Hiding Process Memory via Anti-Forensic Techniques

By:
Frank Block (Friedrich-Alexander Universität Erlangen-Nürnberg (FAU) and ERNW Research GmbH) and Ralph Palutke (Friedrich-Alexander Universität Erlangen-Nürnberg)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2020
July 20 - 24, 2020

Forensic Science International: Digital Investigation 33 (2020) 301012

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA

Hiding Process Memory Via Anti-Forensic Techniques

Ralph Palutke [a, **, 1], Frank Block [a, b, *, 1], Patrick Reichenberger [a], Dominik Stripeika [a]

[a] Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Germany
[b] ERNW Research GmbH, Heidelberg, Germany

ARTICLE INFO

Article history:

Keywords:
Memory subversion
Anti-forensics
Memory forensics
Hiding
Detection
Rekall
Volatility
Rootkits
Malware

ABSTRACT

Nowadays, security practitioners typically use memory acquisition or live forensics to detect and analyze sophisticated malware samples. Subsequently, malware authors began to incorporate anti-forensic techniques that subvert the analysis process by hiding malicious memory areas. Those techniques typically modify characteristics, such as access permissions, or place malicious data near legitimate one, in order to prevent the memory from being identified by analysis tools while still remaining accessible. With this paper, we present three novel methods that prevent malicious user space memory from appearing in analysis tools and additionally making the memory inaccessible from a security analysts perspective. Two of these techniques manipulate kernel structures, namely Page Table Entries and the structures responsible for managing user space memory regions, while the third one utilizes shared memory and hence does not require elevated privileges. As a proof of concept, we implemented all techniques for the Windows and Linux operating systems, and subsequently evaluated these with both, memory forensics and live analysis techniques. Furthermore, we discuss and evaluate several approaches to detect our subversion techniques and introduce two Rekall plugins that automate the detection of hidden memory for the shared memory scenario.

1. Introduction

Up to this day, malware still appears to be the empowering technology for digital cyber crimes. As a result, an ongoing arms race between malware authors and security participants evolved, increasing the sophistication level of both offensive and defensive approaches alike. Nowadays, modern malware mostly resides in volatile memory and can only be observed in a running state, potentially compromising the analysis process. Consequently, investigators began to use memory acquisition or forensic live analysis to detect and analyze potential threats. However, recent malware samples began to incorporate anti-forensic techniques to hinder analysis tools from acquiring meaningful results.

1.1. Motivation

Today, analysts depend on a variety of forensic tools to generate correct analysis results during forensic investigations. As most forensic tools rely on the integrity of certain kernel structures, they appear to be prone to the manipulation of these data. Adversaries gaining kernel level privileges can alter these structures and prevent analysis tools from gaining a correct view of the memory and as a result, hide malicious memory from analysis or acquisition. To be a step ahead of upcoming anti-forensic threats, we researched novel ways to outwit current forensic methods and developed detection approaches which can help tool vendors to gain a lead in the fight against malware.

In this work, we present three novel memory subversion techniques for both Windows and Linux that allow to hide malicious parts of a process' memory from being displayed by forensic tools. The first two techniques manipulate Page Table Entries (PTEs) and Memory Area Structures (MASs), certain kernel structures which provide meta information about a process' memory layout, and hence require kernel level access. The third method is based on shared memory and enables even unprivileged adversaries to hide memory regions from being analyzed. To support vendors of forensic software, we extensively discuss possible detection approaches and provide two Rekall plugins that are able to reveal the hidden regions for the shared memory scenario. Our evaluation states that malicious memory can indeed be hidden from the Rekall

* Corresponding author.
** Corresponding author.
E-mail addresses: ralph.palutke@fau.de (R. Palutke), fblock@ernw.de (F. Block).
[1] Both authors contributed equally to this paper and should be considered as joint first authors.

https://doi.org/10.1016/j.fsidi.2020.301012

and Volatility frameworks and thus shows the necessity to further improve these tools.

The prototypes of our subversion techniques and plugins have been tested and evaluated on both Windows and Linux running on the x86-64 architecture. For an easy reproducibility and verification of our results, all source codes and binary formats of our proof of concepts and plugins can be downloaded from our online repository (Reichenberger et al., 2020).

1.2. Contributions

The contributions of this paper are:

- Three novel subversion techniques for hiding malicious process memory from memory forensics and live analysis.
- Detection approaches for all three subversion techniques.
- Windows and Linux proof-of-concept implementations for all three subversion techniques, and two Rekall plugins which automate the detection of hidden shared memory.
- Extensive evaluation with live analysis tools and the memory forensics frameworks Rekall and Volatility.

1.3. Related work

In the following, we briefly contrast our approach with related work that relies on anti-forensic techniques which subvert the memory view of forensic analysis tools.

Stüttgen and Cohen (2013) presented several techniques that altered kernel functionality, mostly used by forensic acquisition tools, in order to provide a falsified memory map or prevent certain memory ranges from being correctly mapped. As these modifications can be detected through hash-based approaches, our techniques rely on Direct Kernel Object Manipulation (DKOM) to manipulate several dynamic memory data structures instead of static code.

Gargoyle (Lospinoso, 2017) is a memory scanning evasion technique for Windows, that deceives forensic investigators by placing malicious code into non-executable memory. To enable its execution despite the missing permission, Gargoyle initializes a system timer which, on expiry, is configured to launch a Return Oriented Programming (ROP) chain that remaps the malicious memory accordingly. This has the advantage that no additional setup code is required for altering the access permissions of the malicious memory. With the memory's permissions being redeclared, control flow is diverted to the now executable malicious code. Upon completion, Gargoyle tail-calls to a specifically prepared trampoline which withdraws the executable permission from the respective memory ranges and reinitializes the timer to restart the cycle. To detect Gargoyle, researchers suggested to audit system timers and their corresponding handling routines in order to determine conspicuities. While withdrawing executable permissions of malicious ranges successfully conceived several forensic tools, the approach does not actually hide the respective memory. Instead, it takes advantage of most forensic tools failing to classify non-executable memory as code. Hence, the malicious areas could still be found when acquiring the memory map of the respective process. In contrast, our techniques aim at entirely preventing memory from being enumerated by forensic tools instead of solely conceiving its presence. In addition, we implemented all of our proposed techniques for both Windows and Linux.

With *Shadow Walker*, Sparks and Butler (2005) introduced a new kind of sophisticated rootkit that is able to hide memory by subverting the translation process of the Windows operating system and provide memory scanners with a tampered memory view. The rootkit instruments the Windows page fault handler to provide different views of the same virtual page depending on the type of access. While instruction fetches directly access the malicious version, data accesses are intercepted and respective PTEs temporarily modified to provide only benign data. Although the technique is generally applicable, Shadow Walker is restricted to hide kernel memory on the Windows system. Similarly, the approach presented by Ooi (2009) subverts a system's page tables with the help of shadow paging to allow a rootkit's memory footprint to be excluded from the acquisition process. We applied the general idea of manipulating memory management data structures to provide benign data instead of malicious content. While Shadow Walker mainly focuses on hiding certain kernel memory contents from being acquired, our techniques aim at tampering with the actual memory layout of a process' user space. In addition, Shadow Walker requires a modification of the page fault handler which seems rather straightforward to detect. Our approach, on the other hand, allows a variety of ways to switch between malicious and benign memory preventing analysis tools from using a generic detection strategy. Compared to the shadow paging approach, which requires a significant performance overhead as well as a modified page fault handler to keep mappings up to date, our techniques are more fine granular, limiting modifications to a few dynamic kernel data structures.

