Forensic Science International: Digital Investigation 45 (2023) 301561

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2023 USA - Proceedings of the Twenty Third Annual DFRWS Conference

Windows memory forensics: Identification of (malicious) modifications in memory-mapped image files

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

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

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Detection
Volatility
Hooking
Malware
Injection

ABSTRACT

Memory forensics plays a crucial role for the analysis of sophisticated malware, especially with memory-only variants, and has in the past extended its capabilities for detecting various attacker techniques. Several of these techniques affect the memory of victim processes, partly resulting in suspicious private memory regions, but others affect especially the memory-mapped image files (executables and DLLs). One infamous example are API hooks, which commonly are used to redirect the control flow by modifying a few bytes/instructions in the existing code of memory-mapped image files. Other examples are AMSI and ETW bypasses, which also modify just a few bytes, and Module Stomping which has a larger modification effect. While there are already tools for the detection of modified pages and these attacker techniques, one disadvantage they have in common is the inability to pinpoint the exact modified bytes. Instead, they either report modifications on a page level, which means to present 4096 instead of only 3 bytes that actually have been modified, or they use pattern matching in order to identify malicious traces. In this work, we will show that current detection approaches fail to reliably identify modified image-file pages, and even if not, miss some malicious modifications. We then present our novel approach to reliably detect modified pages and to reveal the exact bytes/instructions that have been modified, while filtering benign modifications. With this work we also release a Volatility 3 plugin named imgmalfind, which implements our approach and reports potentially malicious modifications, enriched with some analysis details.

1. Introduction

Due to the rise of tools and techniques for attacker detection, we naturally also see an increase in techniques to circumvent those throughout the last years. Examples are AMSI and ETW bypasses (S3cur3Th1sSh1t, 2022; Korkos, 2022; Teodorescu et al., 2021; Chester, 2019; Kara-4search, 2021; bats3c, 2020), which effectively deactivate detection functionality, but also Module Stomping (Hammond, 2019a; Orr, 2020c) or Process Hollowing (Monnappa, 2017; Block, 2022b) which try to hide malicious content in allegedly benign DLLs and processes. At the same time, older techniques such as API hooks are up to this day still in use for malicious tasks such as bypasses (bats3c, 2020; Cn33liz, 2020), keylogging and password stealing (Case et al., 2019), but also for benign cases such as backwards compatibility (Case et al., 2019). What these techniques have in common is the modification of memory-mapped image files (MMIFs), in particular DLLs and executables in process space. Since these techniques affect running processes, memory forensics is a crucial discipline for an efficient detection. While there are already tools for the detection of modified MMIF pages (Cohen, 2016; Orr, 2020b; Block, 2022c; Hammond, 2019b) and these attacker techniques (Orr, 2020b; Case et al., 2019; Doniec, 2022; Ligh, 2015), they all have their shortcomings. Especially their approaches for identifying modified pages are not reliable and are prone to miss malicious modifications, in particular because of Windows' memory combining feature. Moreover, some tools rely on pattern matching in order to identify specific techniques and by that, fail to detect malicious modifications in some cases, besides generating false positives.

In this work we will describe in detail why existing approaches for identifying modified pages are not reliable and propose a new technique, which is, based on our evaluation, reliable and immune against memory combining effects. We furthermore present a new approach for identifying the exact modified bytes by comparing the

[*] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.
E-mail address: dfrws-usa-2023@f-block.org.

https://doi.org/10.1016/j.fsidi.2023.301561

MMIF with file data in memory (more precisely the Image Section Object (ISO)), and provide an analysis and filtering algorithm for benign modifications. The resources and version numbers, required to reproduce and verify our results, are either referenced in this work or are part of our online repository (Block, 2023), including our Volatility plugin *imgmalfind* which implements our research results. It should be noted that while in this work, we focus on modifications to executable MMIF pages, our approach is also applicable to non-executable pages (see Section 6.2).

1.1. Related work

In the following, we briefly contrast our approach with related work that either also used files in memory, or focused on attack techniques also covered in this paper.

With the words “standing on the shoulders of giants”, our contribution, on the one hand, wouldn't be possible without the work by Butler and Murdock (2011), Hejazi et al. (2008) and van Baar et al. (van Baar et al., 2008) who analyzed file objects in memory, especially the Image Section Object (ISO) (Butler and Murdock, 2011; Hejazi et al., 2008) which is the basis for this work, and on the other hand builds on the approach taken by White et al. (2013). Similar to our goal, they compare MMIFs with a ground truth, which in their case are not files in memory but the files on the file system, which have to be pre-fetched. Furthermore, their comparison approach includes the creation of a hash database, which is then compared against the hashes for MMIF pages. This has two disadvantages: At the one hand, the hash comparison only returns a *true* for a modification, whereas our approach returns the exact modified bytes. The second disadvantage is related to pre-building the hash database. Since the position of te image file is not known upfront, relocations are ignored for the comparison (White et al., 2013, p. 61) and hence could be exploited by attackers. Another work that analyzed files in memory in order to detect malicious modifications has been done by Harrison (2014). Their approach, however, requires a pre-setup environment for the analysis, so it cannot be applied to a different infected machine, and the comparison is only hash based, so again, no analysis of the exact modified bytes.

Case et al. (2019) presented *hooktracer*, an improved detection for API hooks especially in the sense of false positive reduction in contrast to Volatility's *apihooks* plugin (Ligh, 2015). While one major difference is their focus (both are not designed to detect other modifications such as module stomping), another difference is the approach for detecting inline hooks: They search for specific patterns at the beginning of functions, characteristic for API hooks, which can, however, be subverted. Our approach, on the other hand, identifies the actual modifications no matter what or where they are.

The Rekall project used the detection of modified pages in order to narrow the search for API hooks (Cohen, 2016), but also relied on pattern matching for these pages. Furthermore, the tool *Moneta* (Orr, 2020b), the Volatility plugin *ptemalfind* (Block, 2022c) and the approach described by Hammond (2019b) try to identify modified pages but again, do not analyze them for the specific modifications but only present them as modified. Furthermore, their approaches for detecting modified pages are not reliable as we will show in this work.

Regarding AMSI bypasses, Manna et al. (2022) developed a plugin to detect certain bypasses by analyzing specific members of. NET classes. This form of bypass is not covered by our approach, since it does not result in modifications on executable MMIF pages. Another existing form, which does modify code in MMIFs, is not in the scope of their research but according to them, probably detectable by Volatility's *apihooks* plugin. We will show that this is not the case and that our approach is able to detect this form, including latest variants (Korkos, 2022).

Aside from memory forensics, the live analysis tool *hollows_-hunter* does a similar comparison to our approach (it uses image files from the file system). It is hence also able to detect the exact modified bytes, but it lacks allow-listing capabilities (only allows to ignore certain DLLs at a whole) and support for further analysis. A more detailed comparison is part of our evaluation.

1.2. Contributions

The contributions of this paper are:

- A reliable detection of modified MMIF pages in the context of memory combining.
- Identification of the exact modified bytes for an MMIF and hence, the detection of not only API hooks but also other malicious modifications such as AMSI and ETW bypasses and Module Stomping.
- An analysis and filtering of benign modifications.
- A publicly available Volatility plugin (Block, 2023) for the automatic detection and analysis of such modifications.

1.3. Outline

This work is structured as follows: Section 2 provides a brief description of the malware techniques covered in this paper and depicts the necessary details about Windows' memory management. Here on after, Section 3 presents our novel approach for detecting MMIF modifications and Section 4 describes our analysis of benign modifications, which is evaluated in Section 5 with various malware techniques, other detection tools and a benign system state. We conclude this paper in Section 6 and point out limitations respectively future research directions.

2. Fundamentals

This chapter provides fundamental background knowledge for the rest of this work.

2.1. MMIF modification techniques

This section shortly describes several publicly known techniques used by attackers, which all result in modified executable MMIF pages and in particular their machine code instructions.

API hooking is a technique that typically redirects the execution flow from a victim function to another location and hence allows to read and modify the API's arguments and return values. While this can be realized in various ways (Sikorski and Honig, 2012), we will focus on the inline hooking technique which manipulates the code of a victim function. Other variants of this technique, namely virtual table, IAT and EAT hooking, are not in the scope of this paper as they do not affect executable pages (see also Section 6.2).

The Antimalware Scan Interface (AMSI) offers applications an easy integration with antimalware products. It enables them for example to scan processed data for malicious indicators and is already integrated in Windows components such as PowerShell and Windows Script Host (Microsoft Corporation, 2019). The purpose of AMSI bypasses is to deactivate or bypass this functionality. While there are different approaches to accomplish this (Korkos, 2022; Manna et al., 2022; S3cur3Th1sSh1t, 2022), we will focus on those affecting the code in MMIFs (see Section 5.2). The same applies to Event Tracing for Windows (ETW) bypasses (Teodorescu et al., 2021). ETW provides valuable information for various components in the Windows operating system and is e.g., used by various security tools such as AV and EDR systems, making it an interesting target for attackers.

Module Stomping, also known as DLL Hollowing, on the other hand does not manipulate existing code or functions, but simply overwrites them. It uses an existing or newly loaded DLL as a victim to hide its malicious code and overwrites parts or the whole DLL either with shellcode or another DLL (Hammond, 2019a).

A similar technique is Process Hollowing, which creates a benign victim process in suspended state and before it starts executing, replaces the executable with a malicious one. While the more commonly known approaches for this technique either unmap the victim executable or allocate new memory (Monnappa, 2017), the default approach for one of the earliest implementations of this technique (Keong, 2004) was overwriting the victim executable in memory with the new one. This approach is still feasible (Block, 2022b) and evaluated in this paper.

2.2. Files in memory

An image file can have at least four representations in memory (Butler and Murdock, 2011; Uroz and Rodríguez, 2020):

- Memory-mapped in the process space and described by a Virtual Address Descriptor (VAD).
- A pre-loaded template for image files, going to be mapped in a process space, which is already page aligned and has relocations applied (Uroz and Rodríguez, 2020). It can be accessed through a field called `ImageSectionObject` and will be referred to in this work as Image Section Object (ISO). As long as pages have not been modified, each access to an MMIF page within a given process goes to the corresponding ISO page.
- The `DataSectionObject` on the other hand is, besides potential padding, identical to the file on disk. While it typically is being used for non-image files, we observed this form also to be present for image files in some cases.
- The fourth representation is `SharedCacheMap`, which is the result of Windows' file caching functionality and, based on our observations, also contains data identical to the file on disk.

The last three are normally only accessible from kernel space and hence our candidates as ground truth. Based on our observations, the availability of `DataSectionObject`s and `SharedCacheMap`s is negligible for MMIFs and hence ignored for this work. In the case of the Image Section Object, access to the corresponding pages can be achieved by following the `Subsection` field of an MMIF's VAD. The different PE sections with their varying protections are managed by `_SUBSECTION` structs, while the corresponding memory is accessible via the `SubsectionBase` field (Hejazi et al., 2008, p. 130). In this work, we will call pages within the process address space VAD pages, whereas pages referenced by these `_SUBSECTION` structs are referred to as ISO pages.

2.3. PTEs and the Page Frame Number database

A Page Table Entry (PTE) is part of the translation process from a virtual to a physical address and consists of a 64 bit value, split into bitfields (Intel Corporation, 2022, p. 4-27). A prototype PTE on the other hand is a special type, which is not part of the PTEs used by the MMU for translation processes but are related to `_SUBSECTION` structs and used in the context of shared memory (Martignetti, 2012, pp. 295-300). If either PTE type references data in RAM, it contains a field called Page Frame Number (PFN), which points to a physical page. Depending on the bitfields, a PTE can be in different states, including some that indicate no RAM-resident data. For this paper, we define the states as follows:

- ***HARD***: The PTE is valid and has a corresponding physical page in RAM.
- ***TRANS***: The PTE is not valid and the corresponding physical page might get dropped from RAM, but it is currently still in RAM.
- ***SOFT***: Depending on the bitfield values, this state either indicates a paged out, memory compressed or demand zero page.
- ***SUBSEC***: This is a state only used by prototype PTEs and means in our context that the corresponding ISO page is not accessible from RAM but must be gathered from the image file on the file system.

The Page Frame Number Database (PFN DB) on the other hand is an array of `MMPFN` structs, indexed by the PFN, and manages the physical pages (Yosifovich et al., 2017, p. 425). Every PFN DB entry describes one physical page and contains, among others, two fields, which will be covered in this work: The `PrototypePte` field (accessible via the `u4` member), which is set when the PTE, that describes this physical page, is a prototype PTE (means shared memory) (Cohen, 2016), and the `Modified` field (accessible via the `u3.e1` members), which is set when a page has been modified and its content is not yet backed by a file (Yosifovich et al., 2017, p. 441).

3. Identification of MMIF modifications

This chapter describes our approach for identifying modified data in memory-mapped image files (MMIFs), which essentially consists of identifying modified MMIF pages and determining the exact modified bytes within those pages. While in this work we focus on code in executable pages, the approach described in this chapter is also applicable to non-executable pages.

3.1. Identifying modified pages

There are already three publicly known approaches for identifying modified content for MMIFs, which we will call *PrototypePte* (Cohen, 2016), *QueryWorkingSetEx* (Orr, 2020a) and *ModifiedList* (Hammond, 2019b). The *PrototypePte* approach, which is implemented by the *ptemalfind* (Block, 2022c) Volatility plugin and builds the basis for Rekall's API hook detection (Cohen, 2016), is an approach mainly used in memory forensics (because it requires kernel access) and tests the `MMPFN`'s `PrototypePte` field. Depending on the value of this field, a given physical page belonging to an MMIF is considered either modified or not. The QueryWorkingSetEx approach on the other hand, which is used e.g., by the tool Moneta (Orr, 2020a, 2020b) queries the operating system for information about a given page and again, considers a page either modified or not based on a resulting field called `Shared` (Orr, 2020a). What both have in common is the test for a page being shareable, in contrast to being a private page. Unmodified pages belonging to MMIFs are typically shareable, as they are shared among multiple processes. By utilizing the fact that a modification to a page, belonging to an MMIF, results in a private copy of that page (copy-on-write mechanism), these two approaches are in theory able to reliably detect modifications respectively pages which are not modified.

And this is, in part, also still the case: A page belonging to an MMIF, which is not shareable, can be considered modified and will be detected by both approaches. It turns out, however, the other way around is not true: A page, which is shareable, cannot be considered unmodified. The reason for this is Windows' memory combining feature, which tries to minimize memory consumption by combining pages with the same content into one page. Memory combining can be started explicitly via the *NtSetSystemInformation*

API (Yosifovich et al., 2017, p. 459), but is also started automatically by the OS (in our test VMs with Windows 10 21h1 every couple of minutes). The result of such a combination is one remaining physical page for all corresponding duplicates, shared among all affected processes and works as follows: In the beginning, our victim MMIF page is mapped in two separate processes and the virtual address in both processes points to the same physical page, while the `PrototypePte` and `Shared` field state this physical page is shareable. When an attacker modifies the MMIF page with the same data in both processes, each process gets a private copy for that page, containing the same data. Since these are private copies, they are not shareable and their `PrototypePte` and `Shared` fields are set accordingly. As soon as the memory combination is done, the virtual address for the modified MMIF page in both processes points to the same physical page, which is hence now shared between the two and the `PrototypePte` and `Shared` field for this page state shareable. We tested this mechanism and its effect on the detection approaches with custom modifications, API hooks, AMSI bypasses and Module Stomping (see Section 5.2 for the used tools). In all these cases, without any modifications to the attack tools, we observed automatic memory combinations (invoked by the OS) of modified pages when at least two pages were modified with the same data, resulting in malicious modifications not being detected by ptemalfind or Moneta. For AMSI and most ETW bypasses, duplicate pages are almost unpreventable when targeting at least two processes, since they use static bytes. The same applies to Module Stomping, if the same victim DLL is chosen. For API hooks, on the other hand, it depends on the hook-target's address, which can be random. As it turns out, NetRipper tries to stay near the legitimate DLLs and hence, hooks for the same function share the same target across multiple processes, resulting in duplicate pages. It should be noted that for memory combining to occur, the pages do not have to belong to separate processes but can e.g., also be mapped next to each other in the same process space.

The *ModifiedList* approach is described by Hammond (2019b) and uses the `Modified` field of the `MMPFN` struct (Hammond, 2019c), which indicates that this physical page is part of the list of modified pages. The problem with this approach is similar to the one described above: If this field is set, it correctly indicates a modified page, but an unset `Modified` field does not mean the corresponding page has not been modified. Modified in this context means the page's content is not yet backed by a file, which for the private copy of a modified MMIF page would be the pagefile (Yosifovich et al., 2017, p. 441) (Martignetti, 2012, pp. 192-193). As soon as the content of the modified page has been written to the pagefile, this field is cleared. We verified this by forcing modified and executable pages into SOFT state and afterwards reading them back in. After this process, the `Modified` field is cleared while the page is still modified.

The hint for the solution was found in the book by Martignetti, (2012, p. 352), describing the content of the `MMPFN`'s `OriginalPte` field: “For a mapped file backed page, it stores an `_MMPTE_SUBSECTION` pointing to the subsection which covers the file page mapped by the PTE.” The `OriginalPte` field value is used in the case the physical page of an MMIF is removed from memory, in which case the prototype PTE is replaced with the `OriginalPte`. But as long as the MMIF page is in physical memory and unmodified, the `OriginalPte` stays in SUBSEC state. So, by resolving the `OriginalPte`'s state for an MMIF page, we are able to determine if it is modified. One issue with this approach could have been memory combining, since anonymous shared memory also uses `_SUBSECTION` structs, but this is not the case and the `OriginalPte` stays only in SOFT state.

3.2. Identifying the exact modified bytes

Our approach for identifying the exact modified bytes of an MMIF is a comparison of the VAD page with a ground truth, which in this work is the Image Section Object because it is directly available from RAM, but could also be something else (see Section 6.2). Furthermore, since the ISO's version of the MMIF is already memory mapped and relocated, the comparison between a VAD and ISO page can be done without any pre-adjustments. In order to compare a given VAD page with its ISO pendant, we first gather the PTE for the ISO page by enumerating the `_SUBSECTION` structs, referenced by the VAD's `Subsection` field, and following the `SubsectionBase` pointer to the prototype PTE array. If the PTE is either in HARD or TRANS state, we read the content of the whole referenced page. The final step is a byte-by-byte comparison between each VAD-page byte and ISO-page byte and if e.g., the 10th byte in the VAD page has not the same value as the 10th byte in the ISO page, this VAD byte is marked modified.

3.3. Implementation

We created a Volatility 3 plugin named *imgmalfind* (Block, 2023), which uses a modified version of the *ptemalfind* plugin (Block, 2023) that has mainly been updated with our new approach for identifying modified MMIF pages. For each modified page returned by *ptemalfind*, we gather the corresponding ISO page and compare every byte between the two. The result is split into chunks of contiguous modified bytes, so if there are some modified bytes in the beginning of a page and some at the end, these modifications result in two chunks and each one is interpreted and reported separately. Before reporting a chunk, we apply our filtering logic in order to discard benign modifications as described in Section 4.2. This logic can be extended with additional benign triples of affected processes, modified MMIFs and target MMIFs (process - modified MMIF - target MMIF) by supplying them as a plugin argument, and also accepts wildcards for all three. The current implementation uses for example a process-wildcard for the AVG filters. If a modification is not considered benign, we report it and enrich the output with additional information as can be seen in Listing 1. In this order, Line 1 shows the victim PID and process name, the modified MMIF, the affected PE section, the address of the first modified byte and its nearest function, and the number of modified bytes. Starting with Line 3, the bytes and assembly instructions from the ISO page are shown, which represent our ground truth, and Line 10 and below contain the patched bytes and instructions. If the ISO page is unavailable, the `Original Data` part remains empty and `New Data` is filled with the first 64 bytes of the VAD page while the number of modified bytes is set to `-1`. If the patched bytes contain a hook that redirects into another memory region, the target is analyzed and the results added to the output. In the case of an MMIF, the plugin verifies if the target page has been modified and includes this information in the output. The output following Line 17 shows an example, where the target is an unmodified page of an MMIF. imgmalfind has furthermore a `precontext` and `postcontext` option, which both accept an integer as argument specifying the number of bytes to additionally include in the modification-analysis. Section 4.1 describes one use case for these options.

4. Benign modifications

In this chapter we are describing our analysis of benign modifications identified with *imgmalfind* and present our algorithm to filter these.

4.1. Analysis of benign modifications

During our evaluation of the new approach with benign memory dumps, we came across two major types of benign hooks, which we will discuss in this section. The first type was found in Firefox, Microsoft Edge, Chromium and Chrome processes and every time consisted of the same pattern: First, a `movabs` instruction, storing a memory address in a register and afterwards a jump (`jmp`) instruction to that register. Listing 1 shows an example of such a hook. In most observed cases, these hooks were installed in functions of the `ntdll.dll` DLL and each memory address pointed into the executable of that process (so e.g., into `msedge.exe` as shown in Line 17). As every browser instance results in multiple processes (at least eight) and most of them contained 9 to 17 hooks of this type, the total number of these benign modifications was at least around 320. Based on our analysis, the cause for these hooks is the sandbox functionality by the Chromium project (Mozilla Corporation, 2022a), whose list of hooked APIs matches the APIs we identified as hooked. While Microsoft Edge is also based on Chromium (Microsoft Corporation, 2018), Firefox uses at least some functionality such as the sandbox (Mozilla Corporation, 2022b). We confirmed this by analyzing the Firefox executable with debug symbols and identified corresponding handler functions for the hooked APIs.

```
 1  7184  msedge.exe   \Windows\System32\ntdll.dll  .text
          0x7ff876f4db50    ZwCreateFile + 0x0    16
 2
 3  Original Data
 4
 5  4c 8b d1 b8 55 00 00 00    L...U...
 6  f6 04 25 08 03 fe 7f 01    ..%.....
 7  0x7ff876f4db50: mov r10, rcx
 8  0x7ff876f4db53: mov eax, 0x55
 9
10  New Data
11
12  48 b8 70 96 5e 7f f7 7f    H.p.^...
13  00 00 ff e0 aa aa aa aa    ........
14  0x7ff876f4db50: movabs rax, 0x7ff77f5e9670
15  0x7ff876f4db5a: jmp rax
16
17  Target:
18      The target is an unmodified page. Target VAD:
          \Program Files (x86)\Microsoft\Edge\Application\msedge.exe
19  41 56 56 57 55 53 48 83 AVVWUSH.
20  0x7ff77f5e9670: push    r14
21  0x7ff77f5e9672: push    rsi
```

Listing 1: *imgmalfind* output for benign Browser-Sandbox API Hook

One exception, regarding hooks only in the `ntdll.dll` DLL respectively just pointing to the executable, is Firefox, where hooks have also been installed in functions of the DLLs `kernel32.dll`, `KernelBase.dll` and `user32.dll` and furthermore, some hooks pointed to either Firefox's `mozglue.dll` or `xul.dll` DLL. The other exception are the browsers Edge, Chrome and Chromium. In their case, in exactly one of the processes for each browser (a started Chrome browser for example consists of multiple `chrome.exe` processes in the background), exactly one hook (NtMapViewOfSection in `ntdll.dll`) was pointing to the DLL `chrome_elf.dll` for Chrome and Chromium, respectively to `msedge_elf.dll` for Edge. The cause for this is functionality that prevents block-listed third party DLLs from being loaded into the process space (The Chromium Authors, 2018) by inspecting image files going to be mapped via NtMapViewOfSection. This hook also results in two benign modifications in the `msedge_elf.dll` respectively `chrome_elf.dll` DLL, which themselves are no hooks: A copy of the original NtMapViewOfSection code is stored in the `.crthunk` PE section, which is called if the DLL is not block-listed, and a pointer to it is written into the `.oldntma` PE section. The NtMapViewOfSection-hook in the other processes for the three browsers was pointing to the executable.

The second major type of hooks was found in *Excel* and *Word* processes and is probably also part of further Microsoft Office products. We found them in 10 different DLLs, with the most hooks being installed in `ntdll.dll` (33 hooks), `kernel32.dll` (10 hooks) and `combase.dll` (10 hooks) and an overall total of 142 for both processes. As can be seen in Listing 2, the patched instruction for this type is just a `jmp` to a memory address, which does not belong to the executable or any MMIF, but to a private memory area with `PAGE_EXECUTE_READWRITE` protection, like a common malicious-hook scenario.

```
Original Data

0x7ff9e0f51190: jmp qword ptr [rip + 0x60ee9]
0x7ff9e0f51197: int3

New Data

0x7ff9e0f51190: jmp 0x7ff9a2db1918
```

Listing 2: Benign Office Hook

While the addresses for the `jmp`-targets were differing, the instruction located at the target address was in all cases the same. Listing 3 shows one instance of this instruction in Line 3, which reads a pointer from memory, lying right before the instruction itself (the hex bytes in Line 1), and jumps to it. In all observed cases, the target image file was either `MSO.DLL`, `AppvIsvSubsystems64.dll` or `Mso40UIwin32client.dll`. Those hooks are probably related to Microsoft's Application Virtualization.

```
1  0x7ff9a2db1910: 30 2d 4c 94 f9 7f 00 00
2
3  0x7ff9a2db1918: jmp qword ptr [rip - 0xe]
```

Listing 3: Benign Office Trampoline

One edge case that we encountered for this second hook type are existing jumps that get patched with another jump. As the byte for the `jmp` instruction is not changed, only the jump-target is recognized as modified. An example is shown in Listing 4: Line 2 shows the unmodified bytes and Line 5 the modified bytes, but both without the `jmp` byte. Line 8, on the other hand, shows the modified bytes including the preceding `jmp` byte (`precontext`) and the resulting instruction in Line 9. This is a differentiation to be made clear: On a page level we can spot which page has been written to, but on a byte level we can only identify the bytes that have changed. While this is no problem when it affects a few bytes in the middle of modifications, it can, however, lead to disassembly and filtering problems when it happens in the beginning or end of written bytes. Our plugin has support for such edge cases (see Section 4.2 and 3.3).

```
Original Data
0x7ff9e17ad121: 07 00 00 00 cc cc cc cc

New Data
0x7ff9e17ad121: a3 53 60 c1 cc cc cc cc

New Data with Precontext
0x7ff9e17ad120: e9 a3 53 60 c1 cc cc cc
0x7ff9e17ad120: jmp 0x7ff9a2db24c8
```

Listing 4: Patched `jmp`

We also tested AVG Free Antivirus, which mainly uses hooks of the second type and in all observed cases redirected to DLLs in the AVG directory. Our other results are in essence already covered by Case et al. (2019), except for modifications on AVG's own processes: *AVGUI. exe* maps the DLL `chrome_elf.dll` with the same NtMapViewOfSection-hook related modifications as described above, and all AVG processes seem to bypass their SetUnhandledExceptionFilter API with a `return 0` (see also AMSI bypasses described in Section 5.2).

The last benign modifications that we encountered affect the DLL `clr.dll`. These modifications are no hooks, but mostly array-index changes for the `TlsSlots` array within the `TEB` struct. The original instructions involved are mostly in the form of `mov r11, qword ptr gs:[0x1480]` and the patch changes only the index from `0x1480` to e.g., `0x1590`. One exception are three dummy functions only containing a `jmp` instruction to a `return 0`, which are patched with the instructions `mov rax, qword ptr gs:[0x1590]` and `ret` (but with differing array indexes). Besides the `TlsSlots` related patches we also identified a function called JIT_WriteBarrier that contains placeholder pointers: `0xf0f0f0f0f0f0f0f0`. The modifications to this function do, however, not only affect these pointers, but also the function logic and it seems to be related to. NET's garbage collection (Microsoft Corporation, 2007). A deeper analysis of this function has not been done and it is the last benign modification that we encountered.

4.2. Filtering benign modifications

Despite the described differences, both hook types shared a similar pattern and, moreover, were all in the end pointing to a page in the target DLLs, which was unmodified. In order to filter the benign hooks, we implemented a filter which evaluates the following questions:

- Is the process - modified MMIF - target MMIF combination allow-listed (see Section 3.3)?
- Does the instruction sequence exactly match our known patterns (e.g., `movabs` and then `jmp`)?
- Is the final `jmp` target an unmodified page?
- If it is a modified page, are all modifications allow-listed (happens in rare cases for AVG)?

If the first modified byte has a distance of one or two bytes from the start of a function, the filter algorithm is again run with these bytes included (see Section 4.1). Only if all four tests are answered with *yes*, the modification is discarded. Since our approach does not search for hook patterns but identifies the actual modifications, the variety of hooks to consider is minimal. They hence can be filtered based on an exact instruction flow (in contrast to primarily focusing on jump targets), which e.g., prevents allow-listing a hook that executes malicious code before performing a jump into an allow-listed target. Also, our test for an unmodified page in the hook target prevents an allegedly benign target from being allow-listed, since the target MMIF could have been modified and contain malicious code (e.g., the result of Module Stomping). Based on the paper by Case et al. (2019), this is different to the approach taken by *HookTracer*, which does not seem to verify the integrity of the target DLL and hence might allow-list an API hook leading to a malicious implant within an allegedly benign DLL. It should be noted, however, that since *HookTracer* is not publicly available, we were not able to verify this assumption.

The following modifications are not covered by our generic hook-filtering algorithm and hence implemented as specific filters. For `TlsSlots` patches we first test if the instruction is one of the already known ones and then we verify that the memory reference points inside `TEB`'s `TlsSlots` array. For `.oldntma` on the other hand we verify that it is a pointer to the `.crthunk` PE section and furthermore that `.crthunk` contains the actual NtMapViewOfSection instructions by comparing the modified bytes with the function definition from the ISO page. The AVG related SetUnhandledExceptionFilter patch is only filtered for the AVG processes and this API exactly, and only if the patched instructions match.

5. Evaluation

In this chapter we evaluate our now approach for identifying modified MMIF pages, test our plugin with different MMIF modification techniques, evaluate the remaining benign modifications, compare the results with other detection tools and evaluate the availability of ISO pages.

5.1. Identification of modified pages

We evaluated our approach by first verifying that we identify all pages that are also detected by the other approaches. This has been done with a custom-built Volatility plugin and several memory dumps, enumerating all pages of MMIFs (including non-executable ones) that have either the `PrototypePte` field unset or the `Modified` field set and tested whether these also had an `OriginalPte` not in the SUBSEC state. In order to identify modified pages which are only detected by our approach, we performed the test in a second step the other way around and found tens to thousands of pages with an `OriginalPte` not in the SUBSEC state (a modified page), but with either the `PrototypePte` field set or the `Modified` field unset (both indicating an unmodified page). We also found 2 to 4 instances in most memory dumps where both, `PrototypePte` and `Modified` indicated an unmodified page, while the page was in fact modified. Finally, we tested the correct identification of modified pages after memory combination with custom modifications, API hooks, AMSI bypasses and Module Stomping (see Section 5.2 for the used tools). In summary, our approach detected all pages reported by the other approaches, but also identified modified pages not covered by those.

5.2. MMIF modification techniques

The first test focuses on AMSI and ETW bypasses. We use all AMSI bypasses listed in the Github repository (S3cur3Th1sSh1t, 2022) mentioned by Manna et al. (2022), which result in a modification of MMIFs (numbers 1, 2, 3, 5, 16 and 17) and five publicly available ETW bypasses (Chester, 2020; Chester, 2019; Kara-4search, 2021; Cn33liz, 2020; bats3c, 2020). Most of them basically patch the first bytes of a function with a `return`, as shown in

Listing 5 for the `AmsiScanBuffer` function where the first 3 bytes are replaced by a `return 0`. There are just two exceptions (bats3c, 2020; Cn33liz, 2020), which use a `movabs-jmp` hook instead. All these bypasses are detected by imgmalfind.

```
724  powershell  \Windows\System32\amsi.dll  .text
     0x7ffdc57323e0    AmsiScanBuffer + 0x0   3

Original Data

4c 8b dc 49 89 5b 08 49    L..I.[.I
0x7ffdc57323e0: mov r11, rsp

New Data

31 c0 c3 49 89 5b 08 49    1..I.[.I
0x7ffdc57323e0: xor eax, eax
0x7ffdc57323e2: ret
```

Listing 5: *imgmalfind* output for AMSI Bypass

In the second test we evaluate the detection of API hooking by targeting one Microsoft Edge and one Google Chrome instance with the *NetRipper* project (Popescu, 2022), which intercepts network traffic and encryption related functions. As each browser instance creates multiple processes, in our case a total of 17 processes, the execution of *NetRipper* resulted in 114 inline hooks. Listing 6 shows the output for one of the hooks in one of the Microsoft Edge processes, where the first bytes of the *recv* function have been replaced by a jump to *NetRipper* controlled code. One observed discrepancy between the expected hooks and the ones identified by *imgmalfind* were missing hooks for *EncryptMessage* and *DecryptMessage* within `secur32.dll`. The reason is NetRipper's use of GetProcAddress to resolve these APIs, which returns SealMessage and UnsealMessage from `sspicli.dll` instead and hooks these APIs. All installed hooks were identified by imgmalfind.

```
5244  msedge.exe  \Windows\System32\ws2_32.dll  .text
     0x7fff27fb1d90    recv + 0x0    5

Original Data

48 89 5c 24 08 48 89 74    H.\$.H.t
0x7fff27fb1d90: mov qword ptr [rsp + 8], rbx

New Data

e9 3e f2 fd ff 48 89 74    .>...H.t
0x7fff27fb1d90: jmp 0x7fff27f90fd3
```

Listing 6: *imgmalfind* output for API Hooking

In our third test we have a look at Module Stomping and use the *Phantom DLL hollowing* (Orr, 2020c) project as the injector tool. The injected code creates a MessageBox and is written to the first executable page of the victim DLL. Listing 7 shows an excerpt of the 265 lines long *imgmalfind* output for this injection with several strings in the highlighted ASCII dump. Line 1 reports a total of 270 modified bytes, while the actual shellcode is 276 bytes long. The reason for the discrepancy are 6 bytes distributed throughout the written shellcode, which coincide with the same byte in the original data. One example is highlighted in the hex dump in lines 8 and 16.

```
1  7052  Phantom  \Windows\System32\aadauthhelper.dll .text
         0x7fff1c931000    N/A    270
2
3  Original Data
4
5  cc cc cc cc cc cc cc cc    ........
6  cc cc cc cc cc cc cc cc    ........
7  40 53 48 83 ec 40 4c 8b    @SH..@L.
8  94 24 80 00 00 00 48 8b    .$....H.
9  ...
10
11 New Data
12
13 fc 48 83 e4 f0 eb 17 75    .H.....u
14 73 65 72 33 32 00 48 65    ser32.He
15 6c 6c 6f 20 66 72 6f 6d    llo.from
16 20 45 52 4e 57 00 e8 c8    .ERNW...
17 ...
```

Listing 7: *imgmalfind* output for Module Stomping

The last test uses Process Hollowing with the overwrite approach. The resulting *imgmalfind* plugin output consisted of 270 chunks of modified bytes and a total of 64.627 modified bytes, clearly indicating a major modification of the original data. The discrepancy with the actual size of the written PE file (69.120 bytes) is, besides some bytes again coinciding with the original, mainly due to *imgmalfind*'s current focus on executable pages: The used tool (Block, 2022a) sets protections according to the injected PE file in order to appear benign (the header is e.g., read-only).

