Quantifying Paging on Recoverable Data from Windows User-Space Modules

Miguel Martín-Pérez and Ricardo J. Rodríguez(✉)

Department of Computer Science and Systems Engineering,
University of Zaragoza, Zaragoza, Spain
{miguelmartinperez,rjrodriguez}@unizar.es

Abstract. Memory forensic analysis enables a forensic examiner to retrieve evidence of a security incident, such as encryption keys, or analyze malware that resides solely in memory. During this process, the current state of system memory is acquired and saved to a file denoted as *memory dump*, which is then analyzed with dedicated software for evidence. Although a memory dump contains large amounts of data for analysis, its content can be inaccurate and incomplete due to how an operating system's memory management subsystem works: page swapping, on-demand paging, or page smearing are some of the problems that can affect the data that resides in memory. In this paper, we evaluate how these issues affect user-mode modules by measuring the ratio of modules that reside in memory on a Windows 10 system under different memory workloads. On Windows, a module represents an image (that is, an executable, shared dynamic library, or driver) that was loaded as part of the kernel or a user-mode process. We show that this ratio is particularly low in shared dynamic library modules, as opposed to executable modules. We also discuss the issues of memory forensics that can affect scanning for malicious evidences in particular. Additionally, we have developed a Volatility plugin, dubbed `residentmem`, which helps forensic analysts obtain paging information from a memory dump for each process running at the time of acquisition, providing them with information on the amount of data that cannot be properly analyzed.

Keywords: Digital forensics · Memory forensics · Windows modules · Paging · Malware

1 Introduction

Incident response aims to find out what happened in a security incident and, more importantly, preserve *evidence* related to the incident that can then be used to take legal action, trying to respond to the known 6 W (*what*, *who*, *why*, *how*, *when* and *where*) [12]. An important activity performed during the incident response process is digital forensics, which in the event of a computer incident is performed on the computers or network where the incident occurred [20]. While

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
P. Gladyshev et al. (Eds.): ICDF2C 2021, LNICST 441, pp. 1–19, 2022.
https://doi.org/10.1007/978-3-031-06365-7_1

computer forensics attempts to find evidence on computers and digital storage media, network forensics deals with the acquisition and analysis of network traffic.

Computer forensics is divided into different branches, depending on the digital evidence that is analyzed. In particular, this paper focuses on memory forensics, which deals with the retrieval of digital evidence from computer memory rather than from computer storage media, as disk forensics does. Memory forensics is useful in scenarios where encrypted or remote storage is used, improving on traditional forensic techniques more focused on non-volatile storage [28]. For instance, memory forensic analysis enables a forensic examiner to retrieve encryption keys or analyze malicious software (*malware*) that resides solely in RAM. In addition, triage in memory forensic is faster since the amount of data to be analyzed is less than in disk forensics.

The memory of a computer system can be acquired by different means, which are highly dependent on the underlying operating system and the hardware architecture of the system. A recent and comprehensive study of the latest memory acquisition techniques is provided in [26]. The memory acquisition process retrieves the current state of the system, reflected as it is in memory, and dumps it into a snapshot file (known as *memory dump*). This file is then taken offsite and analyzed with dedicated software for evidence (such as Volatility [57], Rekall [44], or Helix3, to name a few).

A memory dump contains data relevant to the analysis of the incident. In forensic terminology, these items are called *memory artifacts* and include items such as running processes, open files, logged in users, or open network connections at the time of memory acquisition. Additionally, a memory dump can also contain recently used items that have been freed but not yet zeroed, such as residual IP packets, Ethernet frames, or other associated data structures [6]. Many of these artifacts are more likely to reside in memory than on disk, due to their volatile nature.

Malware is currently one of the biggest security threats for both businesses and home users. Malware analysis is the process of determining whether a software program performs harmful activity on a system. The malware analysis methodology comprises two steps [48]: (1) *static analysis*, where the binary code of the suspicious software program is analyzed without executing it; and (2) *dynamic analysis*, where the program is executed and its interaction with operating system resources (other processes, persistence mechanisms [54], file system and network) is monitored.

As the rootkit paradox states [25], whenever code wants to run on a system, it must be visible to the system in some way. Therefore, running malware will leave traces of its nefarious activity, which are then useful for finding out what happened during a security incident. These traces, as digital artifacts, can reside in memory or on disk. For instance, fileless malware exists exclusively as a memory-based artifact, unlike file-based malware. According to recent industry security reports, this type of malware is increasing every year, especially taking advantage of PowerShell to launch attacks [5]. Fileless attacks grew 265% in

2019 [4], and Symantec detected a total of 115, 000 malicious PowerShell scripts (on average) each month during that year [40]. In this regard, there is more likely evidence of malicious activity from sophisticated malware or fileless malware in memory than on disk. In this paper, we focus on the Windows operating system (Windows for short), as at the time of this writing is still the most predominant target of malware attacks [2].

On Windows, an executable, shared dynamic library, or driver file that was loaded as part of the kernel or a user-mode process is named *image*, while the file is named *image file*. Finally, an image as well as a process are internally represented by a *module* [33]. In what follows, we adhere to this terminology.

In the event of a malware-related security incident, it is likely that malicious modules exist in a memory dump, as the malicious image file and its dependent images were loaded into memory for execution. However, *to what extent can we trust the contents of a memory dump?* This content may be inaccurate due to the way the memory management subsystem works. This inaccuracy problem, called *page smearing*, is particularly visible when we acquire memory on a live system: while the operating system is running, the references to memory in the running processes are constantly updated and, therefore, memory inconsistencies can occur since the acquisition process is usually carried out non-atomically [41].

Additionally, some optimization methods performed by the memory management subsystem can also affect the data in a memory dump. For example, *page swapping*, which refers to copying pages from the process's virtual address space to the swap device (which is typically non-volatile secondary memory storage), or vice versa. In the same way, *on-demand paging* (also known as deferred paging) delays loading a page from disk until it is absolutely necessary. Both methods affect the contents of a memory dump since parts of memory are likely to be swapped or not loaded at the time of acquisition. Therefore, false negative results are likely to occur when looking for evidence of malware exclusively in memory.

