Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries

By

Frank Block and Andreas Dewald

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2019 USA

Portland, OR (July 15th - 19th)

Digital Investigation 29 (2019) S3–S12

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries

Frank Block [a, b, *], Andreas Dewald [a, b]

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

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Code injection
Detection
Windows
Malware
Rekall

ABSTRACT

Malware utilizes code injection techniques to either manipulate other processes (e.g. done by banking trojans) or hide its existence. With some exceptions, such as ROP gadgets, the injected code needs to be executable by the CPU (at least at some point in time). In this work, we cover and evaluate hiding techniques that prevent executable pages (containing injected code) from being reported by current detection tools. These techniques can either be implemented by malware in order to hide its injected code (as already observed) or can, in one case, unintentionally be taken care of by the operating system through its paging mechanism. In a second step, we present an approach to reveal such pages despite the mentioned hiding techniques by examining Page Table Entries. We implement our approach in a plugin for the memory forensic framework Rekall, which automatically reports any memory region containing executable pages, and evaluate it against own implementations of different hiding techniques, as well as against real-world malware samples.

1. Introduction

Memory forensics has become more and more important over the last decade for different reasons: On the one hand, we observe malware that does not persist itself on a persistent storage device and can only be observed in the running state of the victim host. On the other hand, live analysis is not always capable of generating reliable results as the victim host might be compromised with a kernel level Rootkit, using attack techniques that effectively manipulate information gathered during the analysis. The presence of such malware can be proven by analyzing a main memory image of the system (captured by one of the many existing techniques, which we do not want to discuss here). Besides kernel level malware, there is also user space malware which uses its own set of techniques in order to get its task accomplished. One such technique are code injections.

1.1. Motivation

User space malware utilizes code injection techniques to manipulate other processes or hide its existence. However, current tools/plugins for code injection detection are not able to cope with the existing injection techniques and fail to reliably reveal existing malware utilizing certain hiding techniques. This is due to the information they rely on, for example the VAD (Russinovich et al., 2005; Dolan-Gavitt, 2007), which has a protection field that plugins use to detect executable code. An attacker can, however, create executable memory in a certain way so the VAD does not indicate that it is executable and detection mechanisms won't report it. On the other hand, it is possible to exploit the paging mechanism in order to hide injected code. As some plugins prevent to report empty memory (filled with zeros or not yet allocated), they fail to report memory regions related to code injection when the corresponding pages have been paged out. This can also happen unintentionally, when the Operating Sytem writes malicious pages into the pagefile on memory shortage.

In this work, we introduce a novel approach to reveal all executable pages that are of potential interest for an investigator for a given user space process, despite the hiding techniques covered in this paper. Ignored are only yet not allocated memory pages and unmodified pages of mapped image files (loaded executables and DLLs), as these don't contain any injected code. In order to retrieve the actual executable state of a page and to differentiate yet not allocated memory from currently inaccessible memory, we examine the Page Table Entries which we enumerate via the paging structures, instead of the VADS, as it is faster and more reliable

* Corresponding author.
E-mail address: fblock@ernw.de (F. Block).

https://doi.org/10.1016/j.diin.2019.04.008

(White et al., 2012). While this work also covers mapped files, our focus is on anonymous memory as mapped files are already mostly covered by White et al. (2013).

The resulting plugin has been evaluated on both, x86 and x86_64 Windows 7 and Windows 10 VMs. The source code and binary format (where applicable) respectively links and hashes of all applications, malware samples and plugins used in this work are publicly available in our online repository (Block, 2019), to allow an easy reproducibility and verification of our results.

1.2. Contributions

The contributions of this paper are:

- An algorithm which retrieves a page's actual protection from its PTE value and hence, its executable state despite any hiding technique described in this work.
- A Rekall plugin that implements the algorithm and prints all VADs with executable pages, potentially containing malicious code.

1.3. Related work

This paper can be seen as a supplement of the great work done by White et al. (2013). They presented an approach to automatically compare loaded executables and DLLs in memory with the actual files on disk in order to detect any modifications. While they already examined Page Table Entries in some cases in order to retrieve a page's executable state, they still rely on the VAD protection and did not consider all states a PTE can be in and hence are still prone to the mentioned hiding techniques. The resulting plugin *hashtest* (White, 2013) is part of our evaluation.

Besides this more generic approach to detect injected code, there has been some research concentrating on the detection of specific code injection characteristics, which also resulted in Volatility plugins. One such example is the work by KSL group (2017a), which also covers the VAD-protection hiding technique. While their approach is not prone to this hiding technique in the context of Process Hollowing, the paper does not offer a generic way of detecting executable pages despite this technique.

The other plugins that are part of our evaluation are:

- **hollowfind** (Monnappa, 2016b, 2017) Detects different types of Process Hollowing attacks.
- **threadmap** (KSL group, 2017b) Detects Process Hollowing attacks and anonymous VADs pointed to by threads.
- **malfofind** (Pshoul, 2017a) Detects Process Hollowing attacks.
- **Psinfo** (Monnappa, 2016a, 2016c) Detects suspicious processes by checking parent processes, command line arguments, executable paths and VAD protections.
- **malthfind** (Pshoul, 2017b) Detects call stack addresses pointing to anonymous memory regions.
- **gargoyle** (Countercept, 2018; Hammond, 2018) Detects the Gargoyle hiding technique. During our tests, the plugin failed very often with exceptions and hence, we excluded it from the evaluation.
- **malfind** (Ligh et al., 2014) Detects VADs with execute permission. This plugin is available for Volatility (The Volatility Foundation, 2017) and Rekall (Google Inc, 2019).

1.4. Outline

This work is structured as follows: Section 2 covers several code injection techniques and especially fundamentals about Page Table entries, which are required for the analysis and algorithm presented in Section 4. In Section 3 we describe how injected code can be hidden from detection tools/plugins, Section 5 covers the evaluation of our and other detection plugins on the code injection techniques and real-world malware examples which utilize the hiding techniques, and Section 6 concludes this paper.

2. Fundamentals

This section describes the basics of our work. The first part covers the code injection techniques that are used for the evaluation. As already mentioned, we are in general interested in executable pages but not in all kinds, as for example unmodified pages of mapped image files are not interesting in the context of code injections. Thus, the subsequent sections describe how we can retrieve the information needed to get the executable state of each page and differentiate interesting from uninteresting pages.

2.1. Code injection techniques

This section shortly describes several publicly known injection techniques, against which we evaluate our approach in Section 5.1. We concentrated on injection techniques that use anonymous memory regions, as it is the focus of this work.

2.1.1. Remote Shellcode Injection

Remote Shellcode Injection is the simplest code injection technique, which consists of only three steps:

1. Allocate memory in the target remote process with *EXECUTE_READWRITE* protection.
2. Write the shellocde into the target process.
3. Execute the injected code.

2.1.2. Reflective DLL injection

The result of this technique (Fewer, 2013) is similar to a call of *LoadLibrary* by a victim process: A new library is loaded in the context of the victim process. The loading process is, however, not performed by an Operating System API but instead by a loader function within the DLL itself. The attack steps are similar to *Remote Shellcode Injection* except that the DLL is written into the allocated memory region. When the reflective loader within the DLL is executed, it loads itself (the DLL) into a new memory region by performing the same tasks as *LoadLibrary*. The result of a successful *Reflective DLL Injection* are hence typically two new VADs in the victim process with *EXECUTE_READWRITE* protection and one loaded DLL but without the unwanted side effects that result from a *LoadLibrary* call (the DLL will for example not show up in the list of loaded DLLs).

