

[30] Jesse Kornblum. Windows memory forensics and direct kernel object manipulation. *Retrieved from: Jessekornblum.com/presentations/dodcc11-2. pdf*, 2011.

[31] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[32] Wen Jia Liu. Process hacker. URL: https://processhacker.sourceforge.io/.

[33] LLVM. The pdb file format. URL: https://llvm.org/docs/PDB/index.html/.

[34] Microsoft. Access tokens. URL: https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens/.

[35] Microsoft. microsoft-pdb. URL: https://github.com/microsoft/microsoft-pdb/.

[36] Microsoft. `ExAllocatePoolWithTag`. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepoolwithtag/.

[37] Microsoft. Windbg. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/windbg-overview.

[38] Monnappa. Hollowfind. URL: https://github.com/monnappa22/HollowFind.

[39] Monnappa. Psinfo. URL: https://github.com/monnappa22/Psinfo.

[40] Vergilius Project. `_eprocess`. URL: https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_EPROCESS.

[41] Vergilius Project. `_ethread`. URL: https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_ETHREAD.

[42] Vergilius Project. `_mi_system_information`. URL: https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_MI_SYSTEM_INFORMATION.

[43] Vergilius Project. `_mmvad`. URL: https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_MMVAD.

[44] Mark Russinovich. Autoruns. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/autoruns/.

[45] Mark Russinovich. Process explorer. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer/.

[46] Mark Russinovich. Process monitor. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/procmon/.

[47] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. *digital investigation*, 3:10–16, 2006.

[48] Joe T Sylve, Vico Marziale, and Golden G Richard III. Pool tag quick scanning for windows memory analysis. *Digital Investigation*, 16:S25–S32, 2016.

[49] New Security Ventures (NSV) team at Microsoft Research. Project freta. URL: https://learn.microsoft.com/en-us/security/research/project-freta/.

[50] Cal Waits, Joseph Ayo Akinyele, Richard Nolan, and Larry Rogers. Computer forensics: results of live response inquiry vs. memory image analysis. *CERT program, CMU/SEI-2008-TN-017*, 2008.

[51] AAron Walters and Nick L Petroni. Volatools: Integrating volatile memory into the digital investigation process. *Black Hat DC*, 2007:1–18, 2007.

[52] Inc Winsider Seminars & Solutions. System informer. URL: https://systeminformer.sourceforge.io.

[53] Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 2017.