REGULAR CONTRIBUTION

A new approach for detecting process injection attacks using memory analysis

Mohammed Nasereddin[1] · **Raad Al-Qassas[2]**

Abstract

This paper introduces a new approach for examining and analyzing fileless malware artifacts in computer memory. The proposed approach offers the distinct advantage of conducting a comprehensive live analysis of memory without the need for periodic memory dumping. Once a new process arrives, log files are collected by monitoring the Event Tracing for Windows facility as well as listing the executables of the active process for violation detection. The proposed approach significantly reduces detection time and minimizes resource consumption by adopting parallel computing (programming), where the main software (Master) divides the work, organizes the process of searching for artifacts, and distributes tasks to several agents. A dataset of 17411 malware samples is used in the assessment of the new approach. It provided satisfactory and reliable results in dealing with at least six different process injection techniques including classic DLL injection, reflective DLL injection, process hollowing, hook injection, registry modifications, and .NET DLL injection. The detection accuracy rate has reached 99.93% with a false-positive rate of 0.068%. Moreover, the accuracy was monitored in the case of launching several malwares using different process injection techniques simultaneously, and the detector was able to detect them efficiently. Also, it achieved a detection time with an average of 0.052 msec per detected malware.

Keywords Fileless malware · Intrusion detection · Malware analysis and detection · Memory forensics · Process injection attacks

1 Introduction

Indications of a malware infection can be deduced from the device's impaired performance and decreased efficiency in executing regular tasks, such as using applications or browsing the internet, as the malware excessively consumes system resources. The frequent appearance of intrusive advertisements also serves as an indicator of a malware infection. Moreover, there may be an unexplained disk space expansion, resulting in a significant loss of storage capacity [47]. Additionally, certain types of malware grant unauthorized access to the attacker, enabling them to download secondary infections onto the victim's computer, often accompanied by a noticeable surge in internet activity. A malware infection can be identified when the antivirus software suddenly ceases to function and becomes impossible to restart, indicating that the malware has directly targeted it. The inability to access files and applications on the computer is a common symptom of a ransomware infection.

Statistics [6, 7] indicate that the total number of malware of all kinds amounted to approximately 1018 million samples by the 1st quarter of 2023, which displays the colossal rise in the total number of malware infections in recent years. Moreover, statistics show that 77% of malware in 2017 were fileless, and their rates are expected to increase dramatically by the end of 2023 [23]. Cyberattacks caused massive losses to many sectors, including business, services [53], and healthcare [4] [5], as more than 90% of financial institutions reported being targeted by malware in 2020.

Traditional malware takes the form of deceptive (.exe) files stored on the hard disk, requiring execution to download payloads, and spread malicious code. Nowadays, attackers employ complicated disguises and obfuscation techniques

✉ Mohammed Nasereddin
mnasereddin@iitis.pl

Raad Al-Qassas
raad@psut.edu.jo

1 Institute of Theoretical and Applied Informatics, Polish Academy of Sciences (IITiS PAN), 44100 Gliwice, Poland

2 Department of Computer Science, Princess Sumaya University for Technology, Amman, Jordan

to confound forensic analysis, including process injection attacks. To this end, these attacks inject malicious code into legitimate processes like svchost .exe, compromising their functionality and coercing the system to execute attacker commands. Process injection techniques enable the creation of fileless malware, residing solely in RAM without leaving traces on the hard disk [1]. Scripts, Windows PowerShell, and remote attacks facilitate injecting code into legitimate processes, making such attacks resilient even if the original script is detected and removed [30]. On the other hand, when security engineers receive suspicious activity alerts, they hope for traditional file-based malware, allowing easier tracing and identification of attack origins. Analyzing malware code aids in indicating targeted data and affected regions. In process injection attacks, attackers deceive victims by masquerading targeted processes as legitimate system processes in the task manager [8]. While some antivirus programs detect malicious files, attackers persistently use evolving and dynamic methods to bypass protection protocols and firewalls [2].

Certainly, the main challenges in detecting fileless malware in real-time applications are particularly relevant due to the critical nature of these applications, as they require immediate and predictable responses to inputs or events [10, 62]. Whereas Signature-based detection approach is widely used by commercial antivirus programs and firewalls to detect known malware. However, it falls short in detecting unknown malware as it requires constant updates to the signatures database [58]. Meanwhile, researchers are developing heuristic-based approaches to analyze known and unknown malware behavior. These methods often suffer from high false-positive rates and are time-consuming. In addition, reducing extracted features or API calls will significantly affect detection success rates [19, 41]. The heuristic-based approach proves effective in detecting attacks that involve direct malware files dropping into the system. Extracting API calls for process injection attack detection generates large amounts of data, making it challenging to differentiate between benign and malicious behaviors, especially when attackers create malicious libraries similar to legitimate ones.

The memory-based analysis is the present and future trend in the detection and analysis of fileless malware due to the fact that memory is a rich source of valuable information [59], and as mentioned before the attack takes place entirely inside the memory as an active process without dropping the malware file on the hard drives [60]. According to Ponemon [26], process injection attacks increased by 47% in 2017, and successful attacks increased by 77%. As a result, it invokes to rely on memory analysis and underscore the need for sophisticated and adaptive security measures that can effectively identify and mitigate fileless malware threats. Developing solutions that strike a balance between robust security and minimal latency is crucial in safeguarding the integrity of

real-time applications against the evolving landscape of fileless malware [16].

The substantial contributions of this paper:

1. Analyzing the distinct characteristics of fileless malware and the associated challenges in its detection.
2. Assessing the efficacy of various techniques and tools in identifying fileless malware, while highlighting their respective advantages and limitations.
3. Introducing a novel and robust investigative approach that enhances the detection of process injection attacks by examining the memory artifacts of malware. This pioneering approach is based on a comprehensive analysis of the artifacts left behind, rather than solely focusing on the malware code.
4. Improving analysis and detection time by eliminating the need for memory dumps and instead utilizing live memory analysis and parallel computing.
5. Providing valuable insights into potential research directions for the future advancement of fileless malware detection.

The rest of the paper is structured as follows. Section 2 clarifies the characteristics of fileless malware, memory analysis, and prevalent process injection attacks while summarizing related work. Section 3 expounds on the methodology employed in the proposed approach. Section 4 discusses the outcomes resulting from the performance analysis of the proposed approach, presenting comparisons with other works. Finally, Sect. 5 concludes the work and provides future directions.