1.4. Outline

This work is structured as follows: Section 2 provides a brief elaboration of the memory paging mechanism on modern x86-64 processors and depicts operating system-specific kernel structures which manage the memory layout of a process. Here on after, Section 3 presents our novel memory subversion techniques, which are then examined regarding detection strategies in Section 4. Section 5 evaluates both, the subversion and detection approaches against current memory forensics and live analysis tools. Eventually, we conclude this paper in Section 6 and point out future research directions.

2. Technical background

This section provides fundamental background knowledge that is necessary to understand the technical concepts of our subversion and detection techniques. In this course, we shed light on the memory translation process of modern x86-64 processors, and depict the structure of a process' virtual memory layout for both Linux and Windows. Readers familiar with the concepts are safe to skip this section.

2.1. Memory paging

For both isolation and resource management, modern operating systems provide each process with its own transparent abstraction of the physical memory. These virtual address spaces are partitioned in multiple segments of predefined sizes, called *pages*, which can be accessed through their corresponding virtual addresses. During a context switch, the operating system schedules other processes by switching the respective virtual address spaces. This makes each process believe to own the entire physical memory of the machine, while in reality, only a part of the physical memory belongs to a single process. Similar to the virtual address space, the physical memory is structured into multiple *page frames*, referenced by sequentially ordered Page Frame Numbers (PFNs). To map a virtual page to a physical frame, a processor's Memory Management Unit (MMU) traverses a set of hierarchically ordered paging structures. In this course, the MMU partitions a virtual address into

several parts that serve as indices into the respective paging structures. This process repeats until the MMU either encounters an error or reaches the final Page Table (PT) containing the respective PTE. Amongst others, this entry provides access rights, a present bit (indicating whether a mapping is active), and the PFN of the final frame which maps the actual data of the virtual page. In case of a valid entry (present bit set) and sufficient access rights, the processor is eventually able to address the intended data. In case the MMU encountered an error during the translation process (e.g., through insufficient access rights), the processor generates a page fault which requires subsequent handling by the operating system's kernel. Further information can be taken from Intel's processor manuals (Intel, 2019).

2.2. Process address space

Each virtual address space is subdivided into a kernel and an user space portion. While the first maps an operating system's kernel, and is the same for every virtual address space, the user space appears as the memory range actually accessible by a process. Therefore, the user space is also denoted as the *process address space* or *process memory*. To isolate the kernel from unprivileged user space accesses, the corresponding PTEs are configured to restrict accesses. When a new executable is launched, its contents (e.g., code and data) are loaded and mapped into a process' address space, and bundled into several segments. These are typically represented by OS-specific OS-specific Memory Area Structures (MASs) which allow the separation of code and different data segments, e.g., heap, stack, etc.

The Linux kernel structures the address space of each process into several non-overlapping Virtual Memory Areas (VMAs) (Love, 2010, pp. 309–315). Implemented by `vm_area_struct` structures, these VMAs provide information about both occupied and free areas of the process address space, and include access permissions as well as boundaries of their corresponding memory ranges. VMAs are stored in both a double-linked list and a red-black tree allowing a more efficient access. Similarly, Windows processes use a self-balancing binary AVL tree to keep track of already occupied address ranges (Yosifovich et al., 2017). For each range, the tree includes a Virtual Address Descriptor (VAD) which stores meta information of the corresponding virtual memory range.

2.3. Shared memory

Shared memory is, as the name suggests, intended to be shared among processes (e.g. to allow the exchange of data). It is, however, also possible that a process uses shared memory without actually sharing it. In order to use shared memory, there are, on Linux and Windows, two distinct steps involved: Creating a shared memory segment and mapping it into a process, each realized by a specific API. While it is possible to map files from hard disks into a process' memory space, we will focus on anonymous respectively page-file-backed (Yosifovich et al., 2017; p. 316) memory in this paper. On Windows, shared memory is realized by memory-mapped files (Yosifovich et al., 2017; p. 315, which can be used with the APIs *CreateFileMapping* (Microsoft Corporation, 2018a) and *MapViewOfFile* (Microsoft Corporation, 2018b). On Linux, there are at least three different types of shared memory:

POSIX shared memory objects: Created with *shm_open* (Kerrisk, 2017) and mapped with *mmap* (Brouwer and Kerrisk, 2019).

Anonymous files: Created with *memfd_create* (Kerrisk and Herrmann, 2019) and mapped with *mmap* (Brouwer and Kerrisk, 2019).

System V shared memory segments: Created with *shmget* (Bovet and Cesati, 2005 p. 789; Bai, 2019) and mapped with *shmat* (Ciucci, 2019).

While MASs are also used to describe shared memory, there are more steps and structures involved in order to create and manage shared memory. This is especially the case because MASs are only related to one process and bound to its virtual address space. For each shared memory segment, there is, on Linux and Windows, one structure that can be seen as a central management structure, references wise. On Windows, this structure is `_CONTROL_AREA` (Yosifovich et al., 2017 p. 407) and on Linux it is `file`.

While a handle to a *section* object (an instance of the `_SECTION` structure) is returned upon a call to CreateFileMapping, this structure is more or less just a reference to the `_CONTROL_AREA` structure. In order to get the protection information and references to the actual physical memory about a shared memory segment on Windows, the `_SUBSECTION` can be used, which is referenced by the `_CONTROL_AREA` and VAD structure. There can be more than one `_SUBSECTION` associated with shared memory, which is typically the case for mapped image files, but can also happen in other cases (Martignetti, 2012 p. 342). A `_SUBSECTION` instance contains a reference to the next `_SUBSECTION` in its `NextSubsection` field, while each one describes a particular memory area, e.g., in regards to its protection and range.

On Linux, all three types of shared memory have an associated `file` object, which can be used to get the actual memory via the referenced `address_space` structure (Bovet and Cesati, 2005, p. 601 ff.]. The `file` object can either be accessed with the `vm_area_struct` structure's `vm_file` member (if the shared memory is currently mapped), or via the `task_struct` structure (if the shared memory segment has at least been created).