The contribution of this work is twofold. First, a detailed analysis of how these paging issues affect the user-mode modules that reside in memory on a Windows system with different memory workloads. In particular, we studied them on a Windows 10 64-bit system (build 19041) as at the time of this writing this is the predominant version on the market worldwide, with the 78% share [19]. Second, a thorough discussion on the issues to detect malware artifacts in memory forensics. As a side product of our research, we have developed a Volatility plugin, dubbed `residentmem`, which allows us to extract the number of resident pages (that is, in memory) of each image and each process within a memory dump. This tool therefore provides forensic analysts with information on the amount of binary data that cannot be analyzed correctly.

The outline of this paper is as follows. To provide a better understanding of this paper, we first give some background on the Windows memory subsystem in Sect. 2. Then we review the related work in Sect. 3. Section 4 presents the experiments performed to quantify the effect of paging on the Windows system under study. Next, we discuss how these issues can affect the analysis of malware artifacts in Sect. 5. Finally, Sect. 6 concludes the paper and highlights future directions.

2 Background

A Windows process has a private memory address space that cannot be accessed by other processes and cannot be exceeded. The *virtual address space* of a process (also known as the *page table*) defines the set of virtual addresses available to that particular process. Page tables are only accessible in kernel-mode, and therefore a user-space process cannot modify its own address space layout.

By default, the size of the virtual address space of a 32-bit Windows process is 2 GiB (before Windows 8). This size can be expanded to 3 GiB (or 4 GiB on 64-bit versions of Windows) in certain configurations [60]. The size of the virtual address space of a process in Windows 8.1 64-bit (and later) is 128 TiB, although the maximum amount of physical memory supported by Windows at the time of this writing is less than 24 TiB.

The memory unit by which Windows manages memory is called *memory page* [23]. A memory page defines a contiguous block of virtual memory of fixed length. Page sizes can be small or large. The small page size is 4 KiB, while the large page size ranges from 2 MiB (on x86 and x64 architectures) to 4 MiB (on ARM) [60]. The relationship between virtual memory and physical memory is made through the *page table entries* (PTE), which map a page of process virtual memory to a page of physical memory.

Since the virtual address space of the process can be larger than the physical memory on the machine, the Windows memory subsystem must maintain these page table entries to ensure that when a thread in the context of a process reads/write to addresses in its virtual memory space, they refer to the correct physical addresses [34]. Likewise, when the memory required by running processes exceeds the available physical memory, it also sends some pages to disk that are later retrieved by returning them to physical memory when necessary.

A page of a virtual address space of a process can be in different states [35]: *free*, when the page has never been used or is no longer used (initial state of all pages of a process). A free page is not accessible for the process but can be reserved, committed or simultaneously reserved and committed; *reserved*, when the process has reserved some memory pages within its virtual address space for future use. These pages are not yet accessible to the process, but their address range cannot be used by other memory allocation functions; *committed*, when the page has been allocated from RAM and paging files on disk, ready to be used by the process. Committed pages are also known as private pages, as they cannot be shared with other processes, unlike shareable pages, which are shared but can only be in use by a single process at a time. When a process accesses committed or shareable pages, the memory page is said to have been "touched". This is when the Windows memory manager finally allocates a page of physical memory via a page table entry.

3 Related Work

Forensic analysis of user-space memory has been approached in different ways. The work in [59] introduces an approach based on Virtual Address Descriptors (VAD) [14] that identifies all user allocations and then determines their purpose using kernel and user-space metadata sources. Based on an extensive analysis of the Windows XP SP3 32-bit and Windows 7 SP1 32-bit operating systems, the authors created two Volatility plugins to describe the content of allocations within user-space memory and to verify whether a virtual address of a process not described by a VAD is assigned to a page of physical memory. Paging is an important issue for this approach, as some metadata sources can be paged to disk, thus preventing extraction of their related metadata. Our work complements this approach by providing insight into the internals of the Windows memory manager with regard to paging.

A utility dubbed `PageDumper` that captures traces of attacks based on runtime memory protection tampering in the Linux operating system is proposed in [42]. Implemented as a kernel module, it helps analyze kernel and user-process address spaces, parsing page table entries in both kernel and user contexts. Rather, we focus on Windows and a post-mortem analysis of a complete memory dump. In any case, `PageDumper` can be a good complement to our plugin when analyzing a Linux operating system memory dump. More research is needed to extend our plugin to Linux and integrate it with the output of `PageDumper`.

With regard to malware detection in memory forensics, most works use Virtual Machine Introspection (VMI) techniques to avoid inaccuracy due to memory acquisition on live systems. The fundamental papers in this area are [22,38]. In [16], the authors demonstrated that the memory forensic community can develop tools using VMI and proceed much more quickly with memory analysis. In this regard, in [53] the authors introduce a VMI-based system on top of Xen that can detect malware in virtual machines using Volatility by comparing memory dumps acquired before and after executing a suspicious image file.

Other work focuses on using memory forensics as the basis for malware analysis. The differences between applying YARA signatures to disk or in-memory files and how these can be improved to effectively search for malware in memory are discussed in [13]. YARA is a very popular open-source and multi-platform tool to identify and classify malware samples. In [1], the effectiveness of different machine-learning classifiers is evaluated using information from VADs, registry hives, and other internal process structures such as `EPROCESS`. However, the software used to recover these memory artifacts is unclear. The work in [18] uses the prevalence of certain dynamic link libraries in processes contained in a memory dump as a characteristic of malicious behavior. The work in [37] presents a machine learning model that uses some features (such as registry keys, imported shared libraries, and called operating system functions) extracted from the reports provided by Cuckoo Sandbox to obtain information about a memory dump. Similarly, the work in [43] presents an analysis system composed of Cuckoo Sandbox and Volatility in which, as a final analysis step, the results obtained are compared with the results of VirusTotal. The work in [10] introduces `hooktracer_messagehooks`, a Volatility plugin that helps analyze hooks in a Windows memory dump to determine if they are associated with a malicious keylogger or with benign software. Finally, the authors introduce a system in [8] that first uses the Procdump tool, a Microsoft command line tool, to dump processes from memory in Windows 10 version 1903 systems and then converts them into RGB images for classification using machine learning algorithms.