5.3. Evaluation on benign system state

In this section we analyze the remaining benign modifications for our plugin and describe potential issues due to missing ISO pages. For this purpose, we set up a Windows 10, version 22h2 VM with 4 GB RAM and Firefox, Microsoft Edge, Chrome, Chromium, cmd, PowerShell, Excel and Word running (exact version numbers are documented in our repository (Block, 2023)). If the ISO pages for corresponding modifications in VAD pages are available, *imgmalfind* filters 628 benign modifications and reports only one modification: The function *JIT_WriteBarrier* in `clr.dll` mentioned in Section 4.1. If, however, certain ISO pages are not memory resident, we are not able to determine the exact modified bytes and hence, the filtering algorithm cannot be applied leading to the whole page being reported for every benign modification (in essence the same result as with ptemalfind and Moneta). During our tests, this happened especially with DLLs not used by many other processes such as `clr.dll`, `chrome_elf.dll` and `msedge_elf.dll`, resulting in 3–35 benign modifications being reported. Section 6.2 discusses several approaches to cope with this issue. The probability for this to happen depends on the OS' overall memory consumption, the number of other processes using the unmodified ISO pages and on the frequency of accesses to them. A corresponding evaluation of this topic is described in Section 5.5. Regarding AVG, we used the same VM, installed AVG and set up the same running processes as described above, which totaled in 1735 filtered benign modifications and again, only the JIT_WriteBarrier modification being reported.

5.4. Comparison with other detection tools

In this section we apply the following detection tools on the previously described cases (where applicable) and compare the results with *apihooks*, *hollowfind*, *ptemalfind*, *hollows_hunter* and *Moneta*. As the *hooktracer* plugin is not yet publicly available, we use the *apihooks* output as a comparison basis since *hooktracer* “consumes the output of apihooks” (Case et al., 2019), while ignoring false positives. The tool by White et al. (2013) has not been included, since it does not support Windows 64bit.

All MMIF modifications described in Section 5.2 were also detected by *ptemalfind*, *hollows_hunter* and *Moneta*. For *ptemalfind* and *Moneta* this is, however, only true as long as modified pages are not combined (see Section 3.1). *hollowfind* on the other hand, failed to detect the overwrite-Process-Hollowing and *apihooks* failed to detect all tested AMSI and ETW bypasses, including those using an actual hook instead of a `return`-patch (even after deactivating apihooks' allow-listing). This means, also hooktracer would currently fail to detect these AMSI and ETW bypasses respectively the `movabs-jmp` hooks in general. Besides that, we tested more ways to successfully circumvent a detection by apihooks such as changing the pattern, overjumping an invalid opcode (apihooks stops when encountering an invalid instruction) or placing the hook after three NOPs (apihooks only analyzes three instructions and then stops). As it would be possible to add more patterns and increase the number of instructions to be analyzed, this also would increase the number of false positives and still could be circumvented by changing the hook pattern. This is the major difference to our approach, which identifies MMIF modifications no matter what or where they are.

Regarding the output details for further analysis, *apihooks* is comparable to *imgmalfind* (except for the focus on the exact modified bytes with *imgmalfind*). *pteamlfind* and *Moneta* are only able to pinpoint on a page level, meaning the analyst has the task of identifying the malicious modification(s), which could be just one byte within at least 4096 bytes of code. The reported number of modified bytes by *imgmalfind*, on the other hand, allows a quick differentiation between a potential API hook or AMSI/ETW patch (typically 1–16 bytes), and a potential Module Stomping or Process Hollowing (hundreds to thousands of bytes). This differentiation is also possible with *hollows_hunter* since it creates so called *tag* files, containing the address and number of modified bytes, and for hooks also the first jump-target, but not more. Listing 8 shows two examples of such *tag* entries, the first being an AMSI bypass and the second an API hook. While *hollows_hunter* also dumps the modified MMIF, a deeper analysis means to load every dumped file with an appropriate tool. Moreover, in the context of API hooks, for every new jump-target, the corresponding memory must be gathered from the running process, if still possible, and again be loaded with an analysis tool.

```
35e0;AmsiScanBuffer;3
9d1d;NtOpenProcess->7ff71fb701a0[7ff71fb70000+1a0:abc.dll:0];c
```

Listing 8: *hollows_hunter* tag file content

Regarding the benign system state described in Section 5.3, both, *hollows_hunter* and *Moneta* report the benign modification mentioned there and the ones from Section 4.1 as potentially malicious because these tools do not, to the best of our knowledge, possess any allow-listing functionality for such modifications. *apihooks* on the other hand reported 39.441 *inline/trampoline* hooks, containing some of the benign modifications but not the `movabs-jmp` hooks. A runtime comparison shows a large gap between apihooks and all other tools: apihooks takes 1:57:20 h to analyze the benign system state (kernel space excluded), imgmalfind takes 4:30 min, ptemalfind 4:19 min, Moneta 2:58 min, and hollows_hunter takes 2:14 min. While these runtimes are not ideal for an accurate comparison (taken in a VM vs. on the host system), they primarily shall give a rough impression on the differences.

5.5. ISO-page availability

In order to evaluate the availability of executable ISO pages, we set up a Windows 10 VM with 2 GB of RAM and five processes. Each process maps a given image file, reads all executable pages of that mapped file to get the corresponding VAD and ISO pages in memory, then modifies each VAD page by changing the first byte of each page (resulting in a private copy) and finally pauses execution without any further access to these pages. After the processes were set up, the system was left alone for 72 h without any interaction or further manual process creations. During this timeframe, we took several memory dumps via the hypervisor (so no interaction within the VM) at these time intervals: After 10 min, 90 min, 3.5 h, 7 h, 24 h, 48 h and 72 h. The memory overall consumption during this test in the VM resided between 52 and 56 percent. For this evaluation, pages in *HARD* and *TRANS* state are considered *available*, since their content can be retrieved from a RAM dump, while pages in *SUBSEC* state are considered *unavailable*. The following list describes the executable ISO pages for all five MMIFs at the beginning of this evaluation, right after reading and modifying all pages:

- **`rdpnano.dll`**: 368 pages in TRANS state, resulting in a total of 368 available pages. This DLL is not mapped by any other process in our VM so we do not expect any process from accessing the ISO pages of this DLL.
- **`ntdll.dll`**: 283 pages in HARD state, resulting in a total of 283 available pages. Since this DLL is also mapped in 86 out of 88 other processes in our VM, we expect accesses to corresponding pages (see Section 2.2).
- **`kernel32.dll`**: 41 pages in HARD and 85 in TRANS, resulting in a total of 126 available pages. Since this DLL is also mapped in 82 out of 88 other processes, we expect accesses to corresponding pages.
- **`mod.exe`**: 8 pages in TRANS state, resulting in a total of 8 available pages. This is a unique executable and only mapped in its own process.
- **`ws2_32.dll`**: 23 in HARD and 37 in TRANS state, resulting in a total of 60 available pages. Since this DLL is also mapped in 48 out of 88 other processes, we expect accesses to corresponding pages.

All executable ISO pages remained *available* during the 72 h, except for 18 `ws2_32.dll` pages, which got dropped between the 48 h and 72 h dump.

After the 72 h we took a VM snapshot and performed three further tests, all starting from that snapshot and again for 72 h with the same intervals for dump creation as described above.

- The test ***t_medium*** simulated a moderate usage scenario (memory wise) by keeping the total memory consumption between 59 and 72 percent, with temporary peaks up until 74 percent but not up to 75.
- In test ***t_high*** we simulated a high-usage scenario by keeping the total memory consumption between 70 and 90 percent, with temporary peaks up until 95 percent.
- The test ***t_purge*** was not designed as a long-term test but was intended to see how low the amount of *available* pages can get. For this purpose, we started every minute a custom program, which solely allocates and writes 100 MB of memory (with a unique pattern to prevent memory savings from memory combining) and re-reads those pages every minute (to prevent these pages from dropping out before others). So in theory, after

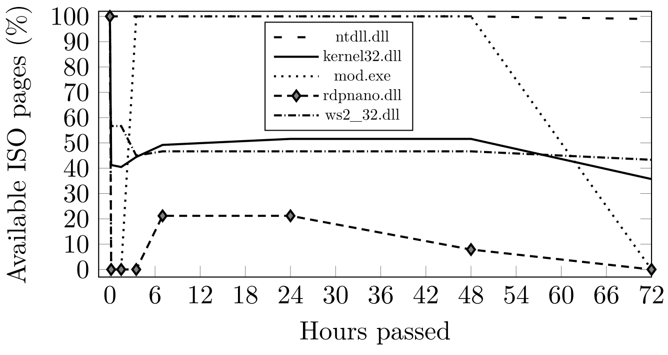

Fig. 1. *t_high* - executable ISO pages.

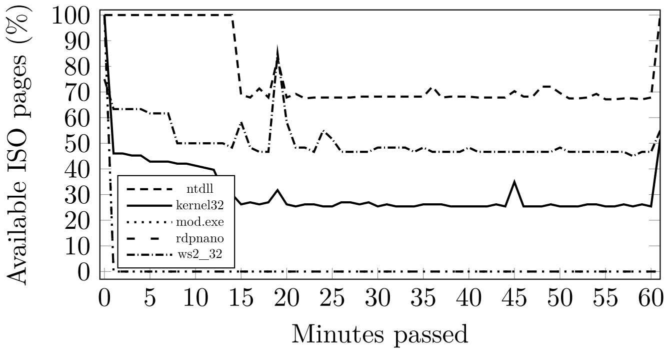

Fig. 2. *t_purge* - executable ISO pages.

10 min there are 10 new processes with a total of 1 GB of allocated memory.

In the case of *t_medium* and *t_high*, we periodically started and stopped instances of *Edge*, *Microsoft Store*, *Notepad* and *cmd* in order to simulate user behavior and to create the desired memory load. The threshold of 75 percent for *t_medium* was chosen during some tests for our snapshotted state, in which we observed rapid page-drops when crossing this mark: All `rdpnano.dll`- and 74 `kernel32.dll`-ISO pages got unavailable. So besides simulating moderate usage, the other goal of the t_medium test was to evaluate how long these pages would stay in memory when not reaching this threshold, while still interacting with the OS and creating new memory consumptions. The result is that the first and only drop occurred with the 48 h dump, where 290 `rdpnano.dll` and 62 `kernel32.dll` ISO pages got unavailable and stayed at this level until the end. All other ISO pages remained available.

The changes for *t_high* were more diverse and are hence illustrated in Fig. 1 (in percent). The *available* ISO pages percentage for both, `rdpnano.dll` and `mod.exe`, dropped to 0% within the first 10 min, but `mod.exe` regained 100% after 3.5 respectively `rdpnano.dll` 21% after 7 h (there was no read triggered by us or the five processes), which again dropped to 0% for both between 48 and 72 h. While the pages for `kernel32.dll` and `ws2_32.dll` mostly remained between 40 and 50%, the available pages for `ntdll.dll` stayed at 100% up until the last memory dump, where 3 pages got unavailable.

After 10 min in *t_purge*, the overall memory consumption reached 99% for the first time and from there on, reached this value every time all processes re-read their pages (with top peaks of 99.7%), followed by a quick drop of most of those VAD pages by the OS, dropping the consumption to 92% until the next process creation and re-read. After 61 min, a last memory dump was taken, and the test stopped because the VM got unresponsive and the screen went black. As can be seen in Fig. 2, all pages for `rdpnano.dll` and `mod.exe` got unavailable after the first minute. The pages of `ntdll.dll`, however, stayed in the first 14 min at 100% and afterwards did not drop below 67%, while `ws2_32.dll` remained around 50% and `kernel32.dll` around 30%. Also interesting to note is the rise of available pages at the end, potentially due to some activity in order to handle the emergency situation.

6. Conclusion and future work

In this work, we demonstrated that by using the Image Section Object, we can pinpoint the exact changes to a memory-mapped image file (MMIF) no matter what and where they are. This allows us to detect not only API hooks but also other MMIF modifications such as AMSI and ETW bypasses, Module Stomping and Process Hollowing. Our approach has, moreover, identified API hooks respectively AMSI and ETW bypasses, which were not detected by Volatility's *apihooks* plugin. Also, the new technique for identifying modified MMIF pages has shown to be more reliable than existing ones, and immune against memory combining effects. We furthermore analyzed benign modifications and proposed a filtering algorithm, which, combined with our approach, is probably more robust against potential subversions than others (due to the lack of a publicly available implementation, we were not able to verify this assumption). Our main advantage is the ability to verify hooks based on an exact instruction flow and the test for an unmodified hook target. Our research results have been implemented in a Volatility 3 plugin that automates the identification of such modifications and filters benign ones.

6.1. Limitations

While our approach identifies modifications in MMIFs, which allows to detect the attacker techniques covered in this work, it is not able to detect variants of these techniques that do not modify executable MMIF pages, such as. NET based AMSI bypasses. Furthermore, if the ISO page for a modification is unavailable, we are not able to identify the exact modified bytes and hence can only report the whole page as modified. Different solutions to handle and resolve this situation are discussed in the next section. A similar case might occur for packed binaries, which get extracted during runtime. Since the Image Section Object does not contain the corresponding unpacked data, *imgmalfind* will report every unpacked page as modified. While we did not encounter such cases in our evaluations, this might happen for some applications. Lastly, for an attacker with kernel space access, it would be possible to subvert our detection approach since he could manipulate the `OriginalPte` value and ISO pages.

6.2. Future work

For cases in which the corresponding ISO page is not available, it is still possible to either use other approaches such as pattern matching in order to identify the malicious modifications, or to include further resources as ground truth. While it would also be possible to compare modified pages against the `DataSectionObject` and `SharedCacheMap`, their availability for MMIFs is, based on our observations, negligible. More promising and reliable (in the sense of availability) are the actual image files, which can be memory-mapped and used as a similar ground truth as the Image Section Object. Besides the file system, an advantageous resource (independent from the accessibility of the file system and more trustworthy than a potentially compromised system) to gather them from are symbol servers, which, at least in the case of Microsoft, not only provide the symbols but also the image files themselves. The range of files available from the symbol servers must, however, be evaluated. While in this work we only focused on executable pages, our approach is also applicable to non-executable memory and might be used to detect IAT, EAT and virtual table hooks.

References

van Baar, R.B., Alink, W., Van Ballegooij, A., 2008. Forensic memory analysis: files mapped in memory. Digit. Invest. 5, S52—S57.

bats3c, 2020. Evtmute - github repository. URL: https://github.com/bats3c/EvtMute/ [Visited on 26.01.2023].

Block, F., 2022a. Process hollowing - github repository. URL: https://github.com/f-block/Process-Hollowing [Visited on 27.01.2023].

Block, F., 2022b. Some experiments with process hollowing. URL: https://insinuator.net/2022/09/some-experiments-with-process-hollowing/ [Visited on 09.01.2023].

Block, F., 2022c. Volatility plugins. URL: https://github.com/f-block/volatility-plugins [Visited on 06.01.2023].

Block, F., 2023. The public repository for this paper. URL: https://github.com/f-block/DFRWS-USA-2023 [Visited on 29.03.2023].

Butler, J., Murdock, J., 2011. Physical Memory Forensics for Files and Cache. Black Hat USA [Link] [Visited on 03.01.2023].

Case, A., Jalalzai, M.M., Firoz-Ul-Amin, M., Maggio, R.D., Ali-Gombe, A., Sun, M., Richard III, G.G., 2019. Hooktracer: a system for automated and accessible api hooks analysis. Digit. Invest. 29, S104—S112.

Chester, A., 2019. Evading sysmon dns monitoring. URL: https://blog.xpnsec.com/evading-sysmon-dns-monitoring/ [Visited on 26.01.2023].

Chester, A., 2020. Etw patching code example. URL: https://gist.github.com/xpn/fabc89c6dc52e038592f3fb9d1374673 [Visited on 12.12.2022].

Cn33liz, 2020. Tamperetw - github repository. URL: https://github.com/outflanknl/TamperETW/ [Visited on 26.01.2023].

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

Doniec, A., 2022. hollows_hunter. URL: https://github.com/hasherezade/hollows_hunter [Visited on 22.11.2022].

Hammond, A., 2019a. Hiding malicious code with "module stomping": Part 1. URL: https://blog.f-secure.com/hiding-malicious-code-with-module-stomping/ [Visited on 27.01.2023].

Hammond, A., 2019b. Hiding malicious code with "module stomping": Part 3. URL: https://blog.f-secure.com/cowspot-real-time-module-stomping-detection/ [Visited on 13.12.2022].

Hammond, A., 2019c. Module stomping - github repository. URL: https://github.com/WithSecureLabs/ModuleStomping/blob/ca714e6c589d7e3fbb226b149ec6ddbcbd7f7bca/driver/driver.c#L352 [Visited on 13.12.2022].

Harrison, C., 2014. Odinn: an In-Vivo Hypervisor-Based Intrusion Detection System for the Cloud. Ph.D. thesis.

Hejazi, S.M., Debbabi, M., Talhi, C., 2008. Automated windows memory file extraction for cyber forensics investigation. J. Digit. Forensic Pract. 2, 117—131.

Intel Corporation, 2022. Volume 3a: System Programming Guide. Intel® 64 and IA-32 Architectures Software Developer's Manual [Link] [Visited on 08.02.2023].

Kara-4search, 2021. Bypassetw_csharp - github repository. URL: https://github.com/Kara-4search/BypassETW_CSharp [Visited on 26.01.2023].

Keong, T.C., 2004. Dynamic forking of win32 exe. URL: https://web.archive.org/web/20070808231220/http://www.security.org.sg/code/loadexe.html [Visited on 24.01.2023].

Korkos, M., 2022. Amsi Unchained. Black Hat Asia [Link] [Visited on 24.01.2023].

Ligh, M.H., 2015. Apihooks Volatility Plugin [Link] [Visited on 22.11.2022].

Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2022. Memory analysis of. net and. net core applications. Forensic Sci. Int.: Digit. Invest. 42.

Martignetti, E., 2012. What Makes it Page? the Windows 7 (X64) Virtual Memory Manager. CreateSpace Independent Publishing Platform.

Microsoft Corporation, 2007. Garbage Collector Basics and Performance Hints [Link] [Visited on 04.02.2023].

Microsoft Corporation, 2018. Microsoft Edge and Chromium Open Source: Our Intent [Link] [Visited on 17.01.2023].

Microsoft Corporation, 2019. Antimalware scan interface (amsi). URL: https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal [Visited on 24.01.2023].

Monnappa, K.A., 2017. Detecting deceptive process hollowing techniques using hollowfind volatility plugin. URL: https://cysinfo.com/detecting-deceptive-hollowing-techniques/ [Visited on 24.01.2023].

Mozilla Corporation, 2022a. Chromium's sandbox. URL: https://source.chromium.org/chromium/chromium/src/+/main:sandbox/win/src/interceptors_64.cc [Visited on 17.01.2023].

Mozilla Corporation, 2022b. Firefox source code - chromium based sandbox. URL: https://searchfox.org/mozilla-central/source/security/sandbox/chromium/sandbox/win/src/interceptors_64.cc [Visited on 17.01.2023].

Orr, F., 2020a. Masking Malicious Memory Artifacts - Part Ii: Blending in with False Positives [Link] [Visited on 15.01.2023].

Orr, F., 2020b. Moneta - github repository. URL: https://github.com/forrest-orr/moneta/tree/4aad531dc7be04d094acfea2e2f66e411366a964 [Visited on 15.01.2023].

Orr, F., 2020c. Phantom dll hollowing - github repository. URL: https://github.com/forrest-orr/phantom-dll-hollower-poc [Visited on 27.01.2023].

Popescu, I., 2022. Netripper - smart traffic sniffing for penetration testers. URL: https://github.com/NytroRST/NetRipper/tree/c763bd0131aab96734f12872ca8451339f1815ed [Visited on 04.01.2023].

S3cur3Th1sSh1t, 2022. Amsi bypass collection. URL: https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell/tree/c3dee519c68711732d784b2618803cd1801c2678 [Visited on 12.01.2023].

Sikorski, M., Honig, A., 2012. Practical Malware Analysis: the Hands-On Guide to Dissecting Malicious Software. no starch press.

Teodorescu, C., Korkin, I., Golchikov, A., 2021. Veni, No Vidi, No Vici: Attacks on Etw Blind Edr Sensors. Black Hat EU [Link [Visited on 26.01.2023].

The Chromium Authors, 2018. Chromium Third Party Dlls [Link] [Visited on 04.02.2023].

Uroz, D., Rodríguez, R.J., 2020. On challenges in verifying trusted executable files in memory forensics. Forensic Sci. Int.: Digit. Invest. 32.

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.