2 Background and related work

In modern operating systems, the CPU operates in two primary modes: user mode and kernel mode (also known as system mode). Kernel mode has unrestricted access to memory addresses and is commonly utilized for executing operating system functions. However, if a process crashes in this mode, it can lead to a system shutdown. On the other hand, in user mode, the CPU is unable to directly access hardware or memory. Instead, it requires special APIs to access these resources. On the other hand, user mode is where the majority of code execution takes place, and it offers recoverability and repairability in the event of interruptions, as it operates at the level of normal processes [66]. Figure 1(Top) illustrates the relationship between kernel mode and user mode, providing a visual representation of their distinct roles and privileges.

Each process is mapped into memory according to Fig. 1(Bottom), and the operating system reserves the kernel region at the highest memory address for devices, paged/non-
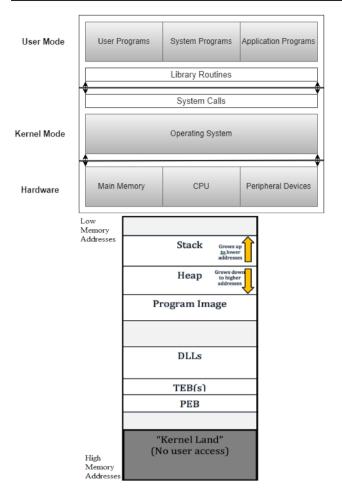
Fig. 1 (Top) Kernel mode and user mode, (Bottom) process memory layout

paged pool, system cache, etc., so that the user cannot access it. Any program stored in memory as a process is responsible for providing all the resources needed to run the program. Each process contains an executive process (_EPROCESS) that also resides in the kernel portion and contains process attributes and pointers to related data structures.

Moreover, each process in memory includes a Process Environment Block (PEB) that holds user-mode parameters specific to the active process, such as loaded modules (DLLs), the base address of the image (executable), the location of the heap, and environment variables. The PEB serves as one of the critical regions examined by the proposed approach to detect process injection attacks. Additionally, each process consists of one or more threads that share the virtual address space and system resources assigned to the parent process but differ in priorities, exception handlers, and local storage. Similar to the PEB, each thread has a Thread Environment Block (TEB) containing context information for various Windows DLLs and the location for the exception handler list and image loader.

Dynamic Link Library (DLL) libraries / modules, also known as executable modules, occupy memory space utilized by programs for efficient code reuse and memory allocation. The Program Image section contains the *.text* section containing executable code/CPU instructions, the *.rsrc* section storing non-executable resources like icons, images, and strings, and the *.data* section holding the program's global data. Furthermore, the process employs the heap as a shared pool to store global variables, which is dynamically allocated in memory using functions like *malloc()*. The program or process manages heap memory allocation, ensuring its persistence until program termination or release[1]

2.1 Fileless malware flow

The transfer methods of traditional malware and fileless malware to a user's computer are almost similar, but they diverge in terms of storage, execution, and system file access. While traditional malware typically follows a certain flow, fileless malware injects itself directly into a process, altering the execution path. Figure 2 demonstrates the disparity in the flow between traditional malware and fileless malware.

The primary distinction lies in the fact that traditional malware exists as an executable file (.exe) stored on the disk, which is activated only when the user initiates it, and hence executing its malicious actions. In contrast, process injection attacks or fileless malware exhibit the following key features:
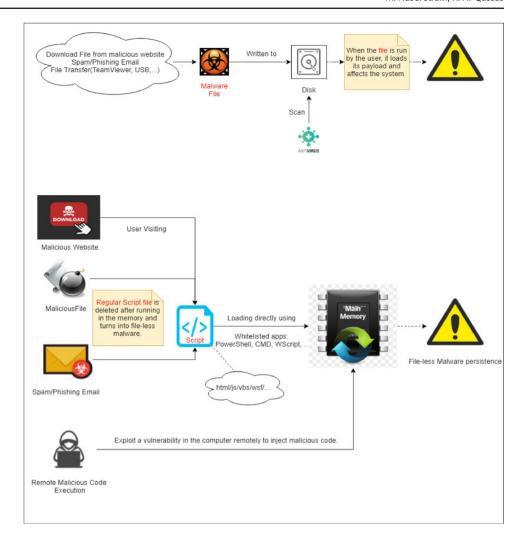
– It does not contain any file that is dropped into the hard disk like in other types, but the malicious code is integrated into one of the system processes to appear as a legitimate process.
– It may require particular files to run in the memory (e.g., scripts, registers, shellcodes, etc.), although there is no direct file. This type of file is attractive to attackers because it can be encrypted or obfuscated to appear as a legitimate file for antivirus programs. Malware exploits whitelisted built-in applications such as Windows Power-Shell, C-Script, and CMD to run these files, which contain commands to load malicious code directly into the memory.

Fileless malware is usually delivered by deceiving the victim using various social engineering methods, as shown in Fig. 2. It should be noted that the attacker can exploit a vulnerability in the victim's computer to gain access and inject malicious code directly and remotely into the memory without using any file. This differs from traditional malware in that the traditional type will not do any harm unless the file on

---

1 To go deeper and read more about the memory management process, Virtual Address Descriptors (VAD)s, Handles, and PE files format, you can refer to the following references: [11, 13, 40, 61, 70].

Fig. 2 Flow of (Top) traditional malware vs. (Bottom) fileless malware

the disk is run. The malicious code can still load its payload as long as it runs inside the memory unless the computer is rebooted. For this reason, some expert attackers use the popular registry files, scheduled tasks, or Windows Management Instrumentation (WMI) to ensure that files containing the malicious code remain even after shutting down the computer.

2.2 Memory analysis

Memory Analysis, a.k.a. memory forensics, is a crucial technique employed by forensic experts to investigate and identify malicious activities that may not be evident through hard drive analysis alone [3]. One of the main challenges in memory analysis is the volatile nature of memory, where data are stored temporarily and lost upon computer shutdown. To capture valuable information preceding an incident such as a system crash or security breach, investigators typically create a snapshot of the computer's memory data called a memory dump. This becomes particularly significant in cases involving fileless malware, where threat data resides solely in

the computer's memory, emphasizing the need for memory forensics to gain unique insights into the malware's behavior [60].