With regard to malware focused on hiding its presence, in [7] an approach is presented to discover executable pages despite the use of stealthy techniques so that they are not reported by current detection tools. The authors implement it in a plugin for the `Rekall` memory forensic framework and evaluate it against own implementations of different stealthy techniques, as well as against real-world malware samples. Instead of VAD, this approach relies on PTE that are listed through paging structures to avoid certain (advanced) stealthy techniques. However, as before this approach does not work if the page tables are paged and the paging file is not provided. A similar work is [3], which introduces different techniques that malware can adopt to hide its presence using GPU memory. This work is very interesting, since the malware that resides in that memory cannot leave a trace in the physical memory. The analysis of another type of memory instead of the physical memory, though, is beyond the scope of this article.

To the best of our knowledge, we are the first to study in detail and quantify the effect of paging in Windows user-space modules. Our work is complementary to all those presented, as it provides information on how paging works in Windows, which is a problem for detecting malware by memory forensic analysis when paging files are not used. Unfortunately, this lack of paging files is common as Volatility does not support analyzing a dump alongside its paging file, despite being the most widely used and powerful memory forensic framework currently available.

4 Quantification and Characterization of the Windows Paging Mechanism

In this section, we first describe the experiments carried out to quantify and characterize the effect of paging on the Windows system under study and then discuss the results.

4.1 Description of Experiments

As an experimental scenario, we use a virtual machine with a base installation of Windows 10 64-bit version 19041 running on the VirtualBox 6.1.18 hypervisor with default paravirtualization and large paging disabled. Note that in the default paravirtualization, VirtualBox only supports paravirtualized clocks, APIC frequency reporting, guest debugging, guest crash reporting, and relaxed timer checks. Therefore, the behavior of the guest memory management unit is not affected by paravirtualization. Furthermore, we use a virtual machine to avoid the problem of page smearing, since we are interested in quantifying paging, which is affected by page swapping and on-demand paging. We consider two configurations of physical memory, 4 GiB and 8 GiB, with an Intel Core i7-6700 3.40 GHz dual-core processor. The Internet has been disconnected after updating the machines.

As memory workloads, we consider 25%, 50%, 75%, 100%, 125%, and 150% of the total physical memory. We have developed a simple C tool to allocate the amount of memory needed to reach the specified percentage of memory used (that is, the tool allocates between 1 GiB and 6 GiB and between 2 GiB and 12 GiB, for the memory configurations of 4 GiB and 8 GiB, respectively). In particular, the tool allocates memory and writes a random byte every 4KiB to ensure that pages are constantly used and avoid their paging as much as possible. Note that this tool will consume a large chunk of memory and will leave less space for the pages of other user-space processes, regardless of the use of these pages (recall that both anonymous mappings and file mappings are backed by the system paging file [11]).

Under these conditions, the system memory has been acquired at various runtimes for each memory workload. First, we initialize the virtual machine and wait 5 min for the machine to reach a stable state. Immediately after, we pause the virtual machine and acquire the *initial* dump. Then we launch the memory allocator tool explained above and dump the memory every 15 s for one minute, pausing and resuming the execution of the virtual machine before and after memory acquisition. After the first minute, we continue to capture the memory every minute for 4 more minutes, also pausing and resuming the execution of the virtual machine between memory acquisitions. These memory dumps make up the *first observation moment*. We then stop the memory allocator process, pause the execution of the virtual machine, and dump the memory using the same pattern: every 15 s for the first minute and every 1 min for the next 4 min. These memory dumps are part of the *second observation moment*. Finally, we shut down and reboot the virtual machine to restart the dump process with another memory workload.

For each memory dump, we get the number of recoverable modules and how many resident pages are in each module. The process of memory acquisition and computation of recoverable data has been replicated 10 times in order to increase the reliability of the evaluation. We finally took into account the average of the 10 independent repeats for each recoverable module. To help us obtain the recoverable data of modules, we have implemented a tool, dubbed `residentmem`, as a Volatility plugin released under GNU/GPL version 3 license and publicly available at https://github.com/reverseame/residentmem. The plugin iterates through the processes contained in the memory dump and, for each process, it checks the memory pages assigned to each module associated with that process through internal Volatility structures. As input, the plugin needs a memory dump. As output, it returns a list of the recoverable modules with the resident pages and the total pages of each module, the process to which the module belongs, the path where is stored in the file system, and other information of interest related to the module (such as its version, base address, and its process identifier, among other information). In addition, it allows obtaining the specific list of physical pages for each resident page of a recoverable module.

The plugin analysis workflow is as follows. We first obtain the list of processes that were running at the time of memory acquisition through Volatility's internal structures. We then iterate through this list, accessing its memory address space and validating with a 4096-byte step (the size of a small page) if each memory address is resident. This gives us the number of resident and total memory pages for each process and its related modules.