2.1.3. Atom Bombing

The part, which the injection technique *AtomBombing* (enSilo inc, 2016) got its name from, is the usage of Windows' global atom table. An atom table is an indexed table, storing strings, while the *atom* (a 16-bit integer) serves as the index (Microsoft Corporation, 2018b). This table is exploited in order to make data, originating from one process, accessible to another process. By combining the atom table with the ability to instruct the target process to call a specific API (*GlobalGetAtomName*), it is possible to write code from one process to another and hence, in essence, rebuilds the functionality of the API *WriteProcessMemory*. The more interesting part of this technique for this work is where and how the injected data is stored, which consists of two parts. The first is a

ROP chain that gets written to an already existing RW memory region and the second is the result of executing the ROP chain: a newly allocated RWX memory region where the shellcode gets copied to and executed.

2.1.4. Process Hollowing

The general approach of this technique has not changed significantly from the first known description in 2004 by Keong (2004) and boils down to the following steps:

1. Create a process in the suspended state (ideally with a benign executable).
2. Unmap the memory region of the original executable (e.g. via the *ZwUnmapViewOfSection* API).
3. Write the new executable to the victim process (e.g. via the *WriteProcessMemory* API).
4. The start address of the suspended thread is patched with the one from the new executable.
5. Thread is resumed (e.g. via the *ResumeThread* API).

2.1.5. Gargoyle

Gargoyle is not an injection but a hiding technique that can however be applied to injected code. The trick of *Gargoyle* is to set the permissions of all pages containing the malicious code to non-executable as long as the code doesn't need to run and only sets them executable as long as the code is running.

Now that we have shortly revisited those different injection techniques, the next sections describe fundamentals about private and shared memory, PTEs, transition state and further basics.

2.2. Private and shared memory

Private memory is described by a VAD, only visible to the owning process and does only contain anonymous memory (no mapped files). While the creation of private memory specifies a certain protection for the resulting memory pages, it is possible to change the protection of all or specific pages, belonging to the VAD, later on. This change can either restrict the access (e.g. from *EXECUTE_READWRITE* to *READWRITE*) or extend it (e.g. from initially *READONLY* to *EXECUTE_READWRITE*).

Shared memory on the other hand is, as the name suggests, intended to be shared among different processes to allow an easy exchange of data. It is, however, also used by the image loader to map the executables, DLLs and device drivers into memory (Yosifovich et al., 2017, p. 316). The image loader typically uses *Copy-on-write* protection for them, so a memory modification will not affect other processes or the file itself.

Besides this automatic creation of shared memory, it is also possible to create shared memory manually, which applies to files and anonymous memory. This memory is normally shared among multiple processes but can also be used by just one process (Yosifovich et al., 2017, p. 315). The shared memory is represented by a so called section object (Yosifovich et al., 2017, p. 405), created with a specific protection. There seems to be, however, no documented way to change that protection after creation. In order to access shared memory, a process must map at least one view of the section object, which creates a VAD and maps it into the process' virtual address space. This view can have a different, but not an arbitrary protection: It is possible to assign a protection less or equal to that of the section object (e.g. from *EXECUTE_READWRITE* to *READWRITE* but not the other way around) (Martignetti, 2012b, p. 310). The same applies for protection changes of the corresponding memory pages later on.

2.3. Page Table Entries and the Page Frame Number database

A Page Table Entry (short PTE) is part of the translation process from a virtual to a physical address and consists of a 64 bit value, split into bitfields and flags (Cohen, 2014) (Intel Corporation, 2018a, p. 4–27). For an active page, it contains the so called Page Frame Number (short PFN), which points at the physical page containing the content for the virtual page. The Page Frame Number database (short *PFN DB*) is an array of *_MMPFN* structs with the *PFN* as an index, which keeps track of physical pages and is maintained by the Windows Operating System (Yosifovich et al., 2017, p. 425). It is primarily used to accelerate the process of finding available physical pages but serves also valuable information for our purposes. Every *PFN DB* entry describes one physical page and contains a field called *PrototypePte* (accessible via its *u4* member), which is a bit flag that is set when the physical page belongs to shared memory.

There are two *major* types of PTEs to distinguish: The ones that are accessed by the MMU to translate a virtual into a physical address and the so called prototype PTEs, which are used in the context of shared memory and are stored in a different area of the kernel address space (Cohen, 2016). Throughout this paper we call the first type MMU PTE and the second prototype PTE. A prototype PTE tries to solve the problem of updating the information for a page shared among different processes (Martignetti, 2012b, p. 295–300) (the details on how a prototype PTE works are not important for this work). The *PFN DB* entry has a member called *PteAddress* which points to the physical page's describing PTE. For private memory, this is a MMU PTE and for shared memory it is a prototype PTE in which case the *PrototypePte* flag indicates just that and hence, allows us to differentiate private from shared memory.

Pages belonging to mapped image files (loaded executable or DLL) have this flag set as long as they are not modified. As soon as they are modified, the *Copy-on-write* protection comes into play and a new private page is mapped for this page and process, preventing modification side effects on other processes. With the new page comes also a new *PFN DB* entry, which now has the *PrototypePte* flag unset (Cohen, 2016) and allows us to identify modified pages for mapped image files.

2.4. The different states of Page Table Entries

For each state, a MMU PTE can be in, there is a specific struct in Windows, describing its bitfields and flags. Depending on the state respectively applied struct, the same bits can have a different meaning so it is important to apply the correct struct before interpreting a MMU PTE value. We refer to a certain state in the following sections also as an *instance* of a specific struct. While the knowledge about the translation process is considered common knowledge and not in particular required to understand the rest of this work, the different states and their function within Windows are fundamental and are explained in more Detail in the following sections.

The following sections will also cover the *Protection* member which is a bit field, storing a value defined by Windows' memory manager and represents a page's protection. The value corresponds to Windows' memory protections (Microsoft Corporation, 2019a) but uses different constants (Martignetti, 2012b, p. 104) (for example a value of 6 means *EXECUTE_READWRITE* (ReactOS Foundation, 2013)).

2.4.1. Hardware state

There is one flag that is shared among all MMU PTEs: The *Valid* flag. Only if this flag is set, the virtual address belongs to an active physical page and the MMU will process the PTE. The struct that can be applied in this case to interpret the PTE is *_MMPTE_HARDWARE*,

which contains a member called *NoExecute* that corresponds to bit 63 (Intel Corporation, 2018a, p. 4–27) (the *NX* bit). By checking this bit we can determine a page's executable state as an unset *NX* bit in this case allows the CPU to fetch and execute instructions. A MMU PTE in hardware state can belong to a private page or to shared memory (anonymous, mapped data or image file) and can't be distinguished solely on the PTE value. For this differentiation, the *PrototypePte* flag of the *PFN DB* entry must be examined.