Live memory analysis involves examining the current state of memory and investigating active processes, user credentials, network information, executable files, etc. By intelligently utilizing live memory analysis to monitor suspicious activities, it becomes possible to eliminate fileless malware without the need for memory dumping, leading to improved analysis and detection time. The proposed approach primarily relies on two key factors: the arrival time of malware in memory and the specific memory regions that warrant investigation for identifying malware artifacts.

When investigating memory artifacts, one of the important things that should be considered is the memory-protection values assigned to the process executables and which give indications of suspected malicious code injection in the process. Table 1 explains the values we need to know in the investigation process.

As noticed from Table 1, every page created is assigned a memory protection value. The presence of some mod-

Table 1 Memory protection constants [39]

Constant/Value
PAGE_EXECUTE '0x10'
Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_READWRITE '0x04'
Enables read-only or read/write access to the committed region of pages. If *Da Execution Prevention* is enabled, attempting to execute code in the committed region results in an access violation.
PAGE_EXECUTE_READWRITE '0x40'
Enables execute, read-only, or read/write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY '0x80'
Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as PAGE_EXECUTE_READWRITE, and the change is written to the new page.

ules in the memory carrying the value *PAGE_EXECUTE_READWRITE* along with other pieces of evidence will help detect injection processes. This will be explained in detail in the Methodology section.

2.3 Related work

Process injection attacks involve injecting malicious code into active processes or creating a suspended process and injecting code into it [35]. Each process possesses its own private virtual memory space, consisting of kernel mode and user mode sections. Modifying the kernel, which impacts all system processes, is challenging and risky, as a single error can lead to a system-wide crash. Moreover, the kernel is protected and not easily modifiable. Consequently, attackers typically opt to inject malicious code into user mode, focusing on affecting the specifically targeted process. Figure 3 illustrates the prevalent techniques employed for injecting malicious code into processes.

Analyzing and detecting malware is a highly complicated and delicate process. The interest in this field is continuously growing in both academia and industry. The approaches followed in detecting malware, particularly fileless malware, were divided into three main approaches: The Signature-Based approach, the Heuristic-Based approach, and the Memory Analysis approach. After reviewing the literature and extracting opinions and conclusions in detecting process injection attacks, the following points were summarized:

- The Signature-Based detection approach is the worst. Although highly accurate, it can be easily bypassed with this advanced type of attack because it relies only on signatures [41].
- The Heuristic-Based detection approach is better than its predecessor in dealing with known and unknown malware. However, it still suffers from a high rate of

	Shellcode Injection	Forcing A DLL To Be Loaded	Sha256
1. DLL Injection		X	07b8f25e7b536f5b6f686c12d04edc37e11347c8acd5c53f98a174723078c365
2. PE Injection	X		ce8d7590182db2e51372a4a04d6a0927a65b2640739f9ec01cfd6c143b1110da
3. Process Hollowing	X		eae72d803bf67df22526f50fc7ab84d838efb2865c27aef1a61592b1c520d144
4. Thread Execution Hijacking	X		787cbc8a6d1bc58ea169e51e1ad029a637f22560660cc129ab8a099a745bd50e
5. Hook Injection		X	5d6ddb8458ee5ab99f3e7d9a21490ff4e5bc9808e18b9e20b6dc2c5b27927ba1
6. Registry Modification		X	9f10ec2786a10971eddc919a5e87a927c652e1655ddbbae72d376856d30fa27c
7. APC Injection		X	f74399cc0be275376dad23151e3d0c2e2a1c966e6db6a695a05ec1a30551c0ad
8. Shell Tray Window Injection	X		5e56a3c4d4c304ee6278df0b32afb62bd0dd01e2a9894ad007f4cc5f873ab5cf
9. Shim Injection		X	6d5048baf2c3bba85adc9ac5ffd96b21c9a27d76003c4aa657157978d7437a20
10. IAT and Inline Hooking	X	X	f827c92fbe832db3f09f47fe0dcaafd89b40c7064ab90833a1f418f2d1e75e8e

Fig. 3 Process injection techniques [29]

false-positive detections since it is not easy to distinguish between harmful and benign behaviors [14, 15, 63].
- Machine learning algorithms can be fooled by creating an amount of noise [27].
- Machine learning algorithms are only influential in detecting DLL injections and registry modifications [18, 20, 30, 45, 46].
- A Hybrid approach provides better results, but it is time-consuming as well as very complicated. Moreover, the false-positive detection rate was not improved sufficiently. It is a convenient approach to dealing with file-based malware [59].
- The Memory-Analysis approach is the optimal solution for investigating this type of threats because the memory is the container of all processes. Therefore, analyzing it greatly helps detect any malicious behavior [58].

– Meanwhile, the large gap between the development and practical implementation of reliability prediction models in software must be addressed, emphasizing the need for robust models and techniques to enhance usability and reduce time and cost investments to obtain high-quality software [51, 52]. some examples are described in [54–56].

In Table 2, the literature is summarized in terms of the approach used, and the attacks dealt with, the extracted results, and some remarks.

Works that followed the Memory-Analysis approach were at the forefront in terms of detection accuracy. Table 3 highlights some related works whose results were discussed and presented transparently and clearly according to the approach, accuracy, error rate, and false-positive rate.

3 Methodology

Generally, in the process injection, the legitimate process is compromised using one of the code injection techniques, through memory allocation, memory writing, forcing the process to execute something, or through a mechanism that forces the DLL to be loaded while the process starts. Using a memory forensic framework such as Volatility [65] gives valuable results in detecting injection techniques, but it will consume computer resources because it always needs to dump the memory, and besides, it may face a time hurdle, as mentioned in [62]. The new approach aimed to work on three main points: improving detection time, reducing resource consumption, and not neglecting high accuracy.

The new approach enhanced the detection of process injection attacks and reported the suspicious anomalies and heuristics by tracing the Windows events and relying on live memory analysis in a systematic way to extract and analyze artifacts that prove a violation occurred. The concept of parallel computing is used at the software level, where the main software (Master) plays the role of the organizer, breaking the large task into smaller tasks and distributing them to the other programs (Agents).

Data capturing is a critical investigation step, and any mistake or delay in the capture process may cause evidence to be lost. Considering that memory is one of the volatile components, we preferred to use the live memory analysis approach and combine it with the idea of using the trigger, which will be detailed later. As a result, the memory is examined as soon as the signal arrived. At the same time, there is no need to dump the memory in each analysis process, which improves the accuracy and time of detection jointly.