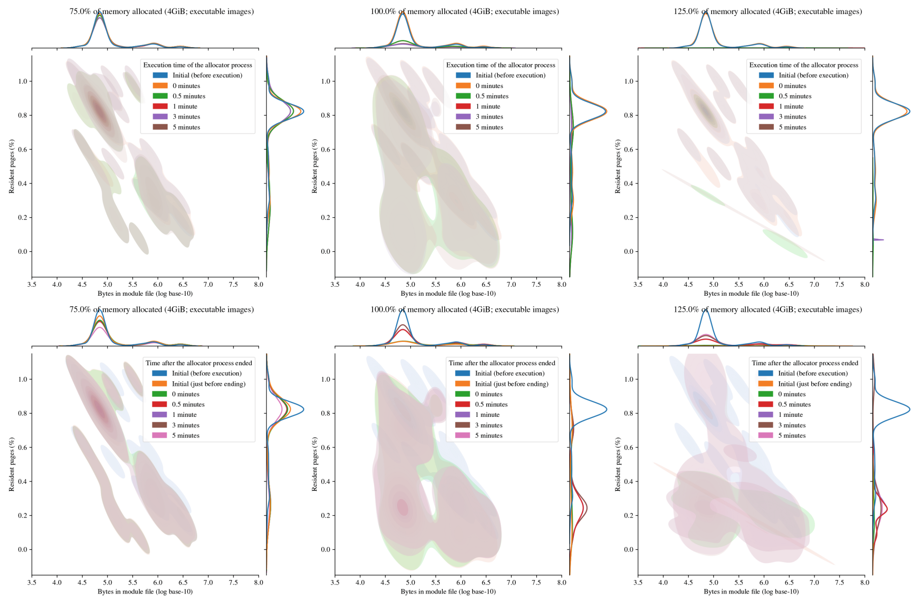
4.2 Discussion of Results

We only discuss the 75%, 100%, and 125% memory workloads because we have empirically observed that experiments below 75% and above 125% behave equal to 75% and 125%, respectively. Likewise, we do not plot all the instants of time because otherwise the graphs are overloaded and difficult to understand. For this reason, we only show the *Initial (before execution)* (before any interaction with the system), 0 min (just after interacting with the system, that is, starting or stopping the memory allocator tool); and *0.5*, *1*, *3*, and 5 min, a subset of the observed instants of time that faithfully represent the complete behavior.

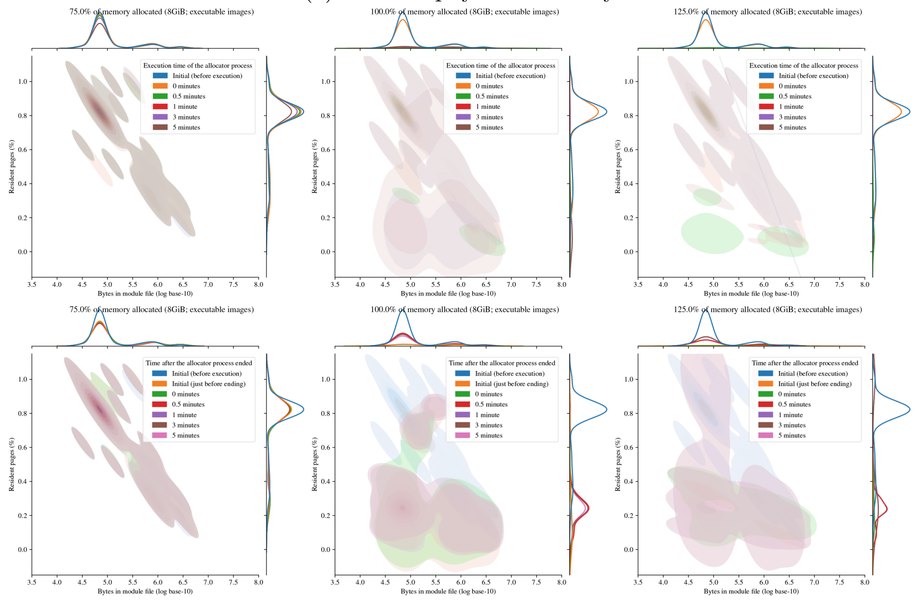
In addition, for the graphs relative to the second observation moment one more moment, *Initial (just before ending)*, has been incorporated to show how the system is just before interacting with it. This instant of time is actually the same as the 5 min instant of the first observation moment. The graphs of both observation moments show *Initial (before execution)* to have a common reference that allows comparing the results.

Modules of Executable Image Files. Figure 1 shows the resident pages of the recoverable modules of executable files under different memory workloads for 4 GiB and 8 GiB of physical memory (Figs. 1a and 1b, respectively), for both observation moments. Every plot shows the distributions of two variables (the size of a module file in log-base 10, on the x-axis, and the percentage of resident pages, on the y-axis) through color intensity. The darker the region, the more data is in that region. The subplots on the top and right of the main plots show a smoothed version of the frequencies of the size and resident pages data, revealing the distribution of resident pages and module file sizes.

Looking at the first observation moment, the initial conditions show that almost 80% of the executable module pages are resident in memory. With a memory workload of 75%, there are no significant changes to the resident pages because there is still enough free memory, regardless of the size of the physical memory. A slight reduction in recoverable modules is observed throughout the acquisition time points, which may be motivated by the paging of unused modules, while the resident pages remain constant. Note that the colored areas are mostly identical. With regard to 100% memory workload, in 0.5 min most modules are expelled and the number of resident pages for recoverable modules is drastically reduced. With the 125% memory workload, there is again a large



(a) 4GiB of physical memory



(b) 8GiB of physical memory

Fig. 1. Resident pages of recoverable executable modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively).

reduction in recoverable modules. In this case, the graph shows two well-defined areas, an area that contains the first two moments of time and another area that contains the remaining rime moments on a diagonal below 60% resident pages. The 8 GiB results show greater variability in this workload, as indicated by the larger color areas below the 60% diagonal.

With regard to the second observation moment, the results are the same as in the first observation moment with a memory workload of 75%. With the 100% and 125% memory workloads, it is observed that the modules progressively come back to memory, but the ratio of resident pages for recoverable modules never reaches a value greater than 25%. Significant increases in 0.5 min and in 3 min are observed for both memory configurations, although the number of recoverable modules is lower for 125% workload.

