applied sciences

Article

MRCIF: A Memory-Reverse-Based Code Injection Forensics Algorithm

Heyu Zhang, Binglong Li *, Wanpeng Li, Lin Zhu, Chaowen Chang and Shilong Yu

College of Cryptographic Engineering, Information Engineering University, Zhengzhou 450001, China
* Correspondence: lbl2017@163.com

Abstract: The new DLL injection method and its variants can prevent the injected process from calling the common system API to load the injected DLL module so that the malicious module is invisible to the LDR linked list of the process. Traditional injection detection methods have low accuracy in forensic detection of new injection attacks. To solve this problem, this paper proposes a code injection covert memory page detection and forensic detection forensic algorithm based on a memory structure reverse analysis named MRCIF. First, the physical memory pages containing DLL features from the memory image are located, and a sub-algorithm is designed for mapping physical memory space and virtual memory space, thus realizing the reverse reconstruction of the physical page subset corresponding to the DLL code module. Then, in the virtual memory space, the LDR linked list structure of the process is reversely reconstructed, and a reverse reconstruction algorithm of the DLL virtual page subset is developed to reconstruct its virtual space. Finally, a DLL injection covert page detection sub-algorithm is designed based on the physical memory page subset and virtual space page subset. The experimental results indicate that MRCIF achieves an accuracy of 88.89%, which is much higher than that of the traditional DLL module injection detection method, and only MRCIF can accurately detect the Virtual Address Descriptor (VAD) remapping attack.

Keywords: memory forensics; DLL injection; reverse analysis

Citation: Zhang, H.; Li, B.; Li, W.; Zhu, L.; Chang, C.; Yu, S. MRCIF: A Memory-Reverse-Based Code Injection Forensics Algorithm. *Appl. Sci.* **2023**, *13*, 2478. https://doi.org/10.3390/app13042478

Academic Editor: Arcangelo Castiglione

Received: 9 December 2022
Revised: 2 February 2023
Accepted: 11 February 2023
Published: 14 February 2023

1. Introduction

DLL injection is a common attack technique of malware, and a series of new variants have been developed. Different from previous DLL injection methods, the new DLL injection method prevents the victim process from calling the system API to load the malicious DLL module so that there is no information about the injected DLL in the LDR list of the victim process. Meanwhile, the new DLL injection method may tamper with the properties of the Virtual Address Descriptor (VAD), realizing the covert injection of malicious DLL modules. Driver-level injection is difficult to implement because Windows requires drivers to have trusted signatures, so the main target of the new DLL injection is the user-space process, and the injection method is more complicated and hidden. The representative injection methods include the reflective code inject technology proposed by Stephen et al. [1], the Process Hollowing technology proposed by Mieleke et al. [2], and the VAD remapping technology proposed by Palutke et al. [3]. Moreover, some malware uses similar attack techniques, such as Duqu2.0 (2015), Dyre banking Trojan (2015), Conti ransomware (2020), etc., which cause widespread and serious harm [4], so the detection of new types of DLL injection is very important.

Existing security software uses static and dynamic detection techniques and focuses on the detection of disk files and the process of injection, but the software has insufficient detection capability for malicious DLLs in memory after an attack occurs. Detection technology based on memory forensics can solve this problem. Memory forensics can find, extract, and analyze volatile evidence from physical memory and page swap files, and focuses more on the traceability of attacks after they occur. However, the existing memory forensics methods only consider the memory management structure in the virtual memory space, and these methods may be ineffective when the malware tampers with the memory management structure. For example, the VAD remapping method proposed by Palutke et al. [3] can modify the protection attribute and address range of VAD so that the VAD node corresponding to the malicious code page is associated with the benign page. The malfind plugin of Volatility relies on the VAD attribute for detection, so it cannot detect VAD remapping attacks. Srivastava et al. [5] proposed a method based on the combined detection of the thread call stack and VAD attributes, but the method cannot detect injected code modules that do not trigger execution. Block et al. [6] proposed a method to detect hidden executable pages based on PTE attributes. Though the method can detect a variety of new injection attacks, it causes too many false-positive pages, making it difficult to accurately locate malicious memory pages.

This paper proposes a code injection covert memory page detection and forensic algorithm named MRCIF (Memory Reverse based Code Injection Forensics). First, the physical memory pages containing DLL features from the memory image are located, and a sub-algorithm for mapping physical memory space and virtual memory space is designed, thus realizing reverse reconstruction of the physical page subset corresponding to the DLL code module. Then, in the virtual memory space, the LDR linked list structure of the process is reversely reconstructed, and a reverse reconstruction algorithm of the DLL virtual page subset is developed to reconstruct its virtual space. Finally, a DLL injection covert page detection sub-algorithm is designed based on the physical memory page subset and virtual space page subset. The experimental results indicate that MRCIF achieves a higher accuracy than that of the traditional DLL module injection detection method, and only MRCIF can accurately detect the VAD remapping attack. In practice, MRCIF helps to quickly determine the direction of investigation for forensic analysis because of its higher accuracy.

The rest of this paper is organized as follows. Section 2 introduces the related work; Section 3 proposes a code injection forensics detection framework and its sub-algorithms based on the reverse analysis of the memory structure; Section 4 presents the experiments and result discussion and analysis; finally, Section 5 concludes this paper.

2. Related Works

The practice of memory forensics technology started in 2005, and was launched by DFRWS for the Windows system memory forensics analysis challenge [7]. Afterward, memory forensics technology began to develop rapidly. Schuster proposed a pool tag search method to extract processes and threads from pool memory [8]; Dolan-Gavitt developed a method to reconstruct the VAD tree [9]; Kornblum analyzed the PTE structure and proposed a virtual address to physical address translation method [10]. These studies lay the foundation for Windows memory forensics.

In terms of the research on memory reconstruction, Guo et al. proposed a method to reconstruct WinXP system memory based on the KPCR structure [11]; Zhang et al. improved the KPCR method and applied it to the Win7 system [12]. However, with the update of Windows versions, the methods based on pool tag scanning and KPCR scanning are difficult to generalize. To address this issue, Cohen et al. proposed a general Windows memory reconstruction method based on PDB [13], and the memory forensics framework Rekall implemented this method [14].

After being able to reconstruct the basic Windows structure, researchers continue to expand data sources for memory forensics. Cohen proposed a method to extract network connections from the Windows heap [15]; Li et al. proposed a memory fragment file carving algorithm based on the reverse of the structure chain [16]; Zhai et al. proposed a stack trace method that does not rely on process debug symbols [17]. These examples of research on Windows memory structure reconstruction, including heap, stack, and other objects, analyze the relationship between Windows memory management objects, which is the foundation of detecting malware in memory.