In addition to what is mentioned in the previous sections, most related research concentrated on analyzing the malware itself. Our new approach focused on investigating the artifacts of malicious behavior. Figure 4 is a structural diagram that lays out the main parts of the following methodology.

The dataset was extracted from several sources (explained in Sect. 4), reused, and compiled into a single project called Injection Tool. Furthermore, the entire project was tested before entering the memory analysis phase, as shown in Fig. 4(Top). Figure 4(Bottom) shows the general idea of the proposed detection approach. In the beginning, all memory processes will be enumerated to retrieve the process ID for all processes and arranged to deal with each one separately. The importance of this critical step lies in systematically collecting details to be examined thoroughly, determining the corresponding parent process for each process, and identifying the suspicious one. The presence of a process that did not start from the parent process gives a sturdy indication of a violation.

Secondly, when the system gives a trigger to the master detector that there is a new event, including (creating process code, creating a thread, creating a file, image loading, registry operations, etc.) in the memory, the process enters the detector, and the log files are collected through monitoring Event Tracing for Windows (ETW) facility which runs at the kernel level that allows the user to trace the events of each process defined in the memory [12]. Moreover, the master detector lists all executable files for the process and operates the secondary detectors (Agents). The secondary detectors start scanning the memory for artifacts related to suspicious activities that indicate malicious code has been injected into the process. The idea behind using parallel computing is to distribute the load and reduce the consumption of computer resources since only the agent responsible for detecting the potential attack is running. Figure 5 shows the communication operation between the master detector and the agents.

The master detector sorts the processes in the memory and lists all executable files for each process. Meanwhile, it also collects log files from tracking Windows events. Agents start working sequentially based on a signal sent from the master detector when there is suspicious activity happens. Each one has the task of searching for artifacts related to the presence of suspicious anomalies in the place allocated for the memory of the process that automatically holds an ID number given to it by the system, such as (Process Identifier (PID):10385). The master detector is also responsible for stopping the damage, organizing the results, and presenting the final report after receiving the response from the agents.
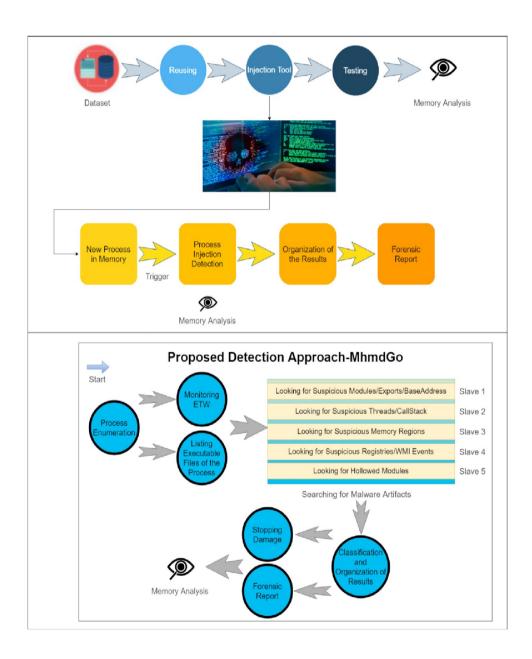
Table 2 Literature summary

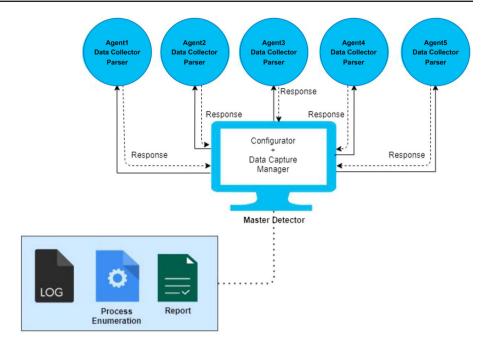| References	Detection approach	Attacks/Techniques	General remarks
N/A	Signature-Based	N/A	It was not used to detect process injection attacks (fileless malware)
[14, 15]	Heuristic-Based	DLL Injection	• Extract sensitive API calls. • Convenience in detecting file-based malware
[18, 20, 30, 45, 46]	Machine-Learning	DLL Injection, Registry Modifications	Relying on examining features (e.g., API function calls, imported DLLs, and registry activity)
[10]	Memory-Analysis	Reflective DLL Injection, Process Hollowing	• Examine PTE for executable pages. • Use a forensic framework (Rekall) to analyze the extracted data
[67]	Memory-Analysis	DLL Injection	Comparing Executables and DLLs between memory and disk
[8]	Memory-Analysis	DLL Injection, Process Hollowing	• Identify anomalies. • Event timeline monitoring when using the CreateRemoteThread and LoadLibrary functions. • Need a forensic framework to analyze the extracted data
[17]	Memory-Analysis	DLL Injection	Bind each process to all DLLs it needs by examining IDT and EDT
[62]	Memory-Analysis	DLL Injection, Hook Injection	A new technique for solving the timing problem based on monitoring suspicious API calls. (Trigger-Based)
[49]	Memory-Analysis	DLL Injection, Hook Injection	Incorporating static analysis and memory forensic analysis to reduce encryption and obfuscation
[47]	Memory-Analysis	DLL Injection	Detect distributed code injection using the stack-tracking-based technique combined with a technique for finding threads that are waiting for synchronization objects
[31]	Memory-Analysis	DLL Injection	• Sending alerts about the presence of malware by monitoring and analyzing endpoints. and, • Involve the human interaction in the memory analysis to identify abnormal behaviors
[42–44]	Memory-Analysis	Process Hollowing	• Parent–Child relationship. • Comparing PEB and VAD
[36]	Memory-Analysis	Process Hollowing	Use the fuzzy hash to calculate the process similarities in virtual memory
[60]	Memory-Analysis	DLL Injection	Applied two use cases to prove changes in VAD and memory protections
Proposed approach	Memory-Analysis	DLL Injection, Reflective DLL Injection, Process Hollowing, Hook Injection, Registry Modifications (AppINIT_DLLs),.NET DLL Injection	• No memory dumps required. Waiting for a trigger (notification). • Need not a forensic framework. • Comprehensive analysis of memory artifacts (Suspicious Modules, Exports, BaseAdress, Threads, CallStack, Memory Regions, Registries, Hollowed Modules).