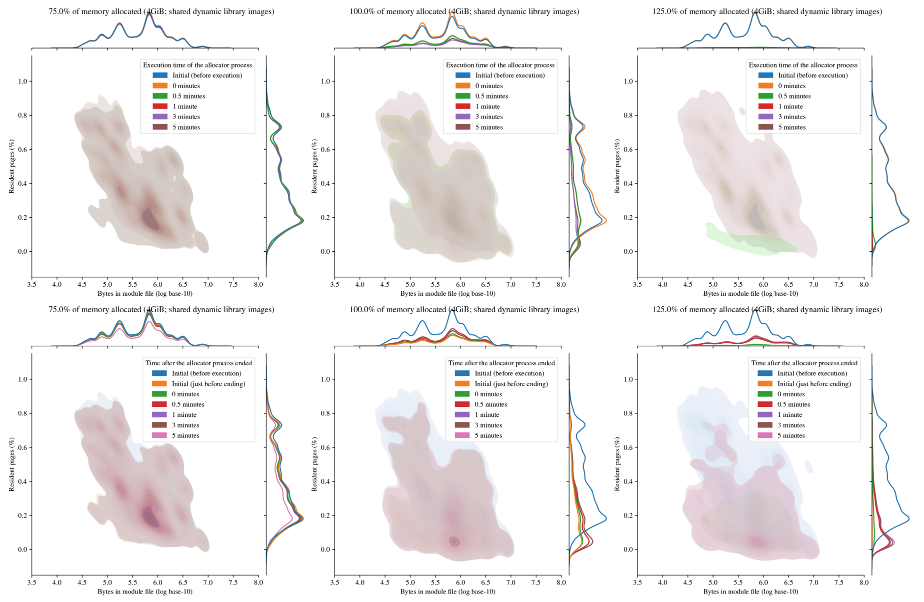
Modules of Shared Dynamic Library Image Files. Figure 2 shows the distribution graphs of resident pages of recoverable modules of shared dynamic libraries under different memory workloads for 4 GiB and 8 GiB of physical memory (Figs. 2a and 2b, respectively), for both observation moments. In this case, most modules only have 20% of their pages resident and are in the range 10^5 to 10^6 bytes. The maximum percentage of resident pages is 75%.

Regarding the first observation moment, no significant changes are observed with 75% of memory workload, similarly to the results of the previous type of module studied. A slight decrease in recoverable modules is observed, but with no effect on resident pages. With 100% of memory workload, the system starts expelling modules for any size in 0.5 min. The number of recoverable modules is reduced, but the distribution shape is similar in both memory configurations. A more aggressive expelling of modules is observed in 8 GiB of physical memory. In any case, most modules have only less than 5% of their pages resident. With 125%, the results are very similar. As before, there is a small colored area in the bottom of the graph, which indicates that the percentage of the resident pages is close to 5% again.

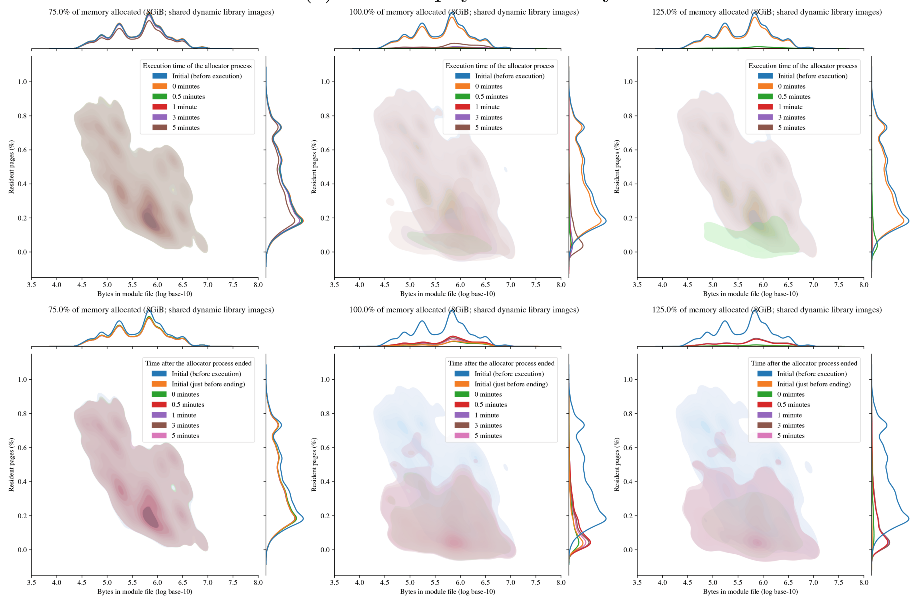
With regard to the second observation moment, the results with a memory workload of 75% are very similar to the results of the first observation moment. With 100% of memory workload, the number of recoverable modules is slowly increasing, but the percentage of resident pages for most modules is still close to 5%. Similar behavior is observed with the memory workload of 125%, where few modules return to memory but the percentage of resident pages remains close to 5% for most of them.

5 Detection of Malware in Memory Forensic Analysis: Current Problems and Solutions

This section details the problems that affect the detection of malware in memory forensic analysis, as well as some solutions to overcome them.



(a) 4GiB of physical memory



(b) 8GiB of physical memory

Fig. 2. Resident pages of recoverable shared dynamic library modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively).

Issue 1. Content Discrepancy Between an Image File and Its Module Image

At process creation, the Windows PE loader maps an image file to memory. As the default size of a memory page is 4 KiB, the image file will be fit into a set of multiple 4 KiB memory segments (see Sect. 2). Therefore, the representation of the image file in memory may contain zero-padding bytes due to memory alignment.

Additionally, the image may be dynamically relocated due to *Address Space Layout Randomization* (ASLR), which randomizes memory segment locations to thwart control-flow hijacking attacks [52]. If necessary, some bytes of the image will be appropriately relocated according to the new address layout. Also, PE sections such as the relocated section or the Authenticode signature [31] are not copied from image files to their corresponding images and hence these parts are not found in the memory representation of image files [55].

In short, *the (byte) data of an image will differ from its image file*. Therefore, we cannot rely on calculating cryptographic hashes such as MD5 or SHA-1 of an image to detect malicious signatures due to their avalanche effect property [58], which guarantees that the hashes of two similar, but not identical, inputs (e.g., inputs in which only one bit is flipped) produce radically different outputs.

Instead of cryptographic hashes, we can use approximate matching algorithms [9,21] to calculate similarity between Windows modules [29]. These algorithms provide similarity measures between two digital artifacts in the range $[0, 1]$. Although the use of these algorithms is feasible in the context of memory forensics, they present some security flaws that can be exploited by a sophisticated adversary to affect the hash calculation and provide perfect similarity or incomparable hashes [27,30]. Allow-list hash databases, such as those provided by NIST [39], can also be adapted to this diversity of hashing algorithms to assist incident response teams during incident analysis.