2.4.2. Transition state

If the *Valid* flag is unset, the MMU does not process the PTE any further but a page fault is generated where the Operating System will interpret the state of the PTE and act accordingly (Cohen, 2014). While a MMU PTE in transition state is not valid (has an unset *Valid* flag), the corresponding physical page is still available and the *PageFrameNumber* still points to it. This state is, as the name suggests, a transition phase from an active state into another one (the next state depends on the type of memory) and gives the process a last chance to access the page before it is removed from its working set and the physical page freed for other content. This state can, similar to the hardware state, be reached for private and shared memory. The struct to apply in this case is *_MMPTE_TRANSITION* and a MMU PTE in this state has the *Valid* and *Prototype* flag unset and the *Transition* flag set. Regarding the executable state for this case see Section 4.1.

2.4.3. Proto-pointer PTE

In this state, the MMU PTE is an instance of *_MMPTE_PROTOTYPE* and should not be confused with a prototype PTE: It serves in fact as a pointer to a prototype PTE and hence is called in this work a proto-pointer PTE (Martignetti, 2012b, p. 297). A proto-pointer PTE has the *Valid* flag unset and the *Prototype* flag set. The proto-pointer PTE is only used in the context of shared memory and only occurs when the corresponding physical page has been accessed before, but is currently not anymore in the working set (a MMU PTE for a not yet accessed shared memory page would be in the *unaccessed state*; see Section 2.4.5).

There are two cases to differentiate:

- If the *ProtoAddress* field has a value of 0xfffffff0000 (or 0xffffffff for x86), the *ProtoAddress* does not directly point to a prototype PTE (its address must be gathered from the VAD (Martignetti, 2012b, p. 311)) and the page protection has to be gathered in a special way: The *_MMPTE_SOFTWARE* struct must be applied to the MMU PTE in order to extract its executable state from the *Protection* field. The differentiation is important since the *Protection* field is on different positions for *_MMPTE_PROTOTYPE* and *_MMPTE_SOFTWARE.*
- In all other cases, the *ProtoAddress* points to a prototype PTE. The protection value can then not be read from the MMU PTE and must be gathered through the prototype PTE (see Section 4.2.4).

2.4.4. Pagefile state

Another invalid state occurs when the physical page has been written to the pagefile (paged out). This state is represented by a MMU PTE instance of *_MMPTE_SOFTWARE*, where the *Valid*, *Prototype* and *Transition* flags are all unset but the *PageFileHigh* field has a non-zero value (Yosifovich et al., 2017, p. 384). In this case, the page's content cannot be read anymore from RAM but must be gathered from the pagefile.

2.4.5. Unaccessed state

When a VAD has been created but its page(s) not yet been accessed, there is no need to actually map a physical page and hence, the MMU PTE value does not need to be set (there isn't a PFN that can be set anyways). This initial MMU PTE value is in this case zero and changed when the page is accessed for the first time. For private memory, such a PTE state is also called *demand zero*, as on access, a page of zeros is mapped in the process' address space (Yosifovich et al., 2017, p. 384). The *unaccessed state* also occurs for all types of shared memory, while in this case a page access typically leads to the mapping of already existing memory into the process' address space. Besides the MMU PTE value of zero, there are two known cases where a not yet accessed page has a non-zero PTE value. The first is the result of changing the protection of a page in the *unaccessed* state. The new protection is then stored in the *Protection* field (can be read by applying the *_MMPTE_SOFTWARE* struct), while all other fields remain zero. The only exception from this are mapped image files. When the protection of a not yet accessed image file's page is changed, it goes into the proto-pointer state.

The second case are so called *guard pages* which technically are also *demand zero* pages. *Guard pages* allow to reserve a huge memory space while not having to commit much of it (the minimum would be one page for the guard page itself), which comes at a lower cost (Martignetti, 2012b, p. 173). On access to a *guard page*, a *STATUS_GUARD_PAGE_VIOLATION* exception is thrown which can be reacted on (e.g. committing more pages). This mechanism is used by the Virtual Memory Manager to automatically increase the user mode stack (Martignetti, 2012b, p. 402) and when used by applications can be handled in the code. These pages share the same characterstic as *demand zero* pages with modified protections: In all our tests, the MMU PTE had only the *Protection* field set.

2.5. Large and huge pages

It is possible to allocate large and huge pages that have a size of 2-Mbyte and 1-Gbyte accordingly on x86 architectures (Yosifovich et al., 2017, pp. 303–304). “Some processors support configurable page sizes, but Windows does not use this feature.” (Yosifovich et al., 2017, p. 405) While the physical page for *normal* sized pages are referenced by the entries in the Page-Table (ignoring all special cases right now), large pages are referenced by an entry in the Page-Directory Table and huge pages by an entry in the Page-Directory-Pointer Table (see Intel's Documentation (Intel Corporation, 2018a, p. 4–21). Large and huge pages have bit 1 (*Valid* or *Present* flag) and 7 (*LargePage* or *PS* flag) set, marking them as a large/huge page (Intel Corporation, 2018a, pp. 4-24–4-27), are non-pageable and not part of the working set (Yosifovich et al., 2017, p. 304).

3. (Un)Intentionally hiding injected code

This Section will primarily describe the hiding techniques in the context of the *malfind* and *hashtest* plugins, as they implement a generic approach of detecting injected code respectively executable pages. The results for the other plugins are described in Section 5.1.

Volatility and Rekall's *malfind* plugin examines the protection field of VADs in order to identify injected code. When a VAD has, besides a few other criteria, a specific protection that includes the *WRITE* and *EXECUTE* rights, *malfind* will identify this VAD as potentially malicious and report it. That means, all VADs without these rights are not considered by this plugin. As already stated in 2014 by Ligh et al. (2014), the VAD's protection field only contains the initial protection, set during its allocation. So a VAD, with a specific protection can contain pages with differing protections. It is for example possible to allocate a VAD with a protection of *READONLY* and later on, change the protection of all containing pages (e.g. via *VirtualProtectEx* (Microsoft Corporation, 2019c)) to *EXECUTE_READWRITE* (KSL group, 2017a). So in order to hide its

code from *malfind* and other plugins relying on the protection field, the attacker just has to allocate the memory initially with a protection without the *WRITE* or *EXECUTE* right and later on, add this right to the pages containing the malicious code. As an alternative, malware can also use an already existing VAD, without the *WRITE* or *EXECUTE* right, and inject its code in an unused area while adding these rights to the corresponding page(s). Because the *WRITE* right is only required as long as code needs to be written, the final page protection does not require this right at all. There is one further scenario: Injected code can be hidden from *malfind* by using shared memory (in particular anonymous) with a protection of *EXECUTE_WRITECOPY* (Monnappa, 2017), as it is explicitly omitted by the plugin. These hiding techniques, except for the *EXECUTE_WRITECOPY* trick, also work for *hashtest*.

The other way to hide injected code is to make it unavailable, from a memory forensics perspective. As paged out pages are not present in RAM anymore but only in the pagefile(s), they are per default not part of a memory dump. The problem with plugins such as *malfind* and *hashtest* occurs when they try to access the data behind an executable page. *malfind* on the one hand wants to ignore empty pages, as there is nothing to investigate, and hence does not report a VAD at all, if all pages appear empty. *hashtest*'s purpose on the other hand is to compare memory pages and hence is not interested in unavailable memory. Since both, an actual empty page and a paged out page that might contain malicious content, appear to *malfind*'s internal check for empty pages as empty, a VAD with e.g. *EXECUTE_READWRITE* protection is not reported by *malfind* if all pages are paged out. *hashtest*'s output shows the VAD in this case in its output, but reports that no page for that VAD is executable.