Currently, the injection page detection method based on memory forensics mainly focuses on VAD objects (e.g., the malfind plugin of Volatility and Rekall) to detect the label, private state, and protection attribute of VAD. However, the detection conditions of this method are too rough, and it cannot detect malicious code that modifies the VAD protection attributes. Pshoul proposed a method to detect thread injection based on the call stack and developed the malthfind plugin [18]. Srivastava et al. proposed a similar injection detection method based on call stack analysis, which can detect the injection code [5] that modifies the VAD protection attribute, but the hidden injection page that has not been executed cannot be detected based on the stack call.

Considering the limitations of existing VAD and call stack-based detection methods, researchers turned to exploit the characteristics of physical memory pages. Cohen presented a method to match YARA signatures in logically discontinuous physical memory [19], but this method requires input target signature of malware; Block et al. [6] proposed to use the executable attribute of PTE as a feature of executable physical memory pages, but this method will report all modified memory maps, including many benign memory pages, so it cannot accurately locate malicious memory pages.

3. MRCIF Algorithm

In the new DLL injection method, the malware prevents the victim process from calling the system API to load the malicious DLL module so that there is no information about the injected DLL in the load module list (LDR linked list) of the victim process. Our code injection forensic aims to detect covert DLL code modules for new injection attacks from physical memory image files.

The principle of the code injection forensic detection method named MRCIF is as follows. First, the physical memory image file is preprocessed, and the original memory data are read and decomposed into physical memory pages. From a physical point of view, the corresponding physical memory pages are located in the memory image based on the DLL characteristics, and then the mapping between the physical memory space and the virtual memory space is performed. Then, a reverse search for the virtual memory page corresponding to the physical page of the DLL code module is performed. Subsequently, from the perspective of virtual memory space, the LDR linked list structure of the process is reversely reconstructed, and the virtual memory page of the code module is obtained.

Finally, the virtual memory page of the code module reversely searched from the physical page and that of the code module obtained from the LDR are compared, and the hidden virtual memory page of the injected code module is found, as shown in Figure 1.

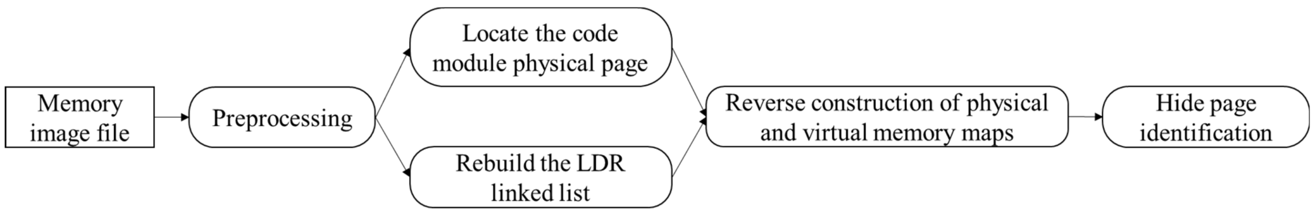

Memory image file
Preprocessing
Locate the code module physical page
Rebuild the LDR linked list
Reverse construction of physical and virtual memory maps
Hide page identification

Figure 1. The flowchart of the proposed method.

A formal description of the method is provided below. Denote a virtual memory page as v_{proc_addr}. A virtual memory page v is uniquely determined by the process and the virtual address. Denote the virtual space of a process as V_{proc}. Denote the subset belonging to the user space as V_{proc_u} and the set of virtual memory pages of the code module in the LDR linked list as V_{proc_l}. Then, the set of virtual memory pages of all processes is $V = \cup_{proc} V_{proc}$. Denote a physical memory page as p_{addr}. A physical memory page p is uniquely determined by its physical address, and all physical memory pages constitute the set of physical memory spaces P. Denote the subset of physical memory pages where the DLL code module is located as P_e. Denote the virtual address to physical address translation as a mapping $t(v) : V \to P$. Further, denote the set of memory pages where the DLL code module is located in the virtual memory as V_e, which is the preimage set of P_e.

Generally, the DLL code module in user space saves information in the LDR linked list of the corresponding process, i.e., the set V_e should satisfy $V_u \cap V_e \subseteq V_l$. When there is covert DLL injection, denote the set of injected DLL module pages as $V_h = \{v|v \in V_u \cap V_e \; and \; v \notin V_l\}$.

According to the expression to detect the covertly injected page set V_h, the following detection method is proposed:

(1) Preprocess the memory image file to obtain the physical memory page set P.
(2) Locate in the physical memory the set of all physical memory pages that contain the DLL header feature P_e.
(3) Reverse the virtual memory space and establish the mapping from the virtual memory space to the physical memory space t.
(4) According to the page map t, traverse each process to obtain V_u and V_l and find the preimage set V_e. Finally, obtain V_h according to the expression of the covertly injected page set V_h.

3.1. File Preprocessing

The purpose of file preprocessing is to read the memory data of the original system from the memory image file. Different memory image formats require different preprocessing approaches. For example, Microsoft's dmp format crash dump file adds metadata to the header, and the rest is the original memory image. The VMware memory snapshot captured by the virtual machine contains two files: the .vmem file is the raw memory data, and the .vmsn file contains the metadata. The memory image obtained by the EnCase forensics tool is in EWF format. The metadata and the compressed original memory data are in the same file, which needs a special tool for parsing.