Issue 2. Lack of Data in a Memory Dump

As explained in Sect. 2, the Windows memory subsystem performs two optimization actions that have a clear impact on memory forensics: page swapping and on-demand paging. Remember that due to page swapping, unused pages are copied from memory to page files. In addition to occurring inadvertently, a sophisticated malware can trigger the paging process for as many pages as possible by calling the `SetProcessWorkingSetSize` API [36] as a way to hide its malicious code. Similarly, since an on-demand paging algorithm is used to know when to load pages into memory [60], parts of an image file will not load until strictly accessed.

In short, the data on a malicious image may be incomplete because page swapping and on-demand paging. If the swapped pages correspond to parts of the module header, the module will not be recoverable. Similarly, if the malicious code has not yet been loaded, it will remain undiscovered in the memory dump.

Also, as part of its life cycle, malware often uses a persistence strategy to ensure that it will persist on the system upon restart [49]. A feature commonly used by malware to achieve persistence is based on the Windows Registry [54], which is a hierarchical database divided into tree-like structures (called hives) that contain data critical to the operation of Windows and other applications [60].

As shown in Dolan-Gavitt's seminal work on memory forensics and the Windows Registry [15], some of these hives are volatile while others reside on disk (mapped to memory during Windows startup). Therefore, as a consequence of the paging issues highlighted above, unused portions of the Windows Registry may be on disk.

In summary, the Windows Registry of a memory dump cannot be treated as reliable evidence for detecting the persistence of malware, as the contents of the registry hives are incomplete. Additionally, the malware can still use other persistence methods that are not based on the Windows Registry and are also not detected by memory forensics [54].

Page swapping is really hard to beat when working with only memory artifacts. One possibility is to use disk forensics to first recover the page files and then use them together with the memory dump to complete the analysis. However, from Windows 8.1 onwards, the pages are swapped to disk in a compressed form, using Microsoft's Xpress algorithm [32], making content access a major challenge for current forensic tools. A recent publication in BlackHat USA 2019 presents a method to recover Windows 10 compressed pages and rebuild memory artifacts regardless of their storage location [47,51]. This method is a promising approach that has yet to be integrated with memory analysis frameworks like Volatility or Rekall.

A similar problem occurs with on-demand paging. In this regard, the best solution that we envision is to also combine memory forensics with disk forensics. Regarding the memory forensic analysis of persistence methods used by malware, we can again combine memory and disk forensics to extract the files on disk that represent the registry hives and get a complete and accurate overview of the Windows Registry.

In any case, we argue that these problems evidence that memory forensics cannot be seen as a single process but as a complementary process during an incident response activity.

Issue 3. Inaccuracy of a Memory Dump

When the memory is acquired in a live system, inaccuracies are highly likely to occur because memory is continually updated and acquired non-atomic. This page smearing problem is particularly relevant as kernel-space structures can be affected and lead to possible parsing errors, as some PTE can refer to wrong addresses (*pointer inconsistency*). Similarly, if a data object spans multiple physical pages, this fragmentation also affects the memory acquisition process, as the content may not be consistent over time (*fragment inconsistency*). These types of problems are commonly found in live memory acquisition on systems with large RAM or under heavy load [41].

Additionally, a rootkit or other sophisticated malware can detect when the memory acquisition is taking place to deliberately produce these inconsistencies through Direct Kernel Object Manipulation (DKOM) attacks, thus avoiding detection [45].

In summary, the data contained in a memory dump is likely to be inaccurate or unreliable if the memory acquisition process was performed on a live system. As possible solutions to avoid inaccuracies in the data, other acquisition techniques can be used [26] (for example, based on DMA or system management mode level). Invasive acquisition processes can also be used to acquire memory, such as cooling RAM modules (using liquid nitrogen or cold sprays) or forcing non-recoverable hardware errors to cause a *Blue Screen of Death* and generate a crash dump (which contains not only the RAM but also other data about the state of the system). These methods, however, can lead to the loss of important and unrecoverable data during the process.

In this regard, a recent contribution in memory forensics is the introduction of temporal dimension [41], which refers to the temporal consistency of the data stored in a memory dump. In [41], the authors present a Volatility plugin that accurately records time information while the memory dump is acquired. This time information is then used to construct a timeline that is displayed during memory dump analysis, suggesting to a forensic analyst the probability of inconsistencies and therefore taking appropriate action, such as additional data validation.

Issue 4. Stealthy Malware

Common signature-based methods of detecting malware artifacts in each process virtual address space in a memory dump can be problematic, as the page containing the searched signature may be non-resident at the time of memory acquisition. Additionally, sophisticated malware and advanced persistent threats can incorporate features to remain stealthy and hidden after infecting a computer [46].

For example, injected code can be hidden from forensic tools such as the Volatility plugin `malfind` because they are typically based on information provided by VAD, which are unfortunately an unreliable source of information. Remember that Windows uses VAD to store information about the memory regions of a process [14]. However, important information such as page permissions are not updated when changed. On this matter, a malware only needs to allocate memory initially with a protection without the write or execute permission and then add any of these permissions to the pages containing the malicious code [7,50]. In addition, as explained above, the malware can trigger the paging process for as many pages as possible by calling the `SetProcessWorkingSetSize` API to increase the likelihood of swapping pages that contain malicious code [36].

In short, stealthy malware can incorporate different features to remain stealthy and hidden on the infected system. To detect them, we can rely on malware signatures. A recent enhancement to `malfind` is the Volatility plugin `malscan`, which integrates `malfind` with Clam-AV antivirus signatures to reduce the number of false positives [56]. Robust kernel signatures can also be used to detect sophisticated attacks, such as DKOM attacks [17].

Another technique widely used by malware to stay hidden is process hollowing, which occurs when a malware creates a process in a suspended state, then unmaps its memory and replaces it with malicious code. In this regard, the Volatility plugin `impfuzzy` allows malware to be detected in a memory dump based on the hash values generated from the import functions of the processes [24].

6 Conclusions and Future Directions

Memory forensics relies on memory dump analysis to look for evidence of security incidents. However, the content of dumps can be inaccurate and incomplete due to page smearing, page swapping, and on-demand paging. This lack of reliable data becomes particularly relevant when looking for evidence of malware exclusively in memory, as false negative results are likely to occur. This issue also shows that memory dumps are unreliable partial sources of evidence.