Table 3 Related works results

References	Detection approach	Accuracy (%)	Error rate (%)	FP rate (%)
[14]	Heuristic	87.70	12.30	N/A
[46]	ML	96.00	4.00	0.922
[30]	ML	93.85	6.15	1.320
[49]	Memory	90.00	10.00	N/A
[8]	Memory	99.84	0.16	0.151
[31]	Memory	96.30	3.70	1.045

Fig. 4 (Top) Top view diagram, (Bottom) workflow of the proposed approach
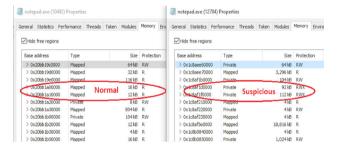
Fig. 5 The relationship between master detector and agents

Fig. 6 Mapped vs. unmapped file

procedure DETECTION

　　Scan the memory (live)

　　Enumerate all active processes

　　while New event in the memory **do**

　　　　Monitor the Event Tracing for Windows (ETW) facility

　　　　List all executable files for the process and collect log files

　　　　Run proper agent // The working principle of each agent is detailed in the methodology section

　　end while

　　Classify and organize the results

　　Produce the forensic report

end procedure

3.1 Workflow of the agents

The five agents work as follows:

Fig. 7 Investigating suspicious exports/imports/modules

1. Agent 1: it is responsible for searching for suspicious modules in memory regions with the *PAGE_EXECUTE_READWRITE* memory protection flags, Read, Write, and Execute (RWX). The main module (.exe) cannot be specified with types private, not shared, or cannot be inherited by other processes, and memory protection flags, but it must always be memory-mapped, as shown in Fig. 6. In other words, a memory-mapped file is a file that is physically located on the disk and has been Bit-by-Bit copied to memory at the specified location, so it is also important to monitor the base address and the full path of the module if it is from an unexpected location.

   This agent usually helps detect DLL injections but may identify as a good indicator that the process was hollowed. Furthermore, it looks for known bad suspicious exports/imports, such as *ReflectiveLoader()*. See Fig. 7.

Fig. 8 (Top) Investigating suspicious threads, (Bottom) Call stack of a suspicious thread
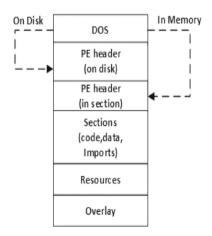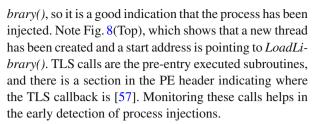

Fig. 9 Portable executable (PE) file format [61]

A deeper look is taken at modules/exports, as shown in Fig. 7. Unlike the remaining three, as shown in the upper right side of the figure, it is an unverified file from Windows, and the timestamp is different from the range of the rest and seems to have been created soon, as well as the difference in the form of the checksum.

2. Agent 2: examines memory protections for threads with RWX flags starting at the pointer of the thread base address. The process hosts a TLS callback (thread-local storage). When a new thread is created, it receives a notification like *CreateRemoteThread( )* or *UserThreadStart( )*. Here, the start address of the thread is checked if *LoadLi-*

brary( ), so it is a good indication that the process has been injected. Note Fig. 8(Top), which shows that a new thread has been created and a start address is pointing to *LoadLibrary( )*. TLS calls are the pre-entry executed subroutines, and there is a section in the PE header indicating where the TLS callback is [57]. Monitoring these calls helps in the early detection of process injections.

When a new thread is created, and it is not mapped to a location in the disk, but from type: private, this file is suspicious because it is an executable file with RWX privileges and is referred to as an unbacked executable file or floating code, so in the proposed approach we are also searching for unbacked symbols that indicate a suspicious thread has been found due to presence of malicious behavior in the process. The call stack of the suspicious thread is always unmapped, as shown in Fig. 8 (Bottom). Agents 1 and 2 play the leading role in detecting most DLL injections and hook installation attacks.

3. Agent 3: this agent looks for any suspicious regions in memory that contain the *PAGE_EXECUTE_READWRITE* memory protection RWX flags. Besides, it checks the PE files of these regions using Fuzzy PE matching, which is based on the principle of contrast with Boolean logic, which holds that the correct value of the variables is either 0 or 1, but what if the value is at the middle or near it? [71]. For example, the temperature can range between *extremely hot* with a value of 1 and *extremely cold* with a value of 0, where it can be called *warm*. This is called a decision based on partial truth. Figure 9 shows the structure of the PE File.

As shown in Fig. 9, a copy of the PE header is stored in a section in the memory region. In the proposed approach, agent 3 compares PE headers for suspicious memory regions that contain RWX flags with those stored on the disk. The comparison is based on the Levenshtein distance algorithm to calculate the difference between two sequences, where the distance is calculated by modifying at least a single character in terms of insertions, deletions, or substitutions that are capable of changing one word to another [72]. The agent identifies suspicious regions based on the differences between the two copies.

The presence of wiped headers with previous pieces of evidence dramatically increases the likelihood of an injection in the process, as shown in Fig. 10. Mostly the unbacked PE memory sections have the full MZ headers, but the attacker might simply wipe these headers. What distinguishes the proposed approach is that it looks for suspicious regions, examines the wiped or unwiped headers, and adds them to the findings.

This agent helps powerfully detect stealth attacks, such as Reflective DLL injection and Classic DLL injection (Stealth). It is good to check out more information about the headers of the files and how they help tremendously in
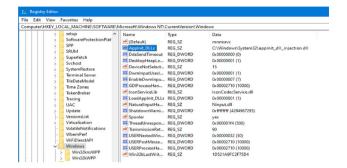
Fig. 10 Wiped PE headers

Fig. 11 Modified Appinit_DLL registry key

detecting file-based malware [21]. We will not dive further because it departs a little from the scope of this work, as it is only helpful for us to consider whether the PE headers have been wiped for suspicious memory regions.

It can also detect suspicious assemblies that are used for DLL injections by matching function prefaces, as it searches for any CLR module loaded without file backing in the associated area of memory.