2.4. The physical view on memory

Both Windows and Linux manage the physical address space with special structures. On Windows, there is the Page Frame Number Database (PFN DB) (Yosifovich et al., 2017; p. 425) which is more or less an array of `_MMPFN` stucts, indexed by the physical page's PFN (Martignetti, 2012, p. 128). Each `_MMPFN` instance describes the corresponding physical page and contains at least the information about the page's state (pages in the same state are managed in lists for faster retrieval). This state can, e.g., be Active (part of a working set), Standby (previously belonged to a working set but was removed) or Zeroed (freed and initialized with zeroes). Furthermore, for some states such as Active, the `_MMPFN` instance points to the corresponding PTE (Martignetti, 2012, p. 128 ff). On Linux, there is a similar view on the physical space: An array of `page` structures (Gorman, 2004), also indexed by the physical page's PFN and managed in lists. Each `page` object contains a field called `mapping` (`page- > u1- > mapping`), pointing, if not null, either to an `address_space` or an `anon_vma` object (the latter is used for reverse mapping of anonymous pages (Corbet, 2010)).

3. Memory subversion techniques

This Section describes three novel memory subversion techniques that allow the hiding of selected parts of the process memory of a user space application. The approaches can either be used individually, or even combined depending on the use case. Usually, this scenario constitutes an attacker hiding malicious parts of an otherwise benign process that could either be launched by the adversary, or infected during its run time. These malicious memory regions could contain injected shellcode, additionally loaded libraries, or parts of the application itself. The first two techniques modify MASs and PTEs, requiring kernel level access similar to

DKOM attacks. The third approach is based on the manipulation of shared memory mappings which does not require elevated privileges altogether. Whenever the hidden parts are required to be accessed, each of the forthcoming techniques temporarily unhides the respective memory by restoring relevant data structures. Therefore, preserving the original data prior to the actual manipulations is indispensable.

3.1. MAS remapping

The general idea of this technique is to modify MASs to remap malicious memory regions of the process address space to areas containing benign data. Consequently, the virtual start and end addresses of a MAS are manipulated accordingly. As many forensic tools rely on the integrity of these kernel data structures, manipulating MASs potentially tampers with their analysis results. In the following, we expound further details for Linux and Windows, since both provide their own implementation of a MAS. Section 2.2 gives further information about the memory layout of a process.

For Linux, remapping certain MASs implies the manipulation of VMAs by altering the `vm_start` and `vm_end` fields within the `vm_area_struct` structures that refer to the malicious areas. Fig. 1 illustrates the remapping of the malicious area M to the benign region B. Both areas could contain dynamically loaded libraries, for example. Initially, VMA B refers to the benign memory area B, while VMA M specifies the address range of the malicious area M (dashed arrows in Fig. 1)). To remap VMA M to reference area B, both `vm_start` and `vm_end` are manipulated to address the memory range of B. Alternatively, M could be redefined to any other memory range within the process address space. However, this could lead analysis tools to display improper data. For example, when displaying data within a process' executable segments, analysts expect code to be discovered. Therefore, the remapping of address ranges should target memory regions of the same type, or manipulate their access permissions accordingly. Regarding executable segments, this could either be the address range of benign code of the application or a shared library. The downside of remapping VMA M to match the boundaries of VMA B is the problem that it might be suspicious for a process specifying two VMAs that refer to the same memory. Hence, adversaries might prefer to slightly alter these ranges. Alternatively, malware might strip the malicious parts of an otherwise benign area by shortening its address range accordingly. As the manipulation does not alter the underlying PTEs, the respective memory can still be accessed by the process without the necessity to revert the respective VMAs. However, additional memory that is allocated by potentially hidden code might corrupt the memory layout of the process as the kernel expects the address range of area B to be unused, potentially corrupting parts of the hidden memory. As new memory is usually allocated on the process heap, this only appears as a minor issue. This is not the case when explicitly specifying a certain memory range, or dynamically loading shared libraries (e.g., via `dlload(3)`) as these are automatically mapped to a distinct area of the process address space that might occupy the hidden memory. Depending on the use case, it might thus be preferable to restore the original boundaries of VMA M each time the hidden code is about to be executed.

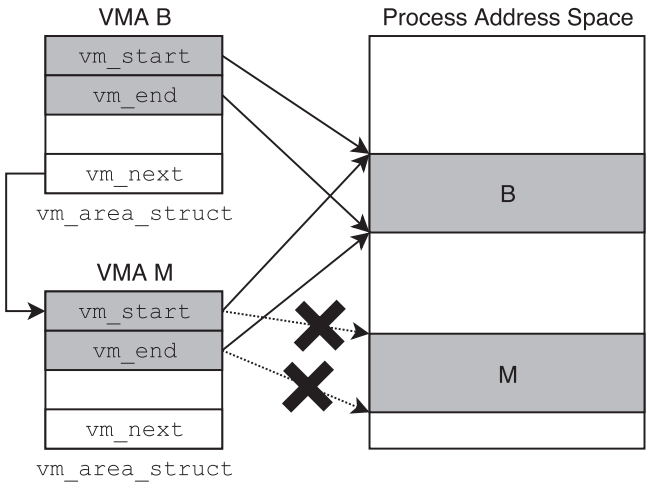

Fig. 1. MAS Remapping for the example of Linux.

A similar situation arises when hiding memory of Windows processes. Manipulating the address ranges within the VAD nodes allows an adversary to deceive analysis tools to investigate false memory areas. The respective Windows kernel structure representing a VAD node can be represented by several structure (e.g., `_MMVAD_SHORT`), describing the actual memory range through its `StartingVPN` and `EndingVPN` fields. Altering these fields to reference benign memory within the process address space would lead investigators to analyze the wrong memory range. Similar to VMAs, the VAD tree is typically not consulted anymore, once it has been fully established and the associated page mappings were created. This allows memory to be accessed despite being hidden within the VAD tree.

3.2. PTE subversion

As described in Section 2.1, PTEs are part of the translation process between the virtual and the physical address space. Among others, their fields indicate the presence of a page and the location of its physical frame through a PFN. Similar to MASs, not only the operating system but also forensic analysis tools depend on these information to successfully access the actual memory. This makes PTEs an interesting target for adversaries, as manipulations might prevent malicious data from showing up in various analysis tools. This section discusses two additional subversion methods which offer a convenient way to hide process memory on both Linux and Windows running on the x86-64 architecture. Although differing, both rely on the manipulation of PTEs, so that these do not refer to the malicious pages anymore.