Paging out pages can for example be triggered manually with the *SetProcessWorkingSetSize* API (Martignetti, 2012a; Microsoft Corporation, 2019b), which allows to set the minimum and maximum working set size of a given process and if both values are set to −1, "removes as many pages as possible from the working set of the specified process" (Microsoft Corporation, 2019b). The result from a successful function call is, for private pages, typically first, a PTE state change from hardware to transition, and afterwards to pagefile state: The pages are written to the pagefile. While the change from transition to pagefile can sometimes take some time, it is possible to accelerate this process by allocating and accessing new memory.

This hiding might, however, also happen unintentionally (without any assistance by the malware). When the Operating System e.g. requires physical pages for new processes and the pages containing malicious code haven't been accessed lately, they can get paged out. This automatic and the manual hiding works for both, private and shared memory and hence can also be used to hide modified mapped image files from *hashtest* (it is, however, not perfectly reliable as the Windows Operating System decides if and when it writes pages to the pagefile). While Rekall supports the integration of pagefiles and hence can prevent this hiding technique, Volatility does not.

It should be noted that e.g. *malfind* prints a VAD referring to non-empty pages, despite the fact that the memory has never been accessed by the process. This happens when a process maps a view of shared memory with active physical pages (they are not paged out) but has not yet accessed them itself. While this behavior doesn't hide any potential malicious content, it adds data to the investigation which has never been part of the associated process.

4. Analysis

In order to detect executable pages of interest despite any of the hiding techniques described in the previous Section, the MMU PTEs must be examined. Its state and the information stored in the corresponding *PFN DB* entry help us decide, whether or not we are interested in the according page, and dictates how we have to gather its protection information. The following sub sections describe our analysis on PTEs and the page protection, an algorithm to retrieve all executable pages of interest and an analysis on processes with deactivated Data Execution Prevention. All analysis results are based on extensive tests on Windows 7 ×64 and Windows 10 ×64 VMs, primarily using the memory allocation tool *MemTest* by Martignetti (2012a), WinDbg and Rekall (Google Inc, 2018a) (see also the documentation for test setups in our public repository (Block, 2019)).

4.1. Retrieving the protection for a page in transition

There is more than one source that offers a *Protection* field for a page in transition state, which includes at least the *PFN DB*'s *OriginalPte* member (used by *hashtest* (White et al., 2013, p. 63)) and the MMU PTE (instance of *_MMPTE_TRANSITION*). The *OriginalPte* field is for example used to store the protection value while the page is active, especially when the permission of a page is changed (Martignetti, 2012b, pp. 198,310). In order to find the correct and fastest source for the page's protection (the MMU PTE would be faster regarding the plugin runtime, as we already have its PTE value and wouldn't have to read and examine the *PFN DB* entry), we set up several experiments to see when and how the MMU PTE's and *OriginalPte*'s *Protection* field are set (the details about the setup and tests are documented in our repository (Block, 2019)). The results for private memory can be summarized as follows:

- When a page in transition state gets active again (hardware state), the old MMU PTE's *Protection* field defines the resulting MMU PTE's protection (*NX* flag and so on). The *OriginalPte* does not influence the final protection.
- When the page goes from active into transition state, the new MMU PTE's *Protection* field is copied from the *OriginalPte*. In this case, the old MMU PTE value does not influence the *Protection* field.

So, as a private page in transition state has the same *Protection* value as *OriginalPte* and moreover, the MMU PTE's *Protection* field dictates the protection for the page when it is getting active again, we use the MMU PTE to retrieve the page's protection.

While the *OriginalPte*'s *Protection* field would be safe to use for private pages, the situation changes for shared memory. A *PFN DB* entry describes one physical page, which, however, can be shared among various processes with different protections for their view. When, for example, mapping a view with *READWRITE* protection of a *EXECUTE_READWRITE* section object, the *OriginalPte*'s *Protection* field will state *EXECUTE_READWRITE*, while the page in the process' address space is not executable. The reason why *hashtest* uses the *OriginalPte* is probably the fact that the MMU PTE for shared memory does not seem to ever enter the transition state but directly changes into the proto-pointer state, in which case the protection is, in particular for mapped image files, not always available from the MMU PTE. When and how the protection must be retrieved for these cases is described in Section 4.2.4.

4.2. Executable page detection algorithm

Fig. 1 illustrates the algorithm used to retrieve executable pages. As mentioned in Section 1, we are not interested in not yet allocated memory pages and unmodified pages of mapped image files. So, there are further tests for some pages, besides their executable state, that have to be done before they are included in the

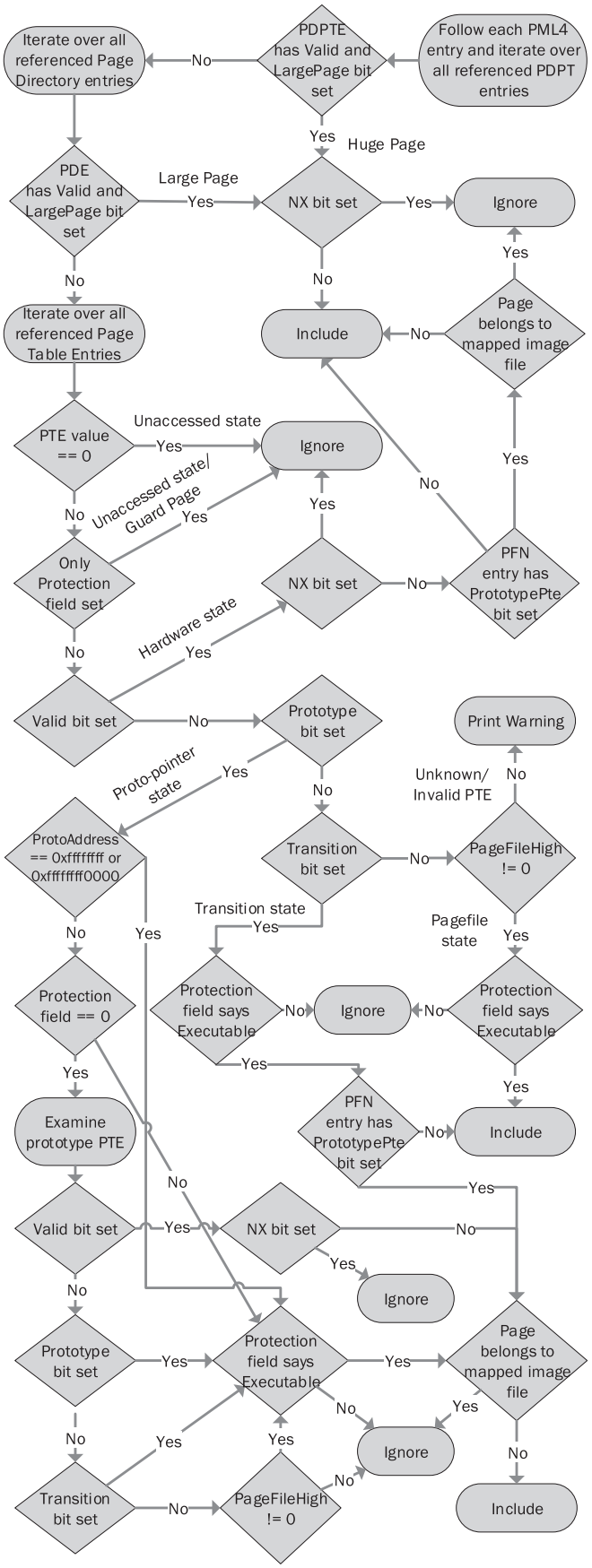

Fig. 1. PTE decision diagram.

collection. It should be noted that, while we don't examine the pages for being writable (as the attacker could have removed the write permission after writing to the page), this algorithm can also be used to retrieve that information.