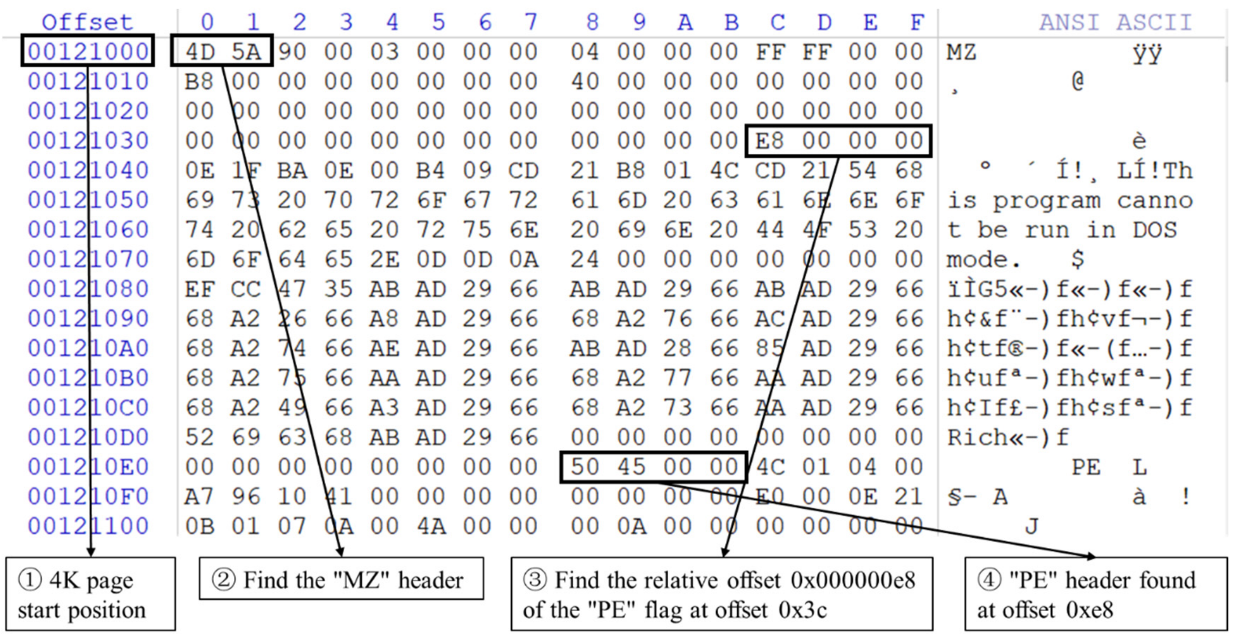
After parsing the memory image file, the original physical memory data are obtained. The operating system generally manages memory pages at the size of 4 KB. Thus, the original memory data can be processed as a set of 4 KB memory pages, and the set of physical memory pages P is formed.

3.2. Physical Locator Sub-Algorithm

The physical locator sub-algorithm is responsible for locating the load address of the code module in physical memory. DLL files are organized in the general format of PE files, whether normally loaded DLLs or covertly injected DLLs. These files need to be applied to memory pages in the virtual memory and loaded in blocks according to the PE file format. Meanwhile, these virtual memory pages must be mapped to physical memory before the DLL module can be executed.

The DLL header code module is less than 4 KB and will be completely loaded at the beginning of a page in memory, and the characteristic string and relative offset remain unchanged. Thus, the PE header is also unchanged in the physical memory space (as shown in Figure 2).

Therefore, a direct search for the PE header character can be performed in the physical memory P to locate the physical memory address where all DLLs are located. The process is shown in detail in Algorithm 1.



```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F   ANSI ASCII
00121000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ          ÿÿ
00121010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ¸       @
00121020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00121030  00 00 00 00 00 00 00 00  00 00 00 00 E8 00 00 00              è
00121040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68    º  ´ Í!¸ LÍ!Th
00121050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
00121060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS
00121070  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode.   $
00121080  EF CC 47 35 AB AD 29 66  AB AD 29 66 AB AD 29 66  ïÌG5«-)f«-)f«-)f
00121090  68 A2 26 66 A8 AD 29 66  68 A2 76 66 AC AD 29 66  h¢&f¨-)fh¢vf¬-)f
001210A0  68 A2 74 66 AE AD 29 66  AB AD 28 66 85 AD 29 66  h¢tf®-)f«-(f…-)f
001210B0  68 A2 75 66 AA AD 29 66  68 A2 77 66 AA AD 29 66  h¢ufª-)fh¢wfª-)f
001210C0  68 A2 49 66 A3 AD 29 66  68 A2 73 66 AA AD 29 66  h¢If£-)fh¢sfª-)f
001210D0  52 69 63 68 AB AD 29 66  00 00 00 00 00 00 00 00  Rich«-)f
001210E0  00 00 00 00 00 00 00 00  50 45 00 00 4C 01 04 00          PE  L
001210F0  A7 96 10 41 00 00 00 00  00 00 00 00 E0 00 0E 21  §– A        à !
00121100  0B 01 07 0A 00 4A 00 00  00 0A 00 00 00 00 00 00       J
```

① 4K page start position
② Find the "MZ" header
③ Find the relative offset 0x000000e8 of the "PE" flag at offset 0x3c
④ "PE" header found at offset 0xe8

Figure 2. Code module physical locator algorithm.

Algorithm 1: Code Module Physical Locator Algorithm

```
Input: Physical memory page set $P$
Output: DLL code module physical memory page set $P_e$
Init: $P_e \leftarrow \varnothing$
for each $p \in P$ do:
    if is PEheader ($p$) = True then
        $P_e \leftarrow P_e \cup \{p\}$
end

Function is PEheader ($p$):
    if $p$[0,1] = 4d5ah then
        e_lfanew ← $p$[0x3c,...,0x3f]
        if $p$[e_lfanew, ..., e_lfanew + 4] = 0x50450000 then
            return True
        else return False
```

3.3. Virtual Space Reverse Reconstruction Sub-Algorithm

The user-space process and its load module management structure, i.e., LDR linked list, exist in the virtual memory space. Modern operating systems generally use the Address Space Layout Randomization (ASLR) mechanism, so the physical memory pages are logically discontinuous. To access the virtual address, the system lookups at the page table of the process to find the physical address to access the data. The lookup mechanism of the page table is proposed by Russinovich et al. [20]. In reverse analysis and reconstruction of the image, it is necessary to find the EPROCESS structure of the process, read the physical-address DTB of the page table (as shown in Figure 3), and then construct the mapping t from virtual pages to physical pages from the memory page table.

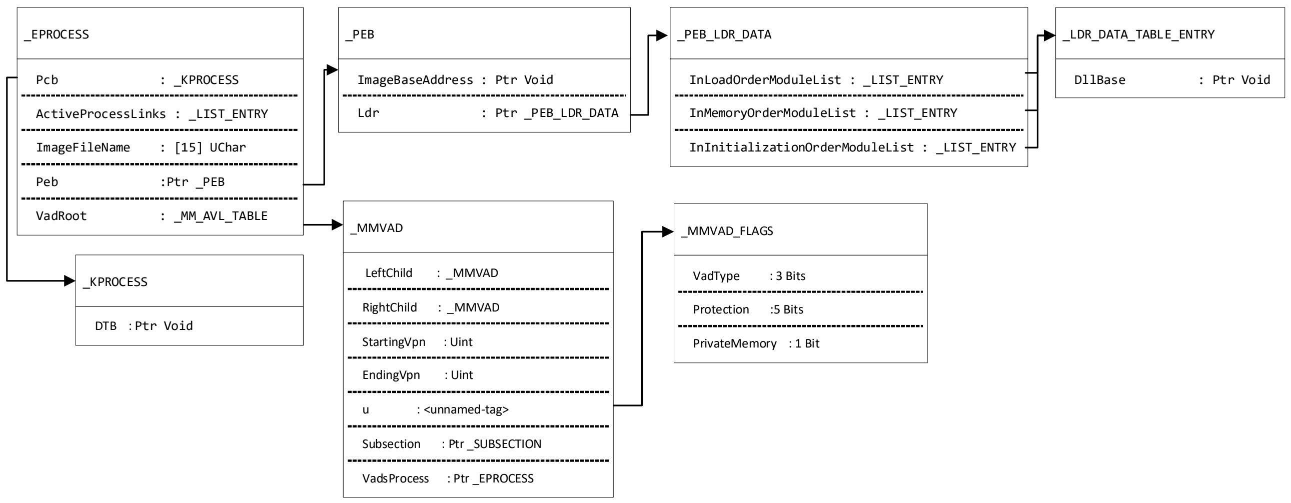


Figure 3. The process management structure of the Windows system. Only the variables involved in this paper are listed.

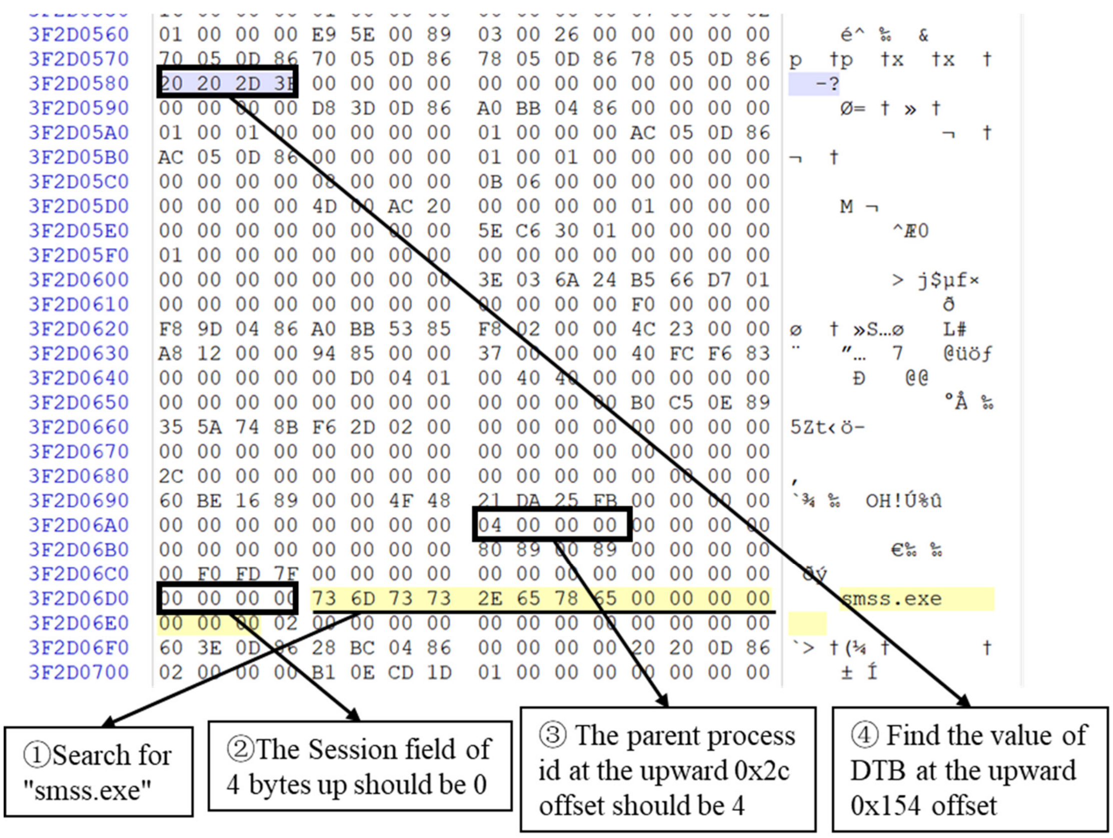
According to Cohen's work [13], the structure of each version of the Windows kernel is almost the same. There are only 10 structural layouts of EPROCESS from WinXP to Win8.1. The structure of EPROCESS is easy to exhaust, so a specific process name can be searched for to locate EPROCESS. Taking the system process "smss.exe" as an example, the search steps are as follows:

(1) The ImageFileName field in EPROCESS indicates the process name, and it has a length of at least 15 bytes(as shown in Figure 3). This work uses '\0' to fill the end of the process name string "smss.exe" to 15 bytes to obtain the hexadecimal character string "73 6d 73 73 2e 65 78 65 00 00 00 00 00 00 00". Then, this string is searched in the physical memory page set P.

(2) If the operating system version is known, the EPROCESS structure is uniquely determined; otherwise, all possible EPROCESS structures are constructed for each search result. Figure 4 shows an EPROCESS structure constructed for a process name search result when the operating system is assumed to be 32-bit Win7.

(3) For each built EPROCESS structure, the system version assumption is verified via the _KUSER_SHARED_DATA structure. The virtual address of _KUSER_SHARED_DATA is 7ffe0000 in each system version, and must correspond to a physical memory page. The NtMajorVersion and NtMinorVersion fields represent the major and minor versions of the operating system, and the two values should correspond to the assumed operating system version.

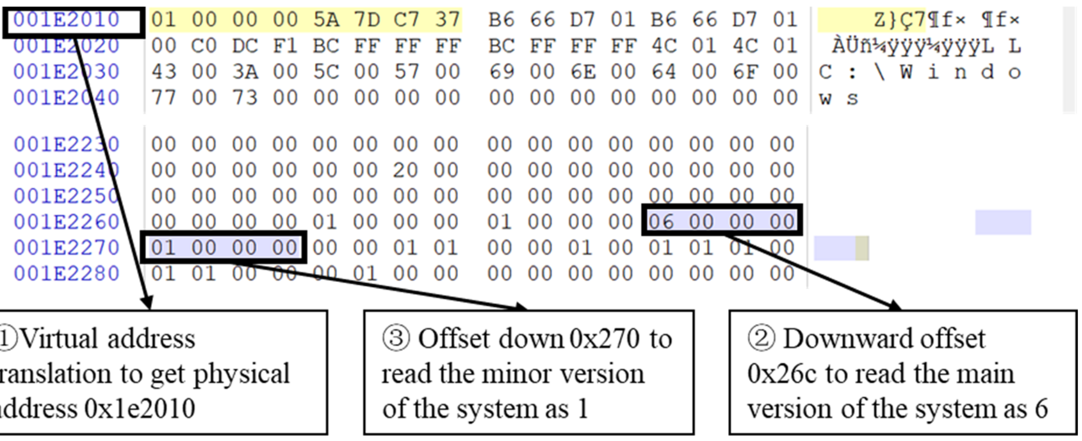
Assuming that the operating system is 32-bit Win7, an EPROCESS structure is constructed for the search result of a process name (as shown in Figure 4). Then, the DTB is read, and the virtual address 0x7ffe0000 of _KUSER_SHARED_DATA is converted to the physical address 0x1e2010 according to the page table conversion method of the 32-bit system (as shown in Figure 5). Since the values of the NtMajorVersion and NtMinorVersion fields are 6 and 1, the system kernel version is 6.1, which is consistent with 32-bit Win7. Thus, it can be determined that the built EPROCESS structure is correct. Then, the mapping $t(v_{smss}) : V_{smss} \to P_{smss}$ between all virtual pages in the smss process space to physical pages can be constructed.



```
3F2D0560  01 00 00 00 E9 5E 00 89  03 00 26 00 00 00 00 00       é^ ‰  &
3F2D0570  70 05 0D 86 70 05 0D 86  78 05 0D 86 78 05 0D 86   p  †p  †x  †x  †
3F2D0580  20 20 2D 3F 00 00 00 00  00 00 00 00 00 00 00 00     -?
3F2D0590  00 00 00 00 D8 3D 0D 86  A0 BB 04 86 00 00 00 00       Ø= † » †
3F2D05A0  01 00 01 00 00 00 00 00  01 00 00 00 AC 05 0D 86              ¬  †
3F2D05B0  AC 05 0D 86 00 00 00 00  01 00 01 00 00 00 00 00   ¬  †
3F2D05C0  00 00 00 00 08 00 00 00  0B 06 00 00 00 00 00 00
3F2D05D0  00 00 00 00 4D 00 AC 20  00 00 00 00 01 00 00 00       M ¬
3F2D05E0  00 00 00 00 00 00 00 00  5E C6 30 01 00 00 00 00           ^Æ0
3F2D05F0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
3F2D0600  00 00 00 00 00 00 00 00  3E 03 6A 24 B5 66 D7 01           > j$µf×
3F2D0610  00 00 00 00 00 00 00 00  00 00 00 00 F0 00 00 00               ð
3F2D0620  F8 9D 04 86 A0 BB 53 85  F8 02 00 00 4C 23 00 00   ø  † »S…ø   L#
3F2D0630  A8 12 00 00 94 85 00 00  37 00 00 00 40 FC F6 83   ¨   ”…  7   @üöƒ
3F2D0640  00 00 00 00 00 D0 04 01  00 40 40 00 00 00 00 00        Ð   @@
3F2D0650  00 00 00 00 00 00 00 00  00 00 00 00 B0 C5 0E 89               °Å ‰
3F2D0660  35 5A 74 8B F6 2D 02 00  00 00 00 00 00 00 00 00   5Zt‹ö-
3F2D0670  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
3F2D0680  2C 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ,
3F2D0690  60 BE 16 89 00 00 4F 48  21 DA 25 FB 00 00 00 00   `¾ ‰  OH!Ú%û
3F2D06A0  00 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00
3F2D06B0  00 00 00 00 00 00 00 00  80 89 00 89 00 00 00 00           €‰ ‰
3F2D06C0  00 F0 FD 7F 00 00 00 00  00 00 00 00 00 00 00 00    ðý
3F2D06D0  00 00 00 00 73 6D 73 73  2E 65 78 65 00 00 00 00       smss.exe
3F2D06E0  00 00 00 02 00 00 00 00  00 00 00 00 00 00 00 00
3F2D06F0  60 3E 0D 86 28 BC 04 86  00 00 00 00 20 20 0D 86   `> †(¼ †        †
3F2D0700  02 00 00 00 B1 0E CD 1D  01 00 00 00 00 00 00 00       ± Í
```

①Search for "smss.exe"
②The Session field of 4 bytes up should be 0
③ The parent process id at the upward 0x2c offset should be 4
④ Find the value of DTB at the upward 0x154 offset

Figure 4. Building the EPROCESS structure in a 32-bit Win7 system.

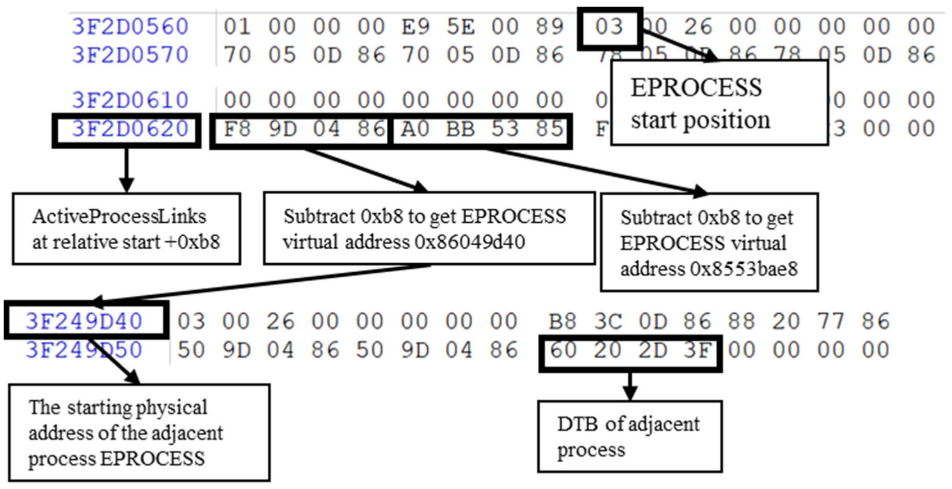


```
001E2010  01 00 00 00 5A 7D C7 37  B6 66 D7 01 B6 66 D7 01       Z}Ç7¶f× ¶f×
001E2020  00 C0 DC F1 BC FF FF FF  BC FF FF FF 4C 01 4C 01    ÀÜñ¼ÿÿÿ¼ÿÿÿL L
001E2030  43 00 3A 00 5C 00 57 00  69 00 6E 00 64 00 6F 00   C : \ W i n d o
001E2040  77 00 73 00 00 00 00 00  00 00 00 00 00 00 00 00   w s

001E2230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
001E2240  00 00 00 00 00 00 20 00  00 00 00 00 00 00 00 00
001E2250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
001E2260  00 00 00 00 01 00 00 00  01 00 00 00 06 00 00 00
001E2270  01 00 00 00 00 00 01 01  00 00 01 00 01 01 01 00
001E2280  01 01 00 00 00 01 00 00  00 00 00 00 00 00 00 00
```

①Virtual address translation to get physical address 0x1e2010
③ Offset down 0x270 to read the minor version of the system as 1
② Downward offset 0x26c to read the main version of the system as 6

Figure 5. Accessing the _KUSER_SHARED_DATA structure to verify the system version assumption.

Next, the map $t(v) : V \to P$ is established through all processes. The ActiveProcessLinks of the EPROCESS structure is a doubly linked list containing two virtual address pointers that point to the ActiveProcessLinks addresses of two adjacent EPROCESSs. As shown in Figure 6, 0x3f2d0568 is the starting physical address of the smss process. The virtual addresses of two adjacent EPROCESS structures can be obtained by reading the virtual addresses of the two linked list items before and after the ActiveProcessLinks linked list and subtracting the offset of the relative starting position. The physical address corresponding to the virtual address 0x86049d40 obtained from the mapping $t(v_{smss}) : V_{smss} \to P_{smss}$ is 0x3f249d40. Based on this, a new EPROCESS structure is built, and it is known that the process is "csrss" from its ImageFileName field. Meanwhile, the DTB of the process is 0x3f2d2060, according to which the address mapping $t(v_{csrss}) : V_{csrss} \to P_{csrss}$ can be established. The above steps are executed iteratively until the EPROCESS of all processes is traversed; finally, the mapping $t(v) : V \to P$ from virtual pages to physical pages of all processes is obtained. Taking InInitializationOrderModuleList as an example, the offset of the linked list in the LDR is 0xc, and the virtual addresses of two adjacent linked list entries are 0x3b1790 and 0x3b1810, respectively.

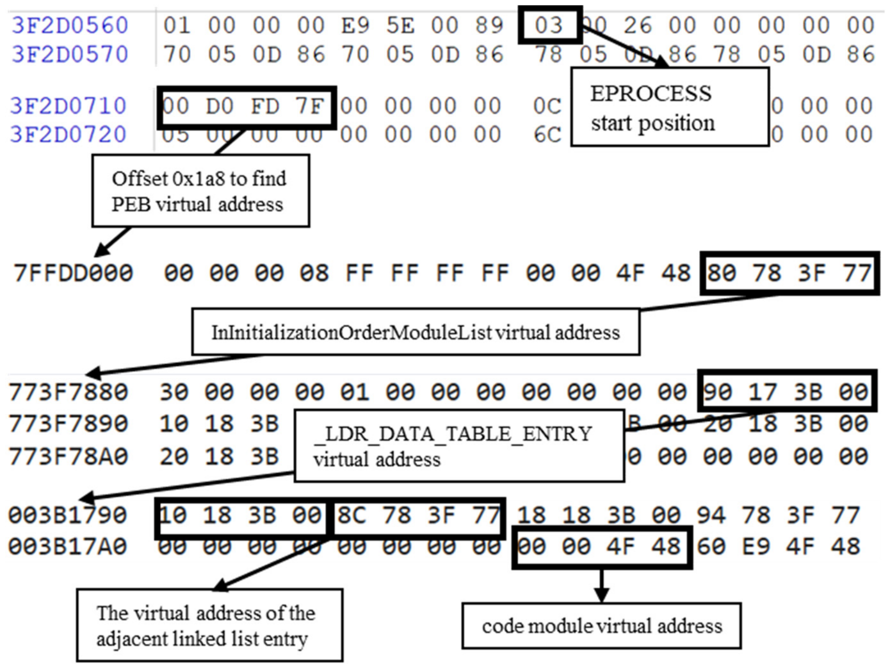


```
3F2D0560  01 00 00 00 E9 5E 00 89 03 00 26 00 00 00 00 00
3F2D0570  70 05 0D 86 70 05 0D 86 78 05 0D 86 78 05 0D 86

3F2D0610  00 00 00 00 00 00 00 00 0                0 00 00
3F2D0620  F8 9D 04 86 A0 BB 53 85 F                3 00 00

3F249D40  03 00 26 00 00 00 00 00  B8 3C 0D 86 88 20 77 86
3F249D50  50 9D 04 86 50 9D 04 86  60 20 2D 3F 00 00 00 00
```

EPROCESS start position
ActiveProcessLinks at relative start +0xb8
Subtract 0xb8 to get EPROCESS virtual address 0x86049d40
Subtract 0xb8 to get EPROCESS virtual address 0x8553bae8
The starting physical address of the adjacent process EPROCESS
DTB of adjacent process

Figure 6. Accessing adjacent processes through ActiveProcessLinks in a 32-bit Win7 system.

Then, build the _LDR_DATA_TABLE_ENTRY structure at virtual address 0x3b1790, and the virtual address of the code module is read to be 0x484f0000, $v_{smss\_0x484f0000} \in V_{smss\_l}$. The beginning of the _LDR_DATA_TABLE_ENTRY structure is the virtual address of two adjacent linked list items, so the InInitializationOrderModuleList linked list can be traversed. The structures of the other two linked lists in the LDR have the same structure, and the address of the load module can be traversed and read in the same way (as shown in Figure 7). Finally, V_{smss_l} is obtained, and the LDR linked list is reconstructed for each process to obtain $V_l = \cup_{proc} V_{proc\_l}$.



```
3F2D0560  01 00 00 00 E9 5E 00 89 03 00 26 00 00 00 00 00
3F2D0570  70 05 0D 86 70 05 0D 86 78 05 0D 86 78 05 0D 86

3F2D0710  00 D0 FD 7F 00 00 00 00 0C               0 00 00
3F2D0720  05 00 00 00 00 00 00 00 6C               0 00 00

7FFDD000  00 00 00 08 FF FF FF FF 00 00 4F 48 80 78 3F 77

773F7880  30 00 00 00 01 00 00 00 00 00 00 00 90 17 3B 00
773F7890  10 18 3B                B 00 20 18 3B 00
773F78A0  20 18 3B                0 00 00 00 00 00

003B1790  10 18 3B 00 8C 78 3F 77 18 18 3B 00 94 78 3F 77
003B17A0  00 00 00 00 00 00 00 00 00 00 4F 48 60 E9 4F 48
```

EPROCESS start position
Offset 0x1a8 to find PEB virtual address
InInitializationOrderModuleList virtual address
_LDR_DATA_TABLE_ENTRY virtual address
The virtual address of the adjacent linked list entry
code module virtual address

Figure 7. Find the virtual address of the code module in the LDR linked list in the 32-bit Win7 system.

3.4. Code Injection Covert Page Identification Sub-Algorithm

Hidden page identification aims to detect the virtual page set V_h of the hidden code module in the user space and find the set V_e containing the memory pages where the DLL code module is located in the virtual memory.

The user-space address range of a process in a 32-bit system is 0~0x7fffffff, and in a 64-bit system, it is 0~0x7fffffffff. According to the address range, the user space page set of the smss process is obtained as $V_{smss\_u} = \{v_{smss\_addr} | addr < 0x7fffffff\}$. The user space of other processes is obtained in a similar way. The set of user-space memory pages of all processes is represented as $V_u = \cup_{proc} V_{proc_u}$.

To reduce physical memory usage by the operating system, when the same module is used by multiple processes, multiple virtual memory pages are mapped to the same physical memory page p. So, there is no inverse mapping for $t(v) : V \to P$. Therefore, to find the preimage of the physical memory page p, we can only traverse all the virtual memory pages v and perform the mapping $t(v)$ to convert the pages. In this way, the preimage set V_e of P_e, $V_e = \{v | t(v) \in P_e\}$ can be obtained.

The set P_e contains a part of the driver code module because the header characteristics of the driver module file are the same as that of the DLL file. However, the driver modules are loaded in the kernel space of virtual memory, and they do not appear in the LDR linked list. These pages where these driver modules are located need to be excluded from the detection, so the set of user-space code module virtual memory pages $V_u \cap V_e$ is obtained. Finally, whether each virtual page appears in the set V_l is judged, and if a virtual page is not in V_l, the virtual page is a covertly injected page. The whole covert page detection process is shown in Algorithm 2.

Algorithm 2: Covert page detection sub-algorithm

```
Input: Virtual memory page set $V$, Code module page set $V_l$ in LDR linked list, Code
module physical memory page set $P_e$
Output: Covert code module virtual page set $V_h$
Init: $V_h \leftarrow \varnothing$
for each $V_{proc} \subset V$ do:
    for each $v \in V_{proc}$ do:
        if $t(v) \in P_e$ and $v \in V_{proc\_u}$ and $v \notin V_{proc\_l}$ then:
            $V_{proc\_h} \leftarrow V_{proc\_h} \cup \{v\}$
    end
end
$V_h = \cup_{proc} V_{proc_h}$
```

4. Experiment and Discussion

In this section, we first introduce the samples and experimental setup, and then we conducted a series of experiments to compare with the current commonly used methods.

4.1. Experiment Samples and Setup

The MRCIF algorithm has been implemented using Python 3.8 and some functions of volatility have been called, regardless of the operating system. The MRCIF algorithm was evaluated on 32-bit WindowXP, 32-bit Windows 7, and 64-bit Windows10 snapshots, using a machine with Intel i7-7700K and 16 GB RAM.

The experiment samples are shown in Table 1, including the source of the samples and the version and size of the system memory image.

Table 1. Experimental samples.

No	Sample	Memory Image
1	Process Hollowing	https://github.com/m0n0ph1/Process-Hollowing/tree/master/executables/ProcessHollowing.exe (accessed on 15 November 2021). Win7SP0x86, 1 GB
2	Reflective DLL (Win7)	https://github.com/stephenfewer/ReflectiveDLLInjection/tree/master/bin/inject.exe (accessed on 15 November 2021). Win7SP0x86, 1 GB
3	Reflective DLL (Win10)	Sample 2, Win10x64_19042, 2 GB
4	VAD Remapping	According to the method implementation of Palutke et al. [3], the default VAD permissions and injection files are modified. Win7SP1x86, 1 GB
5	Spyeye (Win7)	https://s.threatbook.com/report/file/f097ad77b99b3744994a646d6a3577cea2faa8b9e656fcccbbd73244e227c850. (accessed on 15 May 2022) Win7SP1x86, 1 GB
6	Spyeye (Win10)	Sample 4, Win10x64_19042, 2 GB
7	Cridex	https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples. (accessed on 15 March 2021) WinXPSP3x86, 512 M
8	Zeus	https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples. (accessed on 15 March 2021) WinXPSP3x86, 512 M
9	Coreflood	https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples. (accessed on 15 March 2021) WinXPSP3x86, 512 M

4.2. Experiment Results and Discussion

For the above test samples, the method proposed in this paper is compared with the commonly used detection methods, and the results are presented in Tables 2–4. The process accuracy is calculated as the number of correctly reported injection processes divided by the total number of reported processes; the page accuracy rate is calculated as the number of correctly reported injected pages divided by the total number of reported pages. The target of malthfind is the thread, so this work uses the page where the calling code module is located as the hidden page of its report.

Table 2. Process detection accuracy.

Sample No	1	2	3	4	5	6	7	8	9	Average
MRCIF	100%	100%	100%	100%	100%	100%	100%	100%	0	88.89%
malfind	25%	33.3%	50%	0%	42.9%	96.15%	96%	50%	25%	46.48%
malthfind	0	100%	\	0%	60%	\	100%	100%	60%	60%
Ptemalfind	3.9%	3.9%	25%	3.9%	8.6%	89.28%	\	\	\	22.43%

Table 3. Page detection accuracy.

Sample No	1	2	3	4	5	6	7	8	9	Average
MRCIF	100%	100%	100%	100%	100%	100%	100%	100%	0	88.89%
malfind	20%	25%	25%	0%	40%	90.74%	79%	17%	11.10%	34.20%
malthfind	0%	100%	╲	0%	60%	╲	100%	100%	100%	65.71%
Ptemalfind	0.1%	0.1%	6.45%	0.1%	0.2%	64.2%	╲	╲	╲	11.86%

Table 4. Time consumption (second).

Sample No	1	2	3	4	5	6	7	8	9	Average
MRCIF	1915	1523	7769	1756	1884	7483	633	651	628	2693.6
malfind	10	9	59	9	15	67	8	10	9	21.8
malthfind	1433	1762	╲	1528	1607	╲	997	1054	1158	1362.7
Ptemalfind	393	377	190	387	353	263	╲	╲	╲	327.2

Note that in Sample 2 and 3, Reflective DLL will first read the payload DLL file as data directly into the memory during injection, perform this operation again on the memory space, and remap the payload code module in the executable code mode. So, there are two injection modules: one is the file data page, and the other is the executable memory page. Since injected data pages are usually freed, only executable memory pages are counted as detection target pages in the accuracy rate.

As shown in Tables 2 and 3, the malfind method only detects the private attributes and protection attributes of VAD nodes, but the detection conditions are rough, which results in low accuracy.

The malthfind method discovers unknown calling modules by reconstructing the call stack. It has high accuracy in detecting malicious injected pages, such as samples 2, 5, 6, and 7, but it cannot detect potential injected pages that are not executed. As for samples 3 and 4, some of the injected pages are not executed, and there is no stack, so the malthfind method cannot detect these pages. In samples 3 and 6, malthfind fails to rebuild the stack on Windows10 because of the lack of the necessary objects in Volatility profile.

Ptemalfind only achieved slightly better results in the experiment of Windows10, such as Sample 3 and 6. It cannot run on samples 5, 6, and 7 because the PTE structure of the WinXP lacks the properties required for Ptemalfind detection. Meanwhile, this method can detect almost all the memory pages injected by the method, but its accuracy is extremely low, which makes it difficult to apply in practice.

In comparison, MRCIF can completely detect the hidden injected pages for samples 1~8 without false positives. Especially in sample 4, only MRCIF can detect the injected code module with high accuracy. The memory structures used in MRCIF, such as EPROCESS, LDR linked list, and PTE, are all necessary structures provided in Microsoft PDB files. Therefore, compared with malthfind and Ptemalfind, MRCIF is more widely applicable in different versions of Windows with higher accuracy. For sample 9, Coreflood's injection code module erases the PE header, while MRCIF is characterized by the PE header, so the hidden injection page cannot be detected at all.

In terms of time consumption (as shown in Table 4), MRCIF consumes the most time in all the methods. This is due to the translation of all virtual addresses of all processes. Because the detection process is not real-time, the time consumption should be worthwhile. Moreover, the translation of all virtual addresses can be executed in parallel by the process to improve the efficiency of hardware usage, but we have not realized it at present, which is our future work.

5. Conclusions

This paper proposes MRCIF, a code injection covert memory page detection and forensic detection forensic algorithm based on memory structure reverse analysis. First, the physical memory pages containing DLL features in the memory image are located, and a sub-algorithm is designed for mapping physical memory space and virtual memory space, thus realizing reverse reconstruction of the physical page subset corresponding to the DLL code module. Then, in the virtual memory space, the LDR linked list structure of the process is reversely reconstructed, and a reverse reconstruction algorithm of the DLL virtual page subset is designed to reconstruct its virtual space. Finally, a DLL injection covert page detection sub-algorithm is developed based on the physical memory page subset and virtual space page subset. The method proposed in this paper does not use the VAD structure during detection and is therefore immune to VAD attribute tampering attacks. The experimental results indicate that MRCIF achieves an accuracy of 88.89%, which is much higher than that of the traditional DLL module injection detection method, and only MRCIF can accurately detect the VAD remapping attack. In practice, the proposed method is a preferred method for detecting hidden memory pages because of its higher accuracy, and it helps to quickly determine the direction of investigation for forensic analysis. Further research will be conducted on improving the efficiency of MRCIF and the characteristics of executable code in physical memory to deal with PE header erasure.

Author Contributions: Conceptualization, H.Z. and B.L.; methodology, H.Z. and B.L.; software and validation, W.L., L.Z. and S.Y.; formal analysis, H.Z., B.L. and C.C.; writing—original draft preparation, H.Z. and B.L.; writing—review and editing, B.L. and C.C.; funding acquisition, B.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China, grant number 60903220.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data are available from the corresponding author Binglong Li upon reasonable request.

Acknowledgments: The authors would like to thank all the editors and reviewers who participated in the review.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fewer, S. Reflective Dll Injection. 2008. Available online: https://dl.packetstormsecurity.net/papers/general/HS-P005_ReflectiveDllInjection.pdf (accessed on 15 November 2021).
2. Blaam, M. Process Hollowing. 2015. Available online: https://github.com/m0n0ph1/Process-Hollowing (accessed on 23 November 2021).
3. Palutke, R.; Block, F.; Reichenberger, P.; Stripeika, D. Hiding Process Memory Via Anti-Forensic Techniques. *Forensic Sci. Int. Digit. Investig.* **2020**, *33*, 301012. [CrossRef]
4. Galloro, N.; Polino, M.; Carminati, M.; Continella, A.; Zanero, S. A Systematical and Longitudinal Study of Evasive Behaviors in Windows Malware. *Comput. Secur.* **2021**, *113*, 102550. [CrossRef]
5. Srivastava, A.; Jones, J.H. Detecting code injection by cross-validating stack and VAD information in windows physical memory. In Proceedings of the 2017 IEEE Conference on Open Systems (ICOS), Miri, Malaysia, 13–14 November 2017; pp. 83–89. [CrossRef]
6. Block, F.; Dewald, A. Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries. *Digit. Investig.* **2019**, *29*, S3–S12. [CrossRef]
7. DFRWS. DFRWS 2005 Forensics Challenge. 2005. Available online: https://github.com/dfrws/dfrws2005-challenge (accessed on 23 November 2021).
8. Schuster, A. Searching for processes and threads in Microsoft Windows memory dumps. *Digit. Investig.* **2006**, *3*, 10–16. [CrossRef]
9. Dolan-Gavitt, B. The VAD tree: A process-eye view of physical memory. *Digit. Investig.* **2007**, *4*, 62–64. [CrossRef]
10. Kornblum, J.D. Using every part of the buffalo in Windows memory analysis. *Digit. Investig.* **2007**, *4*, 24–29. [CrossRef]
11. Guo, M.; Wang, L. Windows physical memory analysis method based on KPCR structure. *Comput. Eng. Appl.* **2009**, *45*, 74–77+143. [CrossRef]

12. Zhang, S.; Wang, L.; Zhang, R.; Guo, Q. Exploratory study on memory analysis of Windows 7 operating system. In Proceedings of the 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), Chengdu, China, 20–22 August 2010; pp. V6-373–V6-377. [CrossRef]
13. Cohen, M.I. Characterization of the windows kernel version variability for accurate memory analysis. *Digit. Investig.* **2015**, *12*, S38–S49. [CrossRef]
14. Google Rekall Forensics. Available online: http://www.rekall-forensic.com/ (accessed on 24 November 2021).
15. Cohen, M. Forensic analysis of windows user space applications through heap allocations. In Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC), Larnaca, Cyprus, 6–9 July 2015; pp. 237–244. [CrossRef]
16. Li, B.; Zhou, Z.; Zhang, Y.; Zhang, H.; Chang, C. Memory fragment file carving algorithm based on the reverse of the structure chain. *J. Commun.* **2021**, *42*, 117–127. [CrossRef]
17. Zhai, J.; Xu, X.; Chen, P.; Yang, H. Stack Forensics Based on Meta Data and Instruction Flow of 64-bit Windows. *J. Harbin Univ. Sci. Technol.* **2021**, *26*, 51–59. [CrossRef]
18. Pshoul, D. Malthfind Volatility Plugin. 2016. Available online: https://github.com/volatilityfoundation/community/blob/d9fc0727266ec552bb6412142f3f31440c601664/DimaPshoul/malthfind.py (accessed on 29 April 2022).
19. Cohen, M. Scanning memory with Yara. *Digit. Investig.* **2017**, *20*, 34–43. [CrossRef]
20. Russinovich, M.E.; Solomon, D.A.; Ionescu, A. *Windows Internals*, 6th ed.; Microsoft Press: Redmond, WA, USA, 2012.