3.2.1. PTE remapping

Similar to the MAS remapping technique described in Section 3.1, manipulating PTEs enable adversaries to remap malicious memory ranges to benign areas. In contrast, this approach does not necessarily alter the virtual memory layout of a process. To determine the PTEs of the malicious areas, a process' MASs can be consulted for their virtual address ranges and subsequently translated by manually traversing the respective paging hierarchy in a page-wise granularity. Once found, the PFNs of these PTEs are altered to refer to an equal amount of benign page frames. Fig. 2 illustrates the baseline scenario. The graphic shows two distinctive memory areas, one malicious and one benign, each consisting of exactly one page. Two distinctive virtual addresses refer to the malicious page M (`0x1234`) and the benign page B (`0x5678`). Their corresponding PTEs both have the Present bit set and initially contain different PFNs referencing the respective page frames. When changing the PFN of the malicious PTE to point from frame M to frame B, both virtual addresses resolve to the same physical page frame of the benign page B. The PTE remapping approach allows page B to be any physical frame, either belonging to the same process or any other part of the physical memory. Although any page frame could be used as a redirection target, mapping page frames that contain

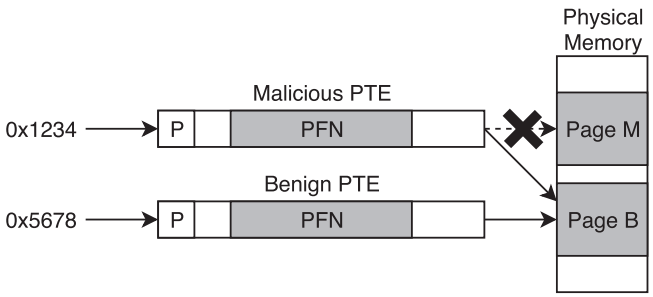

Fig. 2. PTE remapping.

similar data is preferred to avoid conspicuous inconsistencies. This means that malicious pages that initially contained executable code should preferably be remapped to page frames holding executable code. Therefore, benign MASs of the same or other processes can be used as an indicator to find appropriate memory areas with the same access rights that can be used as a target. In contrast to MAS remapping, PTEs of hidden memory portions must be restored prior to be accessed.

3.2.2. PTE erasure

An alternative to remapping PTEs is to let malicious memory ranges appear to be not yet initialized, accessed or present at all. This is achieved by erasing PTEs of malicious pages. Solely invalidating a PTE by clearing its present bit is not sufficient, however, as analysis tools may traverse a process' paging hierarchy searching for remaining information. Therefore, each PTE referencing a malicious area must entirely be nullified. Since this inherently invalidates the PTE, subsequent accesses will generate a page fault to be handled by the kernel (see Section 3.4). Consequently, forensic tools fail to correlate the page to the respective process. As the PTEs of swapped-out pages still contain valid information (despite being marked as non-present), these are not affected by our approach. Similar to PTE Remapping, accessing the hidden pages requires the correct restoration of their original values.

Because neither Windows nor Linux examine a process' PTEs in order to find available memory, the pages related to the erased PTEs shouldn't be at risk of being freed or reused by the OS, and we also didn't encounter any problems in this area during our evaluation. For example, Windows uses mainly the PFN DB lists for the management of physical pages (e.g. the *Zeroed* list is used to serve new requests for physical pages; see Section 2.4), as it is more efficient than iterating through the PTEs of all processes.

3.3. Shared memory subversion

From a process' point of view, data residing in private memory is lost as soon as it is unmapped, whereas shared memory is still accessible and kept available as long as at least one process holds a handle to it. As mentioned in Section 2.3, shared memory does not have to be shared among multiple processes, however. Furthermore, it can be used to store and execute code. The process of hiding memory via shared memory can be broken down into four steps:

1. Creating a shared memory section.
2. Mapping the shared memory in a process.
3. Writing malicious data to the mapped area.
4. Unmap the shared memory.

Assuming an adversary intends to execute malicious code residing in shared memory, the process from this point on is as follows:

Unhide: Mapping the shared memory into the process.
Execute: Execute the code.
Hide: Unmapping the shared memory.

As the shared memory is only mapped during the execution of the malicious code, and most analysis tools solely concentrate on currently mapped memory, the malicious memory stays undetected (if it is not caught during execution). Since this technique does not involve any DKOM and only uses functionality available for unprivileged users, it does not require elevated privileges and runs solely in user space.

3.4. Further considerations

While MAS remapping requires only a small amount of modifications to execute the malicious code, which also do not need to be undone until the termination of the process, both PTE subversion and the shared memory subversion techniques require the hiding to be temporarily undone in order to execute. But there are some more considerations regarding the PTE subversions, to guarantee the stability of the running system. Changing memory management structures is a major intervention in a system and can lead to severe problems or situations that can affect not only the infected process, but the entire system. In case the respective process is forcefully terminated while some of our subversion techniques are active, the kernel misses to clean up several resources and references. Since certain malicious page frames might not be referenced anymore, they are never released. This may result in the inability to terminate the process or incomplete stop jobs during the system shutdown. Therefore, we hooked the process termination routines of the respective operating systems in order to notify the malware to revert all manipulations prior to the process' exit. While on Linux, this can be done by installing an `ftrace` hook at the kernel's `do_exit` function, Windows provides the possibility to catch the event via the `PsSetCreateProcessNotifyRoutine`. In addition, we tested different strategies to unhide the malicious parts of a process. One possibility is to hook the operating system's scheduler and unhide certain memory ranges whenever the infected process is about to be executed. As the manipulation of MASs leaves the remapped areas accessible, this is only required for the PTE-based techniques. This also allows the regions to be rehidden as soon as the scheduler hook detects the preemption of the malicious process. Another approach would be a user/kernel space communication, instructing the kernel component to hide/unhide the memory, whenever necessary, or rely on ROP-based techniques like demonstrated by Gargoyle (Lospinoso, 2017).

Regarding Linux, we came across a few more details that were important for our implementation. To avoid any discrepancies between the modified virtual memory layout and its physical counterpart, it was necessary to adapt the Resident Set Size (RSS) of a process. The RSS gives information about the size of a process' occupied physical memory. In particular, it contains several counters which hold the number of page frames for different types of memory. To prevent these counters from informing about missing page frames, they had to be properly adapted. In addition, each process' `mm_struct` includes a `map_count` which indicates its number of VMAs. Consequently, this number must be decreased when hiding VMAs. The `nr_ptes` counter within a process' memory descriptor is another important piece of information that needs to be forged accordingly. It specifies the number of page tables which are used by a process. When removing VMAs or erasing PTEs, this number should be decreased eventually. Another

issue that arises due to the PTE subversion techniques is that hidden pages are not referenced by the kernel anymore. This could lead the kernel to reclaim these pages and assign them elsewhere. As this potentially corrupts the hidden data, such pages must be locked down to prevent reassignments. This can be done by setting the `PG_reserved` bit within the `page` structures of the respective frames. Furthermore, this structure contains a reference count (`_refcount`) that needs to be adapted before remapping a PTE. Otherwise, the modification could result in a malfunctioning system. To guarantee the correct access to the malicious areas, as well as the termination of the process, all of these additional modifications need to be reverted.

On Windows, changing the PFN can lead to application, and also system crashes. One of the reasons are inconsistencies with the PFN DB (see Section 2.4). If we change a PFN as illustrated in Fig. 2, Page B now is referenced by two PTEs, while the `_MMPFN` instance for Page B only points to the benign PTE. This inconsistency leads to system crashes, e.g. when the process is scanned with YARA. In order to resolve this issue, it might be possible to also modify the `_MMPFN` instance. To limit the modifications of kernel structures and minimize side effects influencing the evaluation, we store the current PFN, erase the PTE (similar to PTE Erasure), and afterwards access the page. As a result, Windows assumes this to be the first access to a committed page and creates a new one, with a new PFN and an according `_MMPFN` instance. From now on, we just switch between the old PFN (containing our malicious data) and the new one, with the benign data. After using this trick, we encountered no more application or system crashes during the process' runtime, and were able to successfully perform all evaluations. However, as soon as the process is terminated or the operating system shuts down, the system crashes. We tried to resolve this issue by modifying the Working Set Size (Martignetti, 2012, p. 115) (similar to the RSS counters), but it did not work. Eventually, we left this issue unresolved as it did not affect our evaluations.

To prevent page swapping mechanisms to interfere with the MAS remapping respectively PTE subversion techniques, we lock the malicous pages in memory prior to their hiding (on Linux via `mlock` and on Windows via `VirtualLock`).

4. Detection

This chapter describes detection techniques for each subversion technique covered in the previous chapter, focusing on the memory forensics point of view.

4.1. Detecting MAS remapping

The process of detecting the remapping of MASs is the same for Linux and Windows, and follows a similar approach as shown by White et al. (2012) respectively Block and Dewald (2019). Since the PTEs for the hidden virtual addresses are still part of the page tables, we enumerate all page table entries and compare them with the information from all memory area structures. If we find a mismatch, which would be a non-zero PTE value for a virtual address that is not part of MAS's range, the page is reported. The PTE value further allows us to retrieve its corresponding memory content.

Another, but Windows specific, detection approach is the examination of Working Set List Entries (WSLEs). The Working Set is a subset of physical memory pages used by a process and includes, amongst others, pageable private and shared pages (Martignetti, 2012, p. 114]. Windows tracks these pages with a list of WSLEs (each in-use entry is represented by the `_MMWSLENTRY` structure), by storing the virtual address for each page in a separate WSLE instance. By enumerating this list and comparing the virtual addresses with the ranges from all VADs, it is possible to find hidden memory areas. It should be noted, however, that WSLEs only allow to identify a VAD range manipulation but do not contain a direct link to the actual memory, whereas the first algorithm supports both.

The PFN DB on Windows, respectively the `page` structures on Linux can also be used to identify this subversion technique, which is, however, more complicated, error prone and has a higher computational complexity than the first algorithm. This approach is described in detail in the next section.

One aspect to consider during an analysis is the fact that there might be pages, at least on Windows, which are legitimately not covered by any MAS (White et al., 2012). Consequently, these pages must be manually examined in order to differentiate benign from malicious data.

4.2. Detecting PTE subversions

This section covers the detection of memory hidden with PTE manipulations. As we are yet not aware of any straightforward detection algorithm for this technique, this section describes some approaches that can identify this technique, and in certain cases also reveal the hidden memory, but might fail in others. The specific limitations are covered in the following paragraphs.

A general approach for detecting the exchange of the PFN, in regards to private memory, is to count the references to physical pages by enumerating all PTEs for private pages in all processes. Since each private page should have a unique physical page, there should not be more than one PTE with the same PFN. Otherwise, it is an indicator for a PFN modification. It should be noted, however, that this approach does not work in our case for PFN modifications on Windows, because we acquire a new page for the remapping (see Section 3.4) which therefore leaves no duplicate PFN references. Furthermore, there are at least two more cases in which this approach might fail:

- If the PFN of shared memory pages is exchanged, this approach does not work since multiple references to the same physical page are common for shared memory. While it would be possible, at least on Windows, to compare the PFN of the *MMU PTE* with the *prototype PTE* (Yosifovich et al., 2017 p. 295 ff.), it is also possible to manipulate the *prototype PTE*'s PFN, in order to match with the *MMU PTE*.
- Windows 10 introduced a feature called *Memory combining*, which attempts to save RAM (Yosifovich et al., 2017 p. 459). One mechanism of this feature is to turn identical private pages into shared memory in order to remove duplicates, which could interfere with this detection.

Another approach for the detection of PTE Remapping, but also for PTE Erasure, employs information about physical pages, which are available on both operating systems. On Windows, these information can be retrieved through the PFN DB with its `_MMPFN` stuctures. If a page is currently valid and in the process' working set, there should be a `_MMPFN` instance with an Active state and a pointer to the corresponding PTE (it should be noted that also other states, such as Standby, can potentially be used for verification, which, however, has not been evaluated in this work). Especially the index for the `_MMPFN` instance and the PTE pointer can be used to identify both techniques for active pages: For PFN modifications, the index for the current `_MMPFN` entry does not correlate with the PFN from the referenced PTE, and for PTE Erasure, the PTE shouldn't be zero for an active `_MMPFN` entry (Martignetti, 2012, p. 128).

On Linux, the `page` structures can be used (see Section 2.4). Because `page` instances belonging to the same MAS seem to share

the same mapping object(s), and the vm_area_struct structure also includes pointers to these mappings (via vma- > vm_file- > f_mapping for shared memory and vma- > anon_vma_chain for anonymous memory respectively Copy-on-Write pages (Corbet, 2010)), it is possible to correlate page objects to their corresponding MAS. The detection of PTE subversions is accomplished by iterating over all page objects for the physical address space and comparing those with the page objects for a vm_area_struct instance, by resolving each virtual address. Any page object that belongs to the same mapping object for a MAS, but is not referenced by any PTE of that MAS, is an indicator for both PTE related subversion techniques. There are, however, two problems with this approach:

- This approach can lead to false positives with Copy-on-Write memory, because the MAS points only to the new copy, while the page objects for the physical address space also contain the original page, and hence must be analyzed manually in such scenarios.
- Since the address_space structure is part of the page cache and not all pages are necessarily part of that cache (have an associated address_space object), this approach will fail for those. Furthermore, it is also possible to explicitly delete a page from the page cache via delete_from_page_cache (Kernel Organization, 2019).

Similar to the approach described in Section 4.1, it is also possible to detect at least *PTE Erasure* on Windows with WSLEs. Since there are only WSLEs for currently active pages, there shouldn't be a WSLE with a corresponding PTE value of zero.

Except for the WSLE-based approach, the physical page-based approaches on Linux and Windows also allow to retrieve the actual memory, as the PFN DB index respectively the page structure offset already reveal the memory page's physical offset.

It should be noted that all structures used for detecting our subversion techniques could also be manipulated, in order to evade the detection mechanisms proposed in this section. However, such manipulations might introduce negative side effects for the malicious process respectively the stability of the system, and have not been evaluated in the context of this paper.

4.3. Detection of shared memory subversion

On Linux and Windows, our process of finding hidden shared memory is basically the same and can be broken down to the following four steps, which are done for each running process:

1. Enumeration of all MASs while gathering the referenced _CONTROL_AREA (Windows) respectively file (Linux) structures: These shared memory segments are currently mapped.
2. Gathering the same type of structure via the process' handles (Windows) respectively task_struct (Linux): These have at least been created.
3. Stripping out of scope items.
4. Reporting all file respectively _CONTROL_AREA structures which are currently not mapped in any MAS.

This process is described in more detail in the following two paragraphs for Windows and Linux respectively. It should be noted that the following information are related to the test environments for the memory forensics part listed in Chapter 5, and might differ for others.

Gathering the _CONTROL_AREA structures for step one is done by enumerating all MASs, referenced by the _EPROCESS structures and accessing them via vad- > Subsection- > ControlArea. In order to get the _CONTROL_AREA structures from the process' handles, we need to interpret the data, referenced by the _OBJECT_HEADER's Body field, as an instance of the _SECTION structure, and access it via section_object- > u1- > ControlArea. If only executable memory should be reported, the results can be filtered by examining each _SUBSECTION for its protection information (subsection- > u- > SubsectionFlags- > Protection), and only include _CONTROL_AREA objects that have at least one associated _SUBSECTION instance with executable access rights. While the MMU PTEs hold the final truth regarding a page's executable state (Block and Dewald, 2019), there are no PTEs in the process' page tables to examine as long as the view is unmapped. Although it would be possible to examine prototype PTEs, the _SUBSECTION's protection information is more authoritative because, upon creation, a view's protection bases on the section's protection (Martignetti, 2012; p. 305). Furthermore, depending on the way a view is created, neither structure provides information about the actual page protection. The reason is that a view on a shared memory segment can be created with different protections. E.g., while the protection of a prototype PTE, or a _SUBSECTION, indicates RWX permissions, the memory can be actually mapped as read-only in the process address space. The protection can, however, only be a subset, so if the shared memory segment has a protection of RW, a view can't request or change its protection to RWX, but a protection of RW or R is possible (Martignetti, 2012; p. 310).

At this point, every remaining _CONTROL_AREA resulting from the second step, that is not included in the result set of step one, will be reported as suspicious. Accessing the memory behind a _CONTROL_AREA structure can then be accomplished by iterating over the referenced _SUBSECTION structures which include a pointer to the prototype PTEs: control_area- > FirstSubsection- > SubsectionBase. If, e.g., a prototype PTE is in the Hardware state, the PFN value can be extracted, and hence the physical page read directly.

On Linux, step one is done by iterating over all vm_area_struct structures, referenced by the respective task_struct, and gathering the file objects via vma- > vm_file. The second step depends on the type of shared memory: For POSIX shared memory and Anonymous files, file objects can be enumerated via task- > files- > fds. For System V shared memory, the same can be achieved via task- > sysvshm- > shm_clist. In order to filter the results, we strip file objects for special files such as character devices and sockets by including only regular files (which all three types are). This can be accomplished by checking the file- > f_mapping- > host- > type value for the type S_IFREG (U-Project and 2019. Testin, 2019). As files from hard disks might appear in the list of file descriptors, and can be opened for reading/writing without being mapped into the virtual address space, we also strip those from our results. To differentiate these files from anonymous shared memory, we use the s_id field of the super_block structure, which contains the name of the corresponding block device (Bovet and Cesati, 2005; p. 462 ff.), and access it through the file structure: file- > vfsmnt- > mnt_sb- > s_id. For files from hard disks, this member contains device names such as sda1, and for anonymous memory files such as tmpfs. Since only System V based approach for creating shared memory does support/require an explicit protection regarding execution, testing for that protection does not seem to be a valid approach on Linux. However, even without stripping any shared segments that might not be executable, the amount of file objects being reported by this approach seems to range from 1 to a maximum of 42, depending on the running processes (see the next paragraph). To get access to the

actual memory of the respective shared memory section, the `address_space` structure can be used, as it stores a reference to `page` structures via the `page_tree` field (Bovet and Cesati, 2005; p. 601 ff.). An example for getting memory from `page` structures is implemented in Rekall's `physical_offset` function (Google Inc and 2020. Rekall', 2020).

As a proof of concept, we created two Rekall plugins (one for Windows and one for Linux), which implement the described detection methodology. In order to test their amount of false positives, we created a test environment for Linux and Windows, in which we started several browsers (Firefox, Microsoft Edge, Chromium), Office applications (Microsoft Word, LibreOffice), and opened PDF Documents in a reader - the exact tools with their version numbers are documented in our online repository (Reichenberger et al., 2020). Without filtering, the amount of reported shared memory sections were 1153 on Linux respectively 266 on Windows, and with the applied filter 42 on Linux (one for *cron* and 41 related to Firefox and Chromium) and none on Windows.

5. Evaluation

The subversion techniques described in Chapter 3 have been implemented as a Proof of Concept for Windows and Linux, and the evaluation results are documented in the following section. Besides implementing these techniques for Windows and Linux, we also did an evaluation from a memory forensics and live analysis perspective on both operating systems, and an evaluation of the detection techniques (see Section 5.2). We did, however, not include any detection of the kernel drivers/modules we used for DKOM. The memory forensics evaluation is done with Rekall (Google Inc and 2019. Rekall, 2019) on commit `041d696`, and Volatility on commit `050dab6` (The Volatility Foundation, 2019). Because Rekall (and also Volatility) does not support System V shared memory in its profile generation, we adjusted the `module.c` (Google Inc and 2019. Rekall', 2019) in order to be able to analyze it with our plugins. Furthermore, Rekall currently suffers from bugs in regards to KASLR on Linux, that prevents it from correctly dumping the content of a file object, and to the processing of handles on Windows, which is why we also adjusted some of its core functionality. Our plugins, all written tools with their source code and all modifications to the Rekall framework are documented in our online repository (Reichenberger et al., 2020).

For the live analysis, we use the latest version of Windows 10 (at the time of writing: 1909). Since Rekall and Volatility yet not fully support this version, we use an older one for the memory forensics evaluation (version 1511). In sum, the evaluation test environment consists of three VirtualBox VMs:

- Windows 10 Pro Version 1511 x64, Build 10586
- Windows 10 Pro Version 1909 x64, Build 18363
- Debian 9.9 4.9.0-11-amd64 (4.9.189-3+deb9u2)

To prevent any false positives (malicious memory appears as successfully hidden, but in fact is not), we deactivated the swap space and *zswap* feature for the Debian VM, respectively the pagefile and memory compression feature (Yosifovich et al., 2017; Microsoft Corporation, 2019) on Windows (we did, however, also test our implementations with those features activated, in order to verify that they work as intended; see also Section 3.4). This ensures that malicious pages are not swapped-out by chance and accidently missed during analysis, as Rekall and Volatility currently fail to analyze swap space. Furthermore, we activated *TESTSIGNING* (Microsoft Corporation, 2017) on all Windows machines, in order to be able to load our driver, and activated the debug mode on the live analysis machine for the evaluation with WinDBG.

5.1. Evaluation of subversion techniques

Our proof of concept implementations are trying to hide code, which is executed during runtime by the controlling process. This malicious code is represented in two flavors: Shellcode and libraries. On Linux, we used in-memory library loading with the technique described by Fernández (2018). On Windows, we concentrated solely on Shellcode, especially because there are techniques such as *Reflective DLL Injection* (Fewer, 2013), which do not use operating system APIs to load a library. Hence, they do not leave traces compared to conventional DLL loading (as it is currently the case on Linux). All tests have been done first with the memory hidden, and then, after instructing the malware to reveal the memory, done again in order to verify the malicious memory is in fact identifiable.

For the memory forensics evaluation, we used the following categories of tests with the corresponding Volatility and/or Rekall plugins. If a plugin is only available for one framework, it is indicated in brackets (V for Volatility and R for Rekall), otherwise the same plugin is used on both.

handles: Information about handles and file descriptors.Linux: *linux_lsof* (V), *lsof* (R). Windows: *handles*.

MAS: Information about MASs.Linux: *linux_proc_maps* (V), *maps* (R). Windows: *vadinfo* (V), *vad* (R).

malfind: Detection of MASs containing hidden memory. Linux: *linux_malfind*. Windows: *malfind*.

ptenum (Block, 2019): Detection of hidden memory. Windows: *ptenum* (R).

read: Attempt to read from a (hidden) virtual address: *dump* (R), *dd* (V).

memdump: Dumping the whole process space. Linux: *memdump* (R). Windows: *memdump*.

libdump: Dumping loaded libraries. Linux: *linux_librarydump* (V). Windows: *dlldump*.

masdump: Dumping the memory of all MASs. Linux: *linux_dump_map* (V), *vaddump* (R).Windows: *vaddump*.

yarascan: Scanning for patterns.*r-yarascan* (R), *v-yarascan* (V).

The results in Table 1 can be read as follows:

F: Full access to hidden memory.
I: An indicator for something suspicious.
A: At least an artifact is shown, which can be used for further analysis. The artifact itself is, however, not unusual/suspicious.
X: No plugin available.
N: Nothing about the hidden memory is revealed.

As can be seen, all subversion techniques fail to hide their memory from Rekall's *yarascan* plugin, since it performs a raw search in the physical memory and ignores all memory management related structures. For this scenario, however, the investigator must know in advance what to search for and it is hence not suited as a generic approach. Besides this detection, *Shared Memory shmget* is able to hide its tracks from all other plugins, while the rest of the *Shared Memory* implementations revealed only their file descriptor/section handle (which is on itself not suspicious for a process to have). MAS remapping and the PTE subversion techniques require a kernel component, and at least our Windows implementation reveals a handle to the corresponding device which is used for communication between user and kernel space.

Table 1
Evaluation of subversion techniques with memory forensics.

	handles	MAS	malfind	ptenum	read	memdump	libdump	masdump	r-yarascan	v-yarascan
PTE Erasure	A [c]	A [a]	N	N	N	N	N	N	F	N
PTE Remapping	A [c]	A [a]	A [d]	A [c]	N	N	N	N	F	N
MAS Remapping	A [c]	I	A [d]	F [c]	F	F [b]	N	N	F	N
Shared Memory Windows	A	N	N	N	N	N	N	N	F	N
Shared Memory shmget	N	N	N	X	N	N	N	N	F	N
Shared Memory shmopen	A	N	N	X	N	N	N	N	F	N
Shared Memory memfd	A	N	N	X	N	N	N	N	F	N

[a] Only on Linux with a loaded Library.
[b] No hit for Rekall's memdump on Windows.
[c] Only on Windows.
[d] Only for Shellcode scenario.

On Linux, we could detect the dynamically loaded libraries, what on itself is not indicator for a memory subversion. Since we intentionally loaded all shellcode related pages with `RWX` permissions, these memory regions are reported by malfind. However, in both cases the malicious memory is not revealed. By adjusting these permissions accordingly (Block and Dewald, 2019), the memory could be prevented from showing up on Windows altogether. Only in the case of MAS Remapping, several plugins are able to reveal the hidden memory. Additionally, the MAS output indicates this technique either with duplicate virtual addresses for memory regions or, at least on Windows, with more committed pages than are included in the respective MAS.

For the live analysis evaluation, we used the following categories with the corresponding tools/tests (Windows/Linux-specific tools are indicated in brackets). The exact commands for each category are documented in our online repository (Reichenberger et al., 2020).

handles: Information about handles and file descriptors:`/proc/PID/fd/`(L), Handle (Russinovich, 2019a) (W).

MAS: Information about MASs:`/proc/PID/maps` (L), VMMap (Russinovich, 2019c) (W).

dbgdump: Dumping the process with a debugger:*WinDbg* (W), *gdb* (L).

tdump: Dumping the process with a second tool:*Process Explorer* (Russinovich, 2019b) (W), *gcore* (L).

read: Attempt to read from the (hidden) virtual address:*WinDbg* (W), *gdb* (L).

memsearch: Hidden pattern search via debugger:*WinDbg* (W), *gdb* (L).

yara: Scanning the process for a hidden pattern:*YARA*.

The results in Table 2 can be read similar to Table 1. As can be seen, the results for all *Shared Memory* implementations are mostly the same as for the memory forensics part, but with the addition that the actual memory is never revealed. Only for *MAS Remapping* it is possible to find and reveal hidden memory. In all other cases, the memory content could not be found. Regarding *MAS Remapping*, revealing the hidden memory was only possible on Windows. *gdb* failed to display the hidden memory after accessing it through its virtual address, or to include it in a core file (neither did *gcore*). However, a warning message was logged with the duplicated `vm_start` address. Furthermore, PTE Remapping prevented WinDbg and Process Explorer from creating a process dump. While the error message itself does not reveal anything suspicious, the fact that the dump fails can still be an indicator. Finally, the results for handles and MAS are similar to the memory forensics evaluation regarding all techniques.

Listing 1
Rekall plugin output for hidden Shared Memory on Linux.

```
1  PID    task_name      file_object          devname
2  -----  -------------  -----------------  --------------------
3  123      benign       0x99800c525d00     /SYSV0001e237
```

5.2. Evaluation of detection techniques

Since the detection algorithms described in Chapter 4 are focused on a memory forensics point of view, this evaluation was done only on the Windows 10 1511 VM (see the beginning of this chapter).

We were able to identify traces for all subversion techniques, and also to dump the hidden memory as described in Chapter 4. Because we applied all described detection algorithms manually (except for our implemented plugins), and only for the malicious processes, these algorithms need to be automated for a broader evaluation. All *Shared Memory Subversions* could be detected by our plugins. In contrast, existing tools failed to report hidden System V

Table 2
Evaluation of subversion techniques with live analysis.

	handles	MAS	dbgdump	tdump	read	memsearch	yara
PTE Erasure	A [b]	A [a]	N	N	N	N	N
PTE Remapping	A [b]	A [a]	I [c]	I [c]	N	N	N
MAS Remapping	A [b]	I	I [d]	I [d]	F [b]	F [b]	N
Shared Memory Windows	A	N	N	N	N	N	N
Shared Memory shmget	N	N	N	N	N	N	N
Shared Memory shmopen	A	N	N	N	N	N	N
Shared Memory memfd	A	N	N	N	N	N	N

[a] Only on Linux with a loaded Library.
[b] Only on Windows.
[c] Dump fails on Windows.
[d] Only on Linux: Warning message with memory address.

shared memory sections.

Listing 1 is an example output for our Linux plugin, which reveals the hidden System V shared memory section that is not reported by any memory forensics or live analysis tool (except Rekall's *yarascan*):

6. Conclusion and future work

In this work, we demonstrated three novel techniques, which successfully hide memory from live analysis and memory forensics. Furthermore, we described several detection approaches, enabling an analyst, at least in some cases, to detect these subversion techniques and also reveal the hidden memory.

Apart from the shared memory detection, the remaining approaches need to be automated and evaluated in the future. Besides the discussed subversion technique for shared memory, mapping just a sub view of the shared memory, while skipping the malicious part, could be a valid alternative, and should hence be evaluated. One aspect that has not been covered in this work so far is the detection of the control code that (un)hides the malicious memory. The same is true for the kernel code that is used for DKOM in both the *PTE Subversion* and the *MAS Remapping* scenarios. Future research should evaluate possibilities to adapt our subversion techniques to the kernel space, and integrate techniques to minimize the memory footprint of the control code (e.g., by using ROP chains and timers as demonstrated by Lospinoso (2017)). In addition, Translation Lookaside Buffer (TLB) priming like shown by Sparks and Butler (2005) could mostly prevent the necessity to temporarily revert PTE modifications. Further research should evaluate the effects of manipulating the structures used for our detection approaches, and the effects of Windows' memory combining feature [Yosifovich et al., 2017 p. 459] regarding both our detection and subversion techniques.

References

Bai, L., 2019. Shmget Function Description [Visited on 15.01.2020]. URL. http://man7.org/linux/man-pages/man2/shmget.2.html.

Block, F., 2019. Ptenum Plugin [Visited on 19.01.2020]. URL. https://github.com/f-block/rekall-plugins.

Block, F., Dewald, A., 2019. Windows memory forensics: detecting (un) intentionally hidden injected code by examining page table entries. Digit. Invest. 29, S3–S12.

Bovet, D.P., Cesati, M., 2005. Understanding the Linux Kernel: from I/O Ports to Process Management. O'Reilly Media, Inc.

Brouwer, A., Kerrisk, M., 2019. Mmap Function Description [Visited on 15.01.2020]. URL. http://man7.org/linux/man-pages/man2/mmap.2.html.

Ciucci, G., 2019. Shmat Function Description [Visited on 15.01.2020]. URL. http://man7.org/linux/man-pages/man2/shmat.2.html.

Corbet, J., 2010. The Case of the Overly Anonymous Anon_vma [Visited on 19.01.2020]. URL. https://lwn.net/Articles/383162/.

Fernández, J.M., 2018. Loading "fileless" Shared Objects (Memfd_create + Dlopen) [Visited on 16.01.2020]. URL. https://x-c3ll.github.io/posts/fileless-memfd_create/.

Fewer, S., 2013. Reflective Dll Injection - Github [Visited on 09.01.2019]. URL. https://github.com/stephenfewer/ReflectiveDLLInjection.

Google Inc, 2019. Rekall memory forensic framework [Visited on 11.09.2019]. URL. https://github.com/google/rekall.

Google Inc, 2019. Rekalls Profile Module [Visited on 13.01.2020]. URL. https://github.com/google/rekall/blob/master/tools/linux/module.c.

Google Inc, 2020. Rekalls Physical offset Function [Visited on 19.01.2020]. URL. https://github.com/google/rekall/blob/master/rekall-core/rekall/plugins/overlays/linux/linux.py#L936.

Gorman, M., 2004. Understanding the Linux Virtual Memory Manager. Prentice Hall, Upper Saddle River.

Intel, 2019. Intel 64 and Ia-32 Architectures Software Developer's Manual Volume 3c, vol. 3. System programming guide part 3.

Kernel Organization, Linux, 2019. delete_from_page_cache function [Visited on 19.12.2019]. URL. https://www.kernel.org/doc/htmldocs/kernel-api/API-delete-from-page-cache.html.

Kerrisk, M., 2017. shm_open Function Description [Visited on 15.01.2020]. URL. http://man7.org/linux/man-pages/man3/shm_open.3.html.

Kerrisk, M., Herrmann, D., 2019. memfd_create Function Description [Visited on 15.01.2020]. URL. http://man7.org/linux/man-pages/man2/memfd_create.2.html.

Lospinoso, J., 2017. Gargoyle - a memory scanning evasion technique [Visited on 20.12.2019]. URL. https://github.com/JLospinoso/gargoyle.

Love, R., 2010. Linux Kernel Development. Pearson Education.

Martignetti, E., 2012. What Makes It Page? The Windows 7 (x64) Virtual Memory Manager. CreateSpace Independent Publishing Platform.

Microsoft Corporation, 2017. Enable loading of test signed drivers [Visited on 16.01.2020]. URL. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/the-testsigning-boot-configuration-option.

Microsoft Corporation, 2018a. Createfilemappinga Function Description [Visited on 15.01.2020]. URL. https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga.

Microsoft Corporation, 2018b. Mapviewoffile Function Description [Visited on 15.01.2020]. URL. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile.

Microsoft Corporation, 2019. Disable Windows 10 Memory Compression [Visited on 13.01.2020]. URL. https://docs.microsoft.com/en-us/powershell/module/mmagent/disable-mmagent?view=win10-ps.

Ooi, T., 2009. Stealthy Rootkit: How Bad Guy Fools Live Memory Forensics.

Reichenberger, P., Stripeika, D., Block, F., Palutke, R., 2020. The Public Repository Containing the Code and Binaries Used in This Work [Visited on 30.03.2020]. URL. https://github.com/DFRWS-memory-subversion/DFRWS-USA-2020.

Russinovich, M., 2019a. Handle Version 4.22 [Visited on 17.01.2020]. URL. https://docs.microsoft.com/de-de/sysinternals/downloads/handle.

Russinovich, M., 2019b. Process Explorer Version 16.31 [Visited on 17.01.2020]. URL. https://docs.microsoft.com/de-de/sysinternals/downloads/process-explorer.

Russinovich, M., 2019a. Vmmap Version 3.26 [Visited on 17.01.2020]. URL. https://docs.microsoft.com/de-de/sysinternals/downloads/vmmap.

Sparks, S., Butler, J., 2005. Shadow walker: raising the bar for rootkit detection. Black Hat Jpn. 11 (63), 504–533.

Stüttgen, J., Cohen, M., 2013. Anti-forensic resilient memory acquisition. Digit. Invest. 10, S105–S115.

The Volatility Foundation, 2019. Volatility [Visited on 15.01.2019]. URL. https://github.com/volatilityfoundation/volatility/tree/9df8aa6daabc29c74bf261574ffb5cde2315c7f8.

GNU-Project, 2019. Testing linux file type [Visited on 12.01.2020]. URL. https://www.gnu.org/software/libc/manual/html_node/Testing-File-Type.html.

White, A., Schatz, B., Foo, E., 2012. Surveying the user space through user allocations. Digit. Invest. 9, S3–S12.

Yosifovich, P., Solomon, D.A., Ionescu, A., 2017. Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More. Microsoft Press.