In this paper, we have studied the effect of paging in Windows modules of the user-space processes. In particular, we have focused on Windows 10 build 19041. At different observation moments and under memory workloads, the number of recoverable modules and the number of resident pages have been quantified considering two sizes of physical memory. Our experimental results show that paging behave different depending on the type of module. At first, almost 80% of the executable module pages and 20% of the shared dynamic library module pages are resident. These values are drastically reduced when the operating system needs memory, as most modules are expelled from memory and then unrecoverable. Once the memory load is no longer high, the system recovers some of the paged modules but very slowly, never returning to the initial conditions (25% and 5% for executable and shared library image files, respectively).

Furthermore, we also describe the problems for malware detection in memory forensics, discussing some solutions to overcome them as well. These issues cause the data in an image to differ from its image file and to be incomplete, inaccurate, and unreliable. Additionally, malware can incorporate features to remain stealthy and hidden from memory forensics.

As future work, we plan to extend our study to other versions of Windows (such as server editions) and to better characterize paging distributions under different system workloads. In addition, we also intend to investigate new methods to detect stealthy malware in memory forensics and quantify the effects of paging on the kernel space.

Acknowledgments. This work was supported in part by the Spanish Ministry of Science, Innovation and Universities under grant MEDRESE-RTI2018-098543-B-I00, by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R), and by the University of Zaragoza and the *Fundación Ibercaja* under grant JIUZ-2020-TIC-08. The research of Miguel-Martín Pérez was also supported by Spanish National Cybersecurity Institute (INCIBE) "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad", grant numbers INCIBEC-2015-02486 and INCIBEI-2015-27300.

References

1. Aghaeikheirabady, M., Farshchi, S.M.R., Shirazi, H.: A new approach to malware detection by comparative analysis of data structures in a memory image. In: 2014 International Congress on Technology, Communication and Knowledge (ICTCK), pp. 1–4 (2014)
2. AV-TEST GmbH: AV-TEST Security Report 2019/20, August 2020. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf. Accessed 15 Apr 2021
3. Balzarotti, D., Di Pietro, R., Villani, A.: The impact of GPU-assisted malware on memory forensics: a case study. Digit. Investig. **14**, S16–S24 (2015). The Proceedings of the Fifteenth Annual DFRWS Conference
4. Barnes, E.: Mitigating the risks of fileless attacks. Comput. Fraud Secur. **2021**(4), 20 (2021)
5. Beek, C., et al.: McAfee Labs Threats Report, June 2018. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2018.pdf. Accessed 15 Apr 2021
6. Beverly, R., Garfinkel, S., Cardwell, G.: Forensic carving of network packets and associated data structures. Digit. Investig. **8**, S78–S89 (2011). The Proceedings of the Eleventh Annual DFRWS Conference
7. Block, F., Dewald, A.: Windows memory forensics: detecting (un)intentionally hidden injected code by examining page table entries. Digit. Investig. **29**, S3–S12 (2019)
8. Bozkir, A.S., Tahillioglu, E., Aydos, M., Kara, I.: Catch them alive: a malware detection approach through memory forensics, manifold learning and computer vision. Comput. Secur. **103**, 102166 (2021)
9. Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D.: Approximate Matching: Definition and Terminology. Techreport NIST Special Publication 800-168, National Institute of Standards and Technology, May 2014. https://doi.org/10.6028/NIST.SP.800-168
10. Case, A., et al.: Hooktracer: automatic detection and analysis of keystroke loggers using memory forensics. Comput. Secur. **96**, 101872 (2020)
11. Chen, R.: The source of much confusion: "backed by the system paging file", March 2013. https://devblogs.microsoft.com/oldnewthing/20130301-00/?p=5093. Accessed 24 Apr 2021
12. Cichonski, P., Millar, T., Grance, T., Scarfone, K.: Computer Security Incident Handling Guide. Techreport SP 800-61 Rev. 2, National Institute of Standards and Technology (NIST), September 2012. Special Publication (NIST SP)
13. Cohen, M.: Scanning memory with Yara. Digit. Investig. **20**, 34–43 (2017)
14. Dolan-Gavitt, B.: The VAD tree: a process-eye view of physical memory. Digit. Investig. **4**, 62–64 (2007)
15. Dolan-Gavitt, B.: Forensic analysis of the windows registry in memory. Digit. Investig. **5**, S26–S32 (2008). The Proceedings of the Eighth Annual DFRWS Conference

16. Dolan-Gavitt, B., Payne, B., Lee, W.: Leveraging forensic tools for virtual machine introspection. Technical report, Georgia Institute of Technology (2011)
17. Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.: Robust signatures for kernel data structures. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 566–577. Association for Computing Machinery, New York (2009)
18. Duan, Y., Fu, X., Luo, B., Wang, Z., Shi, J., Du, X.: Detective: automatically identify and analyze malware processes in forensic scenarios via DLLs. In: 2015 IEEE International Conference on Communications (ICC), pp. 5691–5696 (2015)
19. GlobalStats: Desktop Windows Version Market Share Worldwide, April 2021. https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide. Accessed 15 Apr 2021
20. Granc, T., Chevalier, S., Scarfone, K.K., Dang, H.: Guide to Integrating Forensic Techniques into Incident Response. Techreport 800-86, National Institute of Standards and Technology (NIST), August 2006. Special Publication (NIST SP)
21. Harichandran, V.S., Breitinger, F., Baggili, I.: Bytewise approximate matching: the good, the bad, and the unknown. J. Digit. Forensics Secur. Law **11**(2), 4 (2016)
22. Hay, B., Nance, K.: Forensics examination of volatile system data using virtual introspection. SIGOPS Oper. Syst. Rev. **42**(3), 74–82 (2008)
23. Huffman, C.: Process memory. In: Huffman, C. (ed.) Windows Performance Analysis Field Guide, pp. 93–127. Syngress, Boston (2015)
24. JPCERT/CC: A New Tool to Detect Known Malware from Memory Images - impfuzzy for Volatility, December 2016. https://blogs.jpcert.or.jp/en/2016/12/a-new-tool-to-d-d6bc.html. Accessed 23 Apr 201
25. Kornblum, J.D.: Exploiting the rootkit paradox with windows memory analysis. Int. J. Digit. EVid. **5**(1), 1–5 (2006)
26. Latzo, T., Palutke, R., Freiling, F.: A universal taxonomy and survey of forensic memory acquisition techniques. Digit. Investig. **28**, 56–69 (2019)
27. Lee, A., Atkison, T.: A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In: Proceedings of the SouthEast Conference, ACM SE 2017, pp. 18–25. Association for Computing Machinery, New York (2017)
28. Ligh, M.H., Case, A., Levy, J., Walter, A.: The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, Hoboken (2014)
29. Martín-Pérez, M., Rodríguez, R.J., Balzarotti, D.: Pre-processing memory dumps to improve similarity score of windows modules. Comput. Secur. **101**, 102119 (2021)
30. Martín-Pérez, M., Rodríguez, R.J., Breitinger, F.: Bringing order to approximate matching: classification and attacks on similarity digest algorithms. Forensic Sci. Int. Digit. Investig. **36**, 301120 (2021)
31. Microsoft Corporation: Windows Authenticode Portable Executable Signature Format, March 2008. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_pe.docx. Accessed 25 Sept 2019
32. Microsoft Corporation: [MS-XCA]: Xpress Compression Algorithm, March 2020. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-xca/a8b7cb0a-92a6-4187-a23b-5e14273b96f8. Accessed 15 Apr 2021
33. Microsoft Docs: Modules, May 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/modules. Accessed 15 Feb 2020
34. Microsoft Docs: Memory Management, May 2018. https://docs.microsoft.com/en-us/windows/win32/memory/memory-management. Accessed 15 Feb 2020

35. Microsoft Docs: Page State, May 2018. https://docs.microsoft.com/en-us/windows/win32/memory/page-state. Accessed 15 Feb 2020
36. Microsoft Docs: SetProcessWorkingSetSize function (winbase.h), May 2018. https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setprocessworkingsetsize. Accessed 25 Apr 2021
37. Mosli, R., Li, R., Yuan, B., Pan, Y.: Automated malware detection using artifacts in forensic memory images. In: 2016 IEEE Symposium on Technologies for Homeland Security (HST), pp. 1–6 (2016)
38. Nance, K., Bishop, M., Hay, B.: Investigating the implications of virtual machine introspection for digital forensics. In: 2009 International Conference on Availability, Reliability and Security, pp. 1024–1029 (2009)
39. National Institute of Standards and Technology: National Software Reference Library (NSRL) - Approximate Matching, July 2017. https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl/technical-information-0. Accessed 15 Apr 2021
40. O'Gorman, B., et al.: Internet Security Threat Report - volume 24, February 2019. https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf. Accessed 15 Apr 2021
41. Pagani, F., Fedorov, O., Balzarotti, D.: Introducing the temporal dimension to memory forensics. ACM Trans. Priv. Secur. **22**(2), 9:1–9:21 (2019)
42. Parida, T., Das, S.: PageDumper: a mechanism to collect page table manipulation information at run-time. Int. J. Inf. Secur. **20**(4), 603–619 (2020). https://doi.org/10.1007/s10207-020-00520-9
43. Rathnayaka, C., Jamdagni, A.: An efficient approach for advanced malware analysis using memory forensic technique. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 1145–1150 (2017)
44. Rekall: The Rekall memory forensic framework (2014). http://www.rekall-forensic.com/. Accessed 15 Apr 2021
45. Rodionov, A.M.E., Bratus, S.: Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats, 1st edn. No Starch Press Inc., San Francisco (2019)
46. Rudd, E.M., Rozsa, A., Günther, M., Boult, T.E.: A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. IEEE Commun. Surv. Tutor. **19**(2), 1145–1172 (2017)
47. Sardar, O., Andonov, D.: Paging All Windows Geeks - Finding Evil in Windows 10 Compressed Memory. BlackHat USA (2019)
48. Sikorski, M., Honig, A.: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press, San Francisco (2012)
49. Sood, A.K., Bansal, R., Enbody, R.J.: Cybercrime: dissecting the state of underground enterprise. IEEE Internet Comput. **17**(1), 60–68 (2013)
50. Srivastava, A., Jones, J.H.: Detecting code injection by cross-validating stack and VAD information in windows physical memory. In: 2017 IEEE Conference on Open Systems (ICOS), pp. 83–89, November 2017
51. Stancill, B., Vogl, S., Sardar, O.: Finding Evil in Windows 10 Compressed Memory, Part One: Volatility and Rekall Tools, July 2019. https://www.fireeye.com/blog/threat-research/2019/07/finding-evil-in-windows-ten-compressed-memory-part-one.html. Accessed 15 Apr 2021
52. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy, pp. 48–62, May 2013
53. Tien, C., Liao, J., Chang, S., Kuo, S.: Memory forensics using virtual machine introspection for malware analysis. In: 2017 IEEE Conference on Dependable and Secure Computing, pp. 518–519 (2017)

54. Uroz, D., Rodríguez, R.J.: Characteristics and detectability of windows auto-start extensibility points in memory forensics. Digit. Investig. **28**, S95–S104 (2019)
55. Uroz, D., Rodríguez, R.J.: On challenges in verifying trusted executable files in memory forensics. Forensic Sci. Int. Digit. Investig. **32**, 300917 (2020)
56. Uroz, D., Rodríguez, R.J.: malscan plugin (2020). https://github.com/reverseame/malscan. Accessed 23 Apr 2021
57. Walters, A., Petroni, N.: Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. BlackHat DC (2007)
58. Webster, A.F., Tavares, S.E.: On the design of S-boxes. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 523–534. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_41
59. White, A., Schatz, B., Foo, E.: Surveying the user space through user allocations. Digit. Investig. **9**, S3–S12 (2012). The Proceedings of the Twelfth Annual DFRWS Conference
60. Yosifovich, P., Ionescu, A., Russinovich, M.E., Solomon, D.A.: Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More, 7th edn. Microsoft Press, Redmond (2017)