4. Agent 4: as mentioned earlier, the attacker may use system registries or WMI to save configurations for relaunching the malicious code in case the computer is rebooted, or a specific file is opened. This agent investigates the most common registry persistence keys in "HKEY_LOCAL_MACHINESOFTWAREMicrosoft WindowsCurrentVersionRun" used to obtain the persistence of the injected code (e.g., AppCertDLLs, Appinit _DLL, IFEO, Etc.). Here is an explanation of some of them:

- AppCertDLLs: Each process that uses certain functions of the *CreateProcess()* API has its DLLs value specified by the AppCertDLLs registry key " HKEY_LOCAL_MACHINESystemCurrentControl SetControlSessionManager". This key can be abused as an attacker modifies it to ensure persistence and escalates privileges by helping load malicious DLLs and run in the context of any detached process in the memory [66, 68].
- Appinit_DLL: *user32.dll* is a frequently used library loaded in almost every process. DLLs for this library are specified using the Appinit_DLL registry key " HKEY_LOCAL_MACHINESOFTWAREMicrosoft

WindowsNTCurrentVersionWindows". This library can also be modified to load malicious DLLs and run them in the context of any process that uses this library. Figure 11 shows the Appinit_DLL registry key after it has been modified to contain the path of the malicious DLL file "appinit_dll_injection.dll."

It should be noted that Microsoft defaults for *LoadAppinit_DLLs* is set to 0 to disable the loading of DLLs via AppInit to protect users from malicious files, so the attacker needs to set it to "1" to activate the service. After setting the key value and specifying the path of the malicious DLL file, it is randomly loaded into the "Program Files" directory in most Windows applications by the *user32.dll* library.

- IFEO: This value is specified by the registry key " HKEY_LOCAL_MACHINESOFTWAREMicrosoft WindowsNTCurrentVersionImageFileExecutionOp tions/<program>" to allow developers to attach software debuggers to programs. When this value is set to a program, the specified executable in memory will be run instead of the program as a debugger. Some attackers exploit this feature to force the system to run its malicious code every time this process is run. Agent 4 searches for suspicious phrases in the registries like *user32.dll* so that if it finds this library has been modified, it will add it to the list of evidence that will be shown in the report about the presence of injections in the process. Furthermore, it looks for WMI malicious entries that are also used for malware persistence, privilege escalation, and reconnaissance. where it monitors (__EventFilter, __Event-Consumer, __FilterToConsumerBinding). *__Event-Consumer* indicates persistence payload as well as commands to carry out the malicious code contained in the *ActiveScriptEventConsumer* and *Command-LineEventConsumer* classes. For *__EventFilter*, it indicates the condition required to trigger the event, and *__FilterToConsumerBinding* is responsible for binding the classes.

5. Agent 5: several found artifacts can be inferred that a process hollowing technique was used. This agent is doing the following:

- The legitimate process has a real system parent process, but the parent of the suspicious process is compromised. Figure 12 shows the difference between the two, as it appears in the *Parent* field for the suspicious process *non-existent process*. This agent searches for processes that do not have a parent process, more precisely, processes that do not start from the corresponding parent process that is terminated.
- Compares the differences between PEB (which is located in a process memory and contains the base
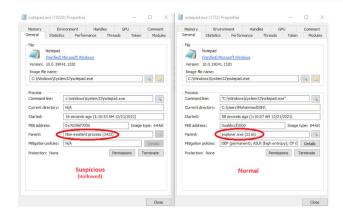
Fig. 12 Normal vs. Hollowed process

address "Entry point" and full path of the executable file) and VAD (which is a data structure located in the kernel and holds information about allocating virtual address space for the process as well as the start & end address, initial protections, and the executable file path if was loaded). The difference in their values confirms that the process has been hollowed. Also, it compares code size and linker version.

– Checks the CREATE_SUSPENDED flag since the process hollowing technique relies on creating a new process in a suspended mode. It acts as a container for the malicious code and is then stored in the designated memory location after unmapped the legitimate process. If it is found that the *CreationProcess* Flag holds the value ($0x00000004$), then this is an indication of the intent to use the process hollowing technique [28]. This indicator is not strong enough because the suspended mode after hollowing the process is stopped by calling *ResumeThread( )*, which removes the flag's value. However, it is not bad to monitor it constantly. Note that Agent 1 can help detect process hollowing by searching suspicious memory protections. More clearly, if the process's VAD has memory protection *PAGE_EXECUTE_READWRITE* with RWX flags, this is an indication that the process has been injected because it is expected that the loaded executable has memory protection *PAGE_EXECUTE_WRITECOPY*.

3.2 Classification results

Each agent responds to the master detector after they have finished examining the artifacts of the suspicious activities that they are looking for with one of the two following responses:

– *First case, not suspicious:* Nothing was found. For example, suppose the process has not been injected with any malicious code, has not been hollowed out, or has already

been injected, but the agent responsible for searching in the specified region could not detect the suspicious behavior. At the same time, the presence of suspicious threads or modules is being investigated; indeed, the responsible agent will reply that it did not find anything.

– *Second case, suspicious:* Assuming that the process was injected using create remote thread technique. Agent 2, which is responsible for searching for suspicious threads, will return the result to the master detector with investigation data proving it. Note that more than one agent can work in parallel to detect more than one type of injection. Finally, the master detector collects and organizes the data and displays the final forensic report containing all the findings. The retrieved information includes suspicious process identifier PID, full path, number of threads, suspicious thread identifier (TID), discovered modules, base address, abnormal user permissions, integrity levels, Thread base priority, code size, unique thread token, Privileges, Logon Session/Type, and other investigation information related to the detected activity.

4 Results and evaluation

This section discusses the results of experiments to evaluate the performance of the proposed approach in detecting process injection attacks. For this purpose, it provides an overview of the testing platform, dataset sources, and criteria used in the evaluation process.

4.1 Testing environment set-up

The virtualized environment provides various options for the computer user. It helps information systems administrators install updates and upgrades in a specific computer region without interfering with working on other programs that may be on a different operating system. It is also essential for software developers to easily write their programs and navigate between the environments [34]. In the event of a failure, the operating system on which it is installed is treated as a program that can be repaired or reloaded by another alternative system quickly without the need to create an image of the hard disk [69]. This technology assisted in accomplishing many things efficiently and smoothly.

A safe and isolated virtual environment was prepared using VMware Workstation 15.5.1 Pro [64] to perform real-world process injection experiments, execute the proposed detection approach, and evaluate the findings. Table 4 presents the prepared virtual machine configurations.