While it would have been easier to enumerate the memory pages via the VAD address information, our algorithm enumerates the paging structures for the following reasons:

- **Runtime:** When using VAD ranges, it would be necessary to translate every virtual address to a physical one by traversing all the levels of paging structures again and again. Furthermore, various virtual addresses would have to be translated, for which either yet no page table (or a higher table) has been created or their paging structures have been paged out. This can be prevented by enumerating the paging structures directly and hence reduces the required runtime.
- **More reliable:** Since all VADs combined don't necessarily describe the whole user space (White et al., 2012, p. 7) and an attacker might alter the VAD's start and end address through DKOM to hide certain pages, the PTEs are a more reliable resource since they must be accessible and point to the correct physical space in order for injected code to get executed by the CPU.

The following sub sections describe the steps of our algorithm from top to bottom according to Fig. 1.

4.2.1. Large and huge pages

Starting with the *PML4* entries, we get references to Page-Directory-Pointer Table entries (short PDPT entries), which we examine for huge pages: A huge page has bit 1 (*Valid* flag) and 7 (*LargePage* flag) set. While large and huge pages are not referenced by a Page-Table entry, the format of their bit fields is almost the same as for a PTE in Hardware state, as illustrated in Intel's Documentation (Intel Corporation, 2018a, pp. 4-24–4-27) (the main difference is the *PS*/*LargePage* flag). This means we can get the executable state of a large and huge page similar to the approach in Section 2.4.1 by applying the *_MMPTE_HARDWARE* struct to an table entry and examining the *NoExecute* flag. If the *NoExecute* flag for the huge page is unset, we include it.

For each valid PDPT entry that is no huge page, we iterate over all referenced Page-Directory Table entries and perform the same steps as for huge pages. As large and huge pages are non-pageable, we don't have to care for special cases such as transition state. When a Page Directory Entry (short PDE) is valid but no large page, it references a Page Table and we iterate over all entries.

4.2.2. Unaccessed state

If the PTE value is zero, the entry can be ignored and skipped directly. This is done for private but also shared memory (both anonymous and mapped files) which in our tests had always a PTE value of zero when the page has not been accessed so far. The other case are not yet accessed pages with modified protections and *guard pages*. So, while the executable state could be read directly from the *Protection* field, a MMU PTE in this state with only the *Protection* field set can also be skipped, as the actual page does not yet exist. Skipping these PTE values also prevents the printing of never accessed shared memory (see Section 3).

4.2.3. Hardware state

This is the simplest case, where the executable state can be retrieved by applying the *_MMPTE_HARDWARE* struct to the PTE value and checking the bit flag *NoExecute* (this field is missing for x86 and we just check bit 63): If unset, the page is executable.

Before we include a page, we first have to consider unmodified

pages of mapped image files, which we don't want to include. As we can't distinguish a private from a shared page solely from its PTE value, we have to examine the *PFN DB* via the PTE's *PageFrameNumber* field. If the *PFN DB* entry has not the *PrototypePte* bit field set, it belongs to a private page (which includes modified pages from mapped image files) and we can include it. If it is set, the page belongs to shared memory and we examine the corresponding VAD for a mapped image file. We only include the page if it is not related to a mapped image file, which includes, besides shared anonymous memory, also mapped data files (see Section 4.3 for further details).

4.2.4. Proto-pointer PTE

If the *Valid* flag is not set but, when applying the *_MMPTE_PROTOYPE* struct, the *Prototype* field is set, the MMU PTE is a proto-pointer PTE. This state is, depending on our tests and observations, in most cases the default for shared memory as soon as the page is not valid anymore. The PTE containing the actual page's state is the prototype PTE, which in some cases has to be accessed in order to get a page's protection as the MMU PTE does not always contain the protection for this state. If, however, the MMU PTE contains the protection, it must be gathered from here as the prototype PTE might not contain the correct protection (see Section 4.1).

If the *ProtoAddress* field has a value of 0xffffffff0000 (or 0xffffffff for x86), we can get the page's protection by applying the *_MMPTE_SOFTWARE* struct and reading the *Protection* field. If it is executable and the page does not belong to a mapped image file, we include it.

If the *ProtoAddress* field has another value, it normally means the MMU PTE belongs to a mapped image file and could be ignored. Besides pages of image files with changed protections (for those, the *ProtoAddress* value is changed to 0xffffffff0000 and we could read the protection as explained before), we observed in Windows 10 single instances of memory that are no mapped image files but still have a *ProtoAddress* different from 0xffffffff0000, so we have to further examine this state as we want to test these pages for being executable. It should be noted that, in the following cases, the page is tested for being part of a mapped image file and only included if it is not. First, we read the MMU PTE's *Protection* field by applying the *_MMPTE_PROTOTYPE* struct. If this field is not zero we check it for being executable. If it is, we have to read the prototype PTE in order to get the page's protection. The states, the prototype PTE can be in are similar to the MMU PTE and also examined in a similar way (as explained in the other sections and shown in Fig. 1), with one exception: If the *Prototype* flag is set, the prototype PTE is an instance of *_MMPTE_SUBSECTION* and its protection can be read from the *Protection* field.

4.2.5. Transition state

As described in Section 4.1 we can retrieve the protection for a page in this state from the MMU PTE by applying the *_MMPTE_TRANSITION* struct and reading its *Protection* value. If the value corresponds with any protection containing *EXECUTE* rights, we perform the same checks regarding shared memory as described in Section 4.2.3, before including the page. Regarding this state and shared memory, see Section 4.2.4.

4.2.6. Pagefile state

If the *Valid*, *Prototype* and *Transition* flags are all unset, the MMU PTE is in the pagefile state. Reaching this state at this point of the algorithm should happen only for one reason: The MMU PTE belongs to memory which has been paged out (see also Section 4.2.4) and the *PageFileHigh* field points to the page in the pagefile. As there is no active *PFN DB* entry anymore (the physical page has been paged out), it leaves us with only one source for the protection information: The *Protection* field of the MMU PTE (instance of *_MMPTE_SOFTWARE*). If the *PageFileHigh* value is greater than zero, we examine the *Protection* field for an *EXECUTE* right and include the page on a positive match. If the field has at this point a value of zero, it would mean an unknown state and will be reported with a warning.

4.3. Mapped data files

While loading an executable/DLL is done by the Windows Loader and involves several tasks (resolving the imported functions, aligning the PE sections in memory, …) and the *Copy-on-write* protection, mapping a file as data file does not. A mapped data file is typically used to perform read/write operations with the speed advantage of an in-memory file (called mapped file I/O (Yosifovich et al., 2017, p. 405). It is, however, also possible to map a data file with the *EXECUTE_READWRITE* protection and execute code contained in that file. As in this case no *Copy-on-write* is used, a modification to a page will not lead to a new physical page respectively *PFN DB* entry and also not to a change of the *PrototypePte* field (this flag will remain set). So while we ignore pages belonging to mapped image files with the *PrototypePte* flag, we include mapped data files for two reasons: We can't be sure if the pages have been modified and a mapped data file with *EXECUTE* permission is something to look into (we did not find a single benign instance in all our test environments).

4.4. Data Execution Prevention

When DEP is not active for a running process, which is the default for non-essential x86 programs and services on Windows client versions (Yosifovich et al., 2017, p. 320), it can execute code from pages with e.g. *READWRITE* protection. Active MMU PTEs (instances of *_MMPTE_HARDWARE*), however, had the *NX* bit set for non-executable pages during our analysis and hence, a CPU with Hardware *NX* support will not fetch and execute instructions from those pages. The way this still works is as follows: When the Windows Operating System gets an access violation from the CPU for a non-executable page belonging to a process without DEP, it unsets the *NX* bit for that page and the CPU is now able to execute the containing code.

This behavior makes it, with our approach, pretty easy to spot for example code execution triggered by stack buffer overflows. If the shellcode is stored and executed in a page from the stack, only this page is marked as executable (*NX* bit unset) and stands out from otherwise only *READWRITE* pages. The VAD's protection stays unaffected by this behavior and hence has still its initial value (typically *READWRITE* for VADs containing stacks). There is, however, one caveat: When the page is for example paged out, the MMU PTE's (instance of *_MMPTE_SOFTWARE*) protection field is set with its actual protection, which is *READWRITE* and not *EXECUTE_READWRITE* since its protection has not been changed explicitly. The same goes for this page when it is paged in again: The *NX* bit will be set again (because the protection field does not say *EXECUTABLE*), until a new code execution attempt occurs. So in those cases, we can't detect the former executable page.

5. Evaluation

The algorithm described in the previous Section has been implemented as a Rekall (Google Inc, 2018b) plugin (called *ptenum*) and evaluated with Rekall version *1.7.2.rc1* using Python version *3.7* on both, x86 and x86_64 Windows 7 and Windows 10 VMs. The Volatility plugins have been run with Volatility on commit 9df8aa6 (The Volatility Foundation, 2019). As the output of Volatility's and

Rekall's *malfind* didn't differ in our evaluation for the identification of suspicious memory regions, we don't differentiate them in the following sections. Regarding the *hashtest* plugin we used a modified version (Block, 2019) because the one from the author's repository (White, 2013) was not compatible with a more current version of Volatility and had a bug in the interpretation of PTE values. The result of each plugin for a specific executable can be read as follows: The first letter (in capitals) indicates if all processes with injected executable code have been identified and the second letter (in non capitals) indicates if all injected executable memory regions/pages have been identified.

- ***A* or** *a* All processes/pages have been identified.
- ***N* or** *n* None of the processes/pages have been identified.
- ***P* or** *p* At least one process/page has been identified, but not all.
- ***F*** The process has been identified by the plugin as malicious but results from a False Positive.

We also evaluated the code injection techniques with paged out pages (see Section 3). If those results differ from the results without paged out pages, the differing result is also given in Table 1, included in brackets after the original result (e.g. "P/a (**N/n**)"). If no differing value is provided in brackets, the result was the same as without paged out pages.

5.1. Evaluation with code injection PoCs

The executables listed in Table 1 implement the injection techniques described in Section 2.1 and are available in our repository (Block, 2019). Most of them are only slight modifications of the original author's code. Among the executables are also some additions, demonstrating the hiding techniques described in Section 3:

- ***RS*** Implements the Remote Shellcode Injection (Block, 2019).
- ***DEP*** Stores and executes shellcode on the stack (a PoC for the scenario described in Section 4.4).
- ***selfmodify*** This executable modifies its own executable code, serving as a test scenario for *hashtest* and paged out pages.
- ***atombombing*** *AtomBombing* PoC by Liberman (2016).
- ***loadExe*** The Process Hollowing PoC by Keong (2004).
- ***procHollow*** A newer implementation of Process Hollowing by Leitch (2014).
- ***reflectiveDLL*** The Reflective DLL Injection PoC by Fewer (2013).
- ***Gargoyle*** The *Gargoyle* hiding technique PoC by Lospinoso (2017).

An appended _m to an executable name means that it has been modified to initially allocate the memory with *READONLY* protection and afterwards changes it to *EXECUTE_READWRITE*. The additional _h and _a for *Gargoyle* indicates, whether the page containing the shellcode is currently **h**idden (not executable) or **a**ctive (executable).

As can be seen in Table 1, no plugin detected all memory regions containing the injected code. Also our plugin did not detect the memory pages containing the shellcode while they are hidden by *Gargoyle*, which is however the expected behavior since those pages are not executable in these cases. The *P/a* status for *malthfind* and *threadmap* means that they were not always able to detect the injected code, but when, they detected all executable memory regions. When looking for the differences between the original code injectors and our modification, we can see that it was possible to hide from *malfind*, *Psinfo* (except Process Hollowing attacks) and *hashtest*.

The detection rate gets worse when the malicious pages are paged out. In this case, *malfind*, *malthfind* and *hashtest* do report none of the executable pages (only *hashtest* does at least print the memory region, but states that zero executable pages are contained in it) while *Psinfo* does now only report the Process Hollowing related pages. Furthermore, *atombombing* and *Gargoyle_a*/*Gargoyle_m_a* are not detected by any plugin, except *ptenum*. The fact that *ptenum* does not detect *DEP* with paged out pages is, again, the expected result, as the pages are not executable anymore (see Section 4.4).

5.2. Evaluation with malware

The malware samples evaluated in this Section have been picked for their code injection behavior, as they implemented some hiding techniques. Their analysis was done with API monitoring and a before and after comparison of memory dumps. In the following we describe the code injection/hiding specific behavior of each malware sample that was present at the time of the memory dump with a focus on anonymous memory. If not specified otherwise, the allocated memory is private.

- ***Rig Exploit Kit*** (Muhammad et al., 2018; Security, 2018b) Creates two new processes with *EXECUTE_READWRITE* memory regions.
- ***Formbook*** (Jullian, 2018; Security, 2017b) Creates one new process with several *EXECUTE_READWRITE* shared memory regions and one *READWRITE* region but with executable pages in it.

Table 1
Evaluation of Code Injection Detection plugins with Code Injection PoCs (without and with paged out pages; the results for the latter case are given in brackets if the results differ).

	malfind	hollowfind	threadmap	malfofind	Psinfo	malthfind	hashtest	ptenum
RS	A/a (**N/n**)	N/n (**F/n**)	P/a	N/n	A/a (**N/n**)	P/a (**N/n**)	A/a (**N/n**)	A/a
RS_m	N/n	F/n	P/a	N/n	N/n	P/a (**N/n**)	N/n	A/a
DEP	N/n	N/n (**F/n**)	N/n	N/n	F/n	P/a (**N/n**)	N/n	A/a (**N/n**)
selfmodify	–	–	–	–	–	–	A/a (**N/n**)	A/a
atombombing	A/a (**N/n**)	N/n	N/n	N/n	A/a (**F/n**)	N/n	A/a (**N/n**)	A/a
loadExe	A/a (**N/n**)	A/a	P/n	A/a	A/a	P/a (**N/n**)	A/a (**N/n**)	A/a
loadExe_m	N/n	A/a	P/n	A/a	A/a	P/a (**N/n**)	N/n	A/a
procHollow	A/a (**N/n**)	A/a	P/a	A/a	A/a	P/a (**N/n**)	A/a (**N/n**)	A/a
procHollow_m	N/n	A/a	P/a	A/a	A/a	P/a (**N/n**)	N/n	A/a
reflectiveDLL	A/a (**N/n**)	N/n (**F/n**)	P/p	N/n	A/a (**F/n**)	P/p (**N/n**)	A/a (**N/n**)	A/a
reflectiveDLL_m	N/n	N/n (**F/n**)	P/p	N/n	F/n (**N/n**)	P/p (**N/n**)	N/n	A/a
Gargoyle_h	A/a (**N/n**)	N/n	N/n	N/n	A/a (**N/n**)	N/n	A/a (**N/n**)	N/n
Gargoyle_m_h	N/n	N/n	N/n	N/n	N/n	N/n	N/n	N/n
Gargoyle_a	A/a (**N/n**)	N/n	N/n	N/n	A/a (**N/n**)	A/a (**N/n**)	A/a (**N/n**)	A/a
Gargoyle_m_a	N/n	N/n	N/n	N/n	N/n	A/a (**N/n**)	N/n	A/a

Table 2
Evaluation of Code Injection Detection plugins with Malware Samples.

	malfind	hollowfind	threadmap	malfofind	Psinfo	malthfind	hashtest	ptenum
Rig Exploit Kit	A/a	N/n	P/p	N/n	A/a	P/p	A/a	A/a
Formbook	A/p	N/n	N/n	N/n	A/p	N/n	A/p	A/a
Form Grabber	P/p	N/n	P/p	N/n	P/p	P/p	A/p	A/a
Ghostminer	A/p	N/n	N/n	N/n	A/p	N/n	A/p	A/a
Kronos	A/p	A/a	A/p	A/p	A/a	N/n	A/a	A/a
Olympic Destroyer	N/n	N/n	N/n	N/n	F/n	N/n	A/n	A/a

- ***Form Grabber*** (Jullian, 2017; Security, 2017a) Has a *READWRITE* memory region in its own address space with executable pages and allocates a new *EXECUTE_READWRITE* memory region within an existing, benign process.
- ***Ghostminer*** (Aprozper and Bitensky, 2018; Chronicle, 2018a) Creates a new process with several *EXECUTE_READWRITE* and *READWRITE* memory regions, all containing executable pages.
- ***Kronos*** (Chronicle, 2018b; Lechtik, 2018) Creates one new process with a *EXECUTE_WRITECOPY* shared memory region and a *EXECUTE_READWRITE* memory region.
- ***Olympic Destroyer*** (Mercer and Rascagneres, 2018; Security, 2018a) Creates one new process with a *READWRITE* memory region containing one executable page.

As we don't possess the source code for the malware samples and hence were not able to influence the memory allocation process or reliably force the process to page out the injected code (see Section 3), we evaluated the samples only as is.

As can be seen in Table 2, no plugin, except *ptenum*, was able to detect all memory regions containing executable pages created by the malware samples. Especially the executable page of *Olympic Destroyer* was revealed by no other plugin (in particular *malfind* and *hashtest*). Only *hashtest* did at least indicate the containing process, as the malware sample drops a new executable which is not part of *hashtest*'s database.

6. Conclusion and future work

In this work, we demonstrate that it is possible to prevent injected code from being reported by current code injection detection plugins. We introduce a novel approach that is able to detect executable pages despite any intentional (or unintentional) hiding technique described in Section 3. Only *DEP* with paged out pages and *Gargoyle* were successful in hiding from our plugin, but this behavior is expected as the affected pages are not executable in these cases (see Sections 4.4 and 2.1.5). We implemented a Rekall plugin that leverages our introduced approach, which we publicly release alongside with this paper.

Because our plugin reports all executable pages (with the mentioned exclusions), no matter if they are part of a code injection or benign, it can produce a huge amount of data that would need to be investigated. The main problem are modified pages of mapped image files as described in the work by White et al. (2013). As the plugin supports to omit those, it can be used as an improved *malfind* plugin (but would miss code injections in mapped image files). Otherwise, it is not suitable for large processes but can be used for small ones or in a before vs. after comparison. This is why our plugin should be integrated with code injection detection plugins, in particular *hashtest*, in order to strip benign data and improve their results.

As we rely on the paging structures to identify executable pages, our approach does not work if the page tables are paged out and the pagefile is not given. For these cases, a fallback mechanism should be implemented which investigates all VADs, similar to the existing *malfind* plugin. This fallback will, however, again be prone to the hiding techniques described in this work. While it would be possible to enumerate the *PFN DB* in order to gather page protections (see Section 4.1), this will only work for pages in hardware and transition state, as all others have no associated *PFN DB* entry.

References

Aprozper, A., Bitensky, G., 2018. Ghostminer: Cryptomining Malware Goes Fileless [Visited on 22.11.2018]. URL. https://blog.minerva-labs.com/ghostminer-cryptomining-malware-goes-fileless.

Block, F., 2019. The Public Repository Containing the Code and Binaries Used in This Work [Visited on 25.03.2019]. URL. https://github.com/f-block/DFRWS-USA-2019.

Chronicle, 2018. Virustotal - ghostminer sample [Visited on 22.11.2018]. URL. https://www.virustotal.com/#/file/40a507a88ba03b9da3de235c9c0afdfcf7a0473c8704cbb26e16b1b782becd4d/detection.

Chronicle, 2018. Virustotal - kronos sample [Visited on 22.11.2018]. URL. https://www.virustotal.com/#/file/9806d1b664c73712bc029e880543dfa013fdd128dd33682c2cfe5ad24de075b9/detection.

Cohen, M., 2014. Windows Virtual Address Translation and the Pagefile [Visited on 19.12.2018]. URL. http://blog.rekall-forensic.com/2014/10/windows-virtual-address-translation-and.html.

Cohen, M., 2016. Rekall and the Windows Pfn Database [Visited on 19.12.2018]. URL. https://web.archive.org/web/20170906073820/http://blog.rekall-forensic.com/2016/05/.

Countercept, 2018. Gargoyle volatility plugin [Visited on 24.12.2018]. URL. https://github.com/countercept/volatility-plugins/blob/master/gargoyle.py.

Dolan-Gavitt, B., 2007. The vad tree: a process-eye view of physical memory. Digit. Invest. 4, 62–64.

enSilo inc, 2016. Atombombing: Brand New Code Injection for Windows [Visited on 20.09.2018]. URL. https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows.

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

Google Inc, 2018. Rekall memory forensic framework [Visited on 23.09.2018]. URL. http://www.rekall-forensic.com.

Google Inc, 2018. Rekall memory forensic framework - github [Visited on 23.09.2018]. URL. https://github.com/google/rekall.

Google Inc, 2019. Rekall's Malfind Plugin [Visited on 16.01.2019]. URL. https://github.com/google/rekall/blob/master/rekall-core/rekall/plugins/windows/malware/malfind.py.

Hammond, A., 2018. Hunting for Gargoyle Memory Scanning Evasion [Visited on 24.12.2018]. URL. https://www.countercept.com/blog/hunting-for-gargoyle/.

Intel Coporation, 2018. Intel® 64 and ia-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide. Part 1 [Visited on 28.12.2018]. URL. https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf.

Jullian, R., 2017. Analyzing a Form-Grabber Malware [Visited on 22.11.2018]. URL. https://thisissecurity.stormshield.com/2017/09/28/analyzing-form-grabber-malware-targeting-browsers/.

Jullian, R., 2018. In-depth Formbook Malware Analysis - Obfuscation and Process Injection [Visited on 22.11.2018]. URL. https://thisissecurity.stormshield.com/2018/03/29/in-depth-formbook-malware-analysis-obfuscation-and-process-injection/.

Keong, T.C., 2004. Dynamic Forking of Win32 Exe [Visited on 20.09.2018]. URL. https://web.archive.org/web/20070808231220/http://www.security.org.sg/code/loadexe.html.

KSL group, 2017. Threadmap Documentation [Visited on 20.01.2019]. URLhttps://github.com/kslgroup/threadmap/raw/master/threadmap/%20documentation.pdf.

KSL group, 2017. Threadmap Volatility Plugin [Visited on 20.10.2018]. URL. https://github.com/kslgroup/threadmap.

Lechtik, M., 2018. Deep Dive into Upas Kit vs. Kronos [Visited on 22.11.2018]. URL. https://research.checkpoint.com/deep-dive-upas-kit-vs-kronos/.

Leitch, J., 2014. Process Hollowing Poc - Github [Visited on 21.09.2018]. URL. https://github.com/m0n0ph1/Process-Hollowing.

Liberman, T., 2016. Atombombing: Brand New Code Injection for Windows - Github [Visited on 20.09.2018]. URL. https://github.com/BreakingMalwareResearch/atom-bombing.

Ligh, M.H., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons.

Lospinoso, J., 2017. Gargoyle - a Memory Scanning Evasion Technique - Github [Visited on 20.12.2018]. URL. https://github.com/JLospinoso/gargoyle.

Martignetti, E., 2012. What makes it page? sample programs [Visited on 13.01.2019]. URL. http://www.opening-windows.com/wmip/testcode/download/license.html.

Martignetti, E., 2012. What makes it page? Windows 7 (x64) Virtual Memory Manager. CreateSpace Independent Publishing Platform.

Mercer, W., Rascagneres, P., 2018. Olympic Destroyer Takes Aim at Winter Olympics [Visited on 22.11.2018]. URL. https://blog.talosintelligence.com/2018/02/olympic-destroyer.html#more.

Microsoft Corporation, 2018. About Atom Tables [Visited on 09.01.2019]. URL. https://docs.microsoft.com/en-us/windows/desktop/dataxchg/about-atom-tables.

Microsoft Corporation, 2019. Memory Protection Constants [Visited on 19.01.2019]. URL. https://docs.microsoft.com/en-us/windows/desktop/Memory/memory-protection-constants.

Microsoft Corporation, 2019. Setprocessworkingsetsize Function [Visited on 18.01.2019]. URL. https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-setprocessworkingsetsize.

Microsoft Corporation, 2019. Virtualprotectex function [Visited on 09.01.2019]. URL. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366899(v=vs.85).aspx.

Monnappa, K.A., 2016. Detecting malicious processes using psinfo volatility plugin [Visited on 25.10.2018]. URL. https://cysinfo.com/detecting-malicious-processes-psinfo-volatility-plugin/.

Monnappa, K.A., 2016. Hollowfind Volatility Plugin [Visited on 11.10.2018]. URL. https://github.com/monnappa22/HollowFind.

Monnappa, K.A., 2016. Psinfo Volatility Plugin [Visited on 25.10.2018]. URL. https://github.com/monnappa22/Psinfo.

Monnappa, K.A., 2017. Detecting Deceptive Process Hollowing Techniques Using Hollowfind Volatility Plugin [Visited on 09.01.2019]. URL. https://cysinfo.com/detecting-deceptive-hollowing-techniques/.

Muhammad, I., Ahmed, S., Faizan, H., Gardezi, Z., 2018. A Deep Dive into Rig Exploit Kit Delivering Grobios Trojan [Visited on 22.11.2018]. URL. https://www.fireeye.com/blog/threat-research/2018/05/deep-dive-into-rig-exploit-kit-delivering-grobios-trojan.html.

Pshoul, D., 2017. Malfofind Volatility Plugin [Visited on 20.10.2018]. URL. https://github.com/volatilityfoundation/community/blob/master/DimaPshoul/malfofind.py.

Pshoul, D., 2017. Malthfind Volatility Plugin [Visited on 20.10.2018]. URL. https://github.com/volatilityfoundation/community/blob/master/DimaPshoul/malthfind.py.

ReactOS Foundation, 2013. Techwiki:memory protection constants [Visited on 22.11.2018]. URL. https://www.reactos.org/wiki/Techwiki:Memory_Protection_constants.

Russinovich, M.E., Solomon, D.A., Allchin, J., 2005. Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000, vol. 4. Microsoft Press Redmond.

Security, Payload, 2017. Hybrid analysis - form grabber sample [Visited on 22.11.2018]. URL. https://www.hybrid-analysis.com/sample/9cdb1a336d111fd9fc2451f0bdd883f99756da12156f7e59cca9d63c1c1742ce?environmentId.

Security, Payload, 2017. Hybrid analysis - formbook sample [Visited on 22.11.2018]. URL. https://www.hybrid-analysis.com/sample/6e4ec3712cf641a31f4e9e4af7d9d7a84fd7da4cc2875c6aceb9a283ed0330d7?environmentId=100.

Security, Payload, 2018. Hybrid analysis - olympic destroyer sample [Visited on 22.11.2018]. URL. https://www.hybrid-analysis.com/sample/edb1ff2521fb4bf748111f92786d260d40407a2e8463dcd24bb09f908ee13eb9?environmentId=100.

Security, Payload, 2018. Hybrid analysis - rig exploit kit sample [Visited on 22.11.2018]. URL. https://www.hybrid-analysis.com/sample/8b86662ab617d11079f16d95d4d584e8acb4a374b87edf341195ab9e043ed1d2?environmentId=100.

The Volatility Foundation, 2017. Volatility's Malfind Plugin [Visited on 16.01.2019]. URL. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py.

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

White, A., 2013. Hashtest Volatility Plugin [Visited on 16.01.2019]. URL. https://github.com/a-white/Hashtest.

White, A., Schatz, B., Foo, E., 2012. Surveying the user space through user allocations. Digit. Invest. 9. S3–S12, [Visited on 15.01.2019]. URL. https://www.dfrws.org/sites/default/files/session-files/paper-surveying_the_user_space_through_user_allocations.pdf.

White, A., Schatz, B., Foo, E., 2013. Integrity verification of user space code. Digit. Invest. 10, S59–S68.

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