The reason behind testing Windows 10 was that it is the most used among computer users. Moreover, several types of process injection attacks only target Windows operating systems [8]. Two separate projects were created using Microsoft

Table 4 Virtual environment configurations

Operating System	Windows 10 Enterprise, version 20H2(10.0.19042.0) [38]
OS Type	64-bit Operating System
OS Architecture	x64-based processor
Processor	Intel(R) Core (TM) $i5 - 3230M$ CPU @2.60GHz
Disk Space	40.00GB
Physical Memory (RAM)	4.00GB
Number of Cores	4
Number of Threads	8
Tools	Visual Studio 2019, ProcessHacker 2 [37], GitHub Copilot [24]

Visual Studio 2019 to perform the injections and simulate the proposed approach. Furthermore, the dataset of the process injection attacks was extracted from several sources, reused, and organized under the "Injection Tool" project. Besides, the proposed approach was implemented in the "Detection Tool" project. For this purpose, the C/C++ was mainly used, and the C# in.NET DLL injection. Furthermore, the ProcessHacker 2 software, which helps capture live images of the memory, was used to monitor the memory, list the running processes, compare the status of the legitimate and injected process executables (before, during, and after injection) with the results of the proposed approach, and record all required information. More details are in the appendices.

The fantastic tool GitHub Copilot was used to build some code blocks to implement the proposed detection approach. This tool has been trained using artificial intelligence to help developers write their code quickly. It offers a wealth of examples, solutions, and suggestions in the form of individual lines and whole functions. It also converts comments into code. Besides, it does Autofill for repetitive code, and it helps a lot in testing the code. The tool was used as a good helper in constructing and implementing the proposed approach.

4.1.1 Dataset sources

In order to evaluate the proposed approach performance, the dataset was carefully selected from several trustworthy sources due to the difficulty of extracting them from a single source, given that each injection tactic is a separate complete project.

- For Classic DLL Injection, Reflective DLL Injection, Hook Injection, and Process Hollowing techniques. The experiments were conducted on the datasets that were extracted respectively from projects [9, 25, 32].

Table 5 Evaluation Metrics

Metrics	Description
Number of handled techniques	Denotes the number of techniques that can be identified in their early stages
Accuracy	It is calculated using equations (1), (2), and (3)
Time	Indicates the actual time spent to detect the injection in the process from notifying the trigger until printing the results
Resources consumption	Denotes the amount of computer resource consumption during the analysis process
Automated extracted records(artifacts)	The obtained forensic information as a result of the memory analysis and its importance in detecting the attack

- Also, for Reflective DLL Injection, was extracted from the source [22], as he is the father and first developer of this technique.
- For attacks using (AppINIT_DLLs, and.NET DLL Injection) injection tech-niques. The source was the Specialized Group (Red Teaming [50]) & [48].

4.2 Experiments and discussion

This section exhibits the results of experiments conducted on the proposed approach compared to related works.

4.2.1 Evaluation metrics

To assess the efficiency of the proposed approach in comparison with related work, the set of evaluation metrics listed in Table 5 was used:

$$Accuracy = \frac{Number\ of\ detected\ attacks}{Total\ number\ of\ attacks} * 100\% \quad (1)$$

$$Error = 1 - Accuracy \quad (2)$$

$$FPR = \frac{Number\ of\ false\ detected\ attacks}{Number\ of\ detected\ attacks} * 100\% \quad (3)$$

4.2.2 Scenarios and assumptions

For the purposes of conducting injection experiments, the Notepad process located in the path C:\Windows\notepad.exe was used. It was assumed that the attacker already had a *meterpreter shell* on the victim's system and tried injecting the notepad system process using different injection tech-

niques. The number of attacks was gradually increased to measure the performance of the proposed approach. Also, the virtual environment used was safe, well-checked, and not targeted by any attacks, and the VAD region was entirely emptied, which reduced the percentage of false positives.

It should be noted that the attacker's primary goal in using process injection techniques is to facilitate the launch of the malware payload in order to escalate the privileges and perform malicious activities such as evasion and data theft through one of the system processes. Processes other than Notepad can be injected (e.g., lsass.exe, calc.exe, werfault.exe, svchost.exe, regsvr32.exe, etc.).

For each type of attack, the memory was analyzed through several scenarios before injection, during DLL execution, after DLL termination, and after the injection process was terminated as follows:

– No injections.
– Injection on a single process using one technique.
– Injection on a single process using more than one technique simultaneously.
– Injection on multiple processes using one technique.
– Injection on multiple processes using more than one technique.

4.2.3 Results and discussion

1. Number of Handled Techniques
   The proposed approach achieved satisfactory results in detecting process injection attacks compared to other works. Regarding the number of techniques that can be detected in their early stages, the approach has successfully dealt with six different techniques: Classic DLL Injection, Reflective DLL Injection, Process Hollowing, Hook Injection, Registry Modification (AppINIT_DLLs), and .NET DLL Injection. It is expected to help detect more techniques with more work.
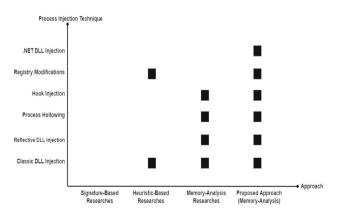

Fig. 13 Comparison of the number of detected techniques

On the other hand, the Signature-Based approach did not report any research working on detecting this type of attack but rather file-based malware. Research results using the Heuristic-Based approach indicated its ability to detect some types of DLL injection attacks by extracting sensitive API calls. Moreover, machine learning research helped detect registry modifications.

Works that relied on memory analysis took turns detecting a higher number of process injection techniques. For example, [8, 10] focused on DLL injection and process hollowing, while [49] on DLL Injection and Hook Installation, and others focused on one type, such as [31] on DLL Injection and [36] on process hollowing. What distinguished the proposed approach was that it focused on a greater number of techniques. Figure 13 illustrates the comparison.

2. Accuracy and Time
   The detection accuracy of the proposed approach was fixed at 99.93% after testing 17411 malware samples using the different techniques discussed, and the average detection time was approximately 0.052 msec/attack. The experiment was repeated 30 times in order to verify the results, and the average was calculated. Table 6 displays the final results of the experiments.
   The plugins that were developed and tested in many related works using the popular open-source memory forensics framework *Volatility* were also tested on the same dataset, the number of attacks, scenarios, and conditions (FindDLLInj [8], Malfind [65] [60], Threadmap [33], and Hollowfind [43]). The proposed approach achieved the highest detection accuracy rate among them, in addition to the process injection techniques it can deal with. Tables 7 and 8 and Fig. 14 shows the results of the comparison.
   The experiment was also performed 30 times, and the average was calculated in order to have a fair comparison with the mentioned plugins in terms of accuracy and detection time.
   As mentioned earlier, the works that followed the memory analysis approach fulfilled a high detection accuracy in general compared to their counterparts. The proposed approach greatly reduced the false-positive detection rate, as shown in Table 8.
   Fortunately, the comprehensive proposed approach endured a modest false-positive rate of 0.068%. Sometimes it identified the Injection Tool used in the experiments as a threat, and this can be considered a reasonable thing because the project contains instructions to build and compile the malicious code. False-positive results may be encountered due to protected applications or some behaviorally erratic security programs that behave similarly to attacks to improve their detection, but this may make the detection of real threats more difficult.

Table 6 Results of experiments

Number of malware samples	Number of detected malware	Accuracy (%)	Average detection time (msec)	Average detection time/malware (msec)
1	1	100.0	0.73	0.730
10	10	100.0	1.14	0.114
50	50	100.0	2.91	0.058
100	100	100.0	5.30	0.053
500	500	100.0	26.18	0.052
750	750	100.0	38.49	0.051
1000	999	99.90	52.10	0.052
5000	4996	99.92	260.27	0.052
10000	9993	99.93	520.09	0.052
$\sum$ 17411	17399	99.93	910.74	0.052

Table 7 Process injection techniques that can be handled compared with Volatility Plugins

References	Process injection techniques
FindDLLInj	Classic DLL Injection
Malfind	Classic DLL Injection, Reflective DLL Injection, Process Hollowing, Hook Injection
Threadmap	Process Hollowing
Hollowfind	Process Hollowing
Proposed Approach	Classic DLL Injection, Reflective DLL Injection, Process Hollowing, Hook Injection, Registry Modifications,.NET DLL Injection

We will endeavor to reduce it to as close to zero as possible in the future. Besides, the performance was monitored when attacks were executed using more than one technique simultaneously, and a successful detection rate of 99.91% was achieved.

The approach enhanced the detection and reduced the remediation time because it did not require a memory dump while not affecting the accuracy. Meanwhile, it was easy to implement and did not need a complex infrastructure. Table 9 and Fig. 15 show the relationship between the injection time and detection time under the optimal conditions for the number of executed attacks.

It can be noticed that the injection timeline had a gap in the form of an increase after the number of attacks exceeded 1000, but the detection line was not affected, as it kept increasing almost steadily. This means that the approach's performance was not affected by the increased number of attacks.

By monitoring the detection time on the same dataset used to test the performance of the proposed approach compared to Volatility plugins, it achieved a very short detection time compared to its counterparts. The difference can be seen in Table 10 and Fig. 16.

Figure 16 shows a clear superiority of the proposed approach in terms of detection time. It fulfilled the best average detection time when the number of carried out attacks passed 10, 000 with a stable increase in proportion to the increase in the number of attacks. Also, a sudden and unexplained increase in detection time for the Malfind plugin can be seen after increasing the number of attacks to more than 2000.

In the memory analysis, dumping memory is cumbersome and requires a lot of time, but many researchers such as [47, 62] developed solutions that were slightly similar to the proposed approach, where they monitored some activities in the system. When suspicious behavior was noticed, the memory was dumped, and the resulting image was analyzed. Also, one of the suggested solutions is to dump

Table 8 Comparison of detection accuracy with Volatility Plugins

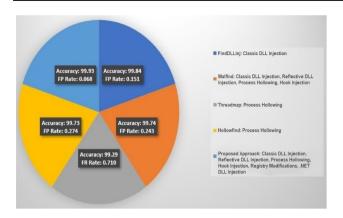| Reference	Accuracy (%)	Error Rate (%)	FP Rate (%)
FindDLLInj	99.84	0.16	0.151
Malfind	99.74	0.26	0.243
Threadmap	99.29	0.71	0.710
Hollowfind	99.73	0.27	0.274
Proposed approach	99.93	0.07	0.068

Fig. 14 A Comparison chart of detection accuracy with Volatility Plugins

Table 9 Relation between injection time and detection time (Optimal conditions)

Number of malware samples	Injection time (msec)	Detection time (msec)
1	1.01	0.68
10	2.94	1.13
50	11.07	2.82
100	20.91	5.18
500	157.14	25.98
750	244.62	38.09
1000	312.05	51.44
5000	1711.42	255.02
10000	3522.23	509.37

Fig. 15 Comparison between detection time and injection time

resulting from the investigation process in addition to real experiments results are shown in the appendices.

As shown in Tables 12 and 13. The proposed approach sought to produce a forensic report containing detailed information about the location of the threat in the memory, its effect, its type, and the escalated privileges. Figures 17, 18, 19, and 20 are examples of the forensic reports generated by (FindDLLInj, Malfind, Threadmap, Hollowfind) Volatility tool plugins.

5 Conclusion and future work

This work covered process injection attacks based on fileless malware. The differences between them and traditional malware are flow methods and consequences. By reviewing the literature, we concluded that memory analysis is the optimal approach for analyzing and detecting these types of attacks considering that the malicious code is executed entirely through an active process in the memory. Works that adopted the memory analysis approach fulfilled better results in terms of detection accuracy and false-positive detection rate compared to those that depended on the Signature-Based approach or Heuristic-Based approach.

The proposed approach introduced new method to perform a thorough live analysis of memory artifacts and generate evidence of the presence of malicious code without dumping the memory. Moreover, the new approach helped reduce resource consumption and significantly decreased detection and remediation time by adopting the parallel computing principle. The performance of the proposed approach was tested and assessed on a reliable dataset, and the results were compared with related works according to the appropriate evaluation metrics. The new approach achieved the

the memory every 30 seconds. As mentioned earlier, malware may execute and be cleaned up in a very short time. The proposed approach addressed this dilemma in two ways. First, by monitoring the ETW to identify any suspicious behavior and adopting the parallel computing principle, which eliminates the need to dump the memory every period, thus, helping to enhance detection time and reduce resource consumption. As mentioned in Sect. 3, each agent is responsible for searching for specific artifacts inside the memory. Table 11 binds each agent and the attacks it helped detect.

As seen from Table 11, most types of attacks were detected by more than one agent, and each agent contributed to the detection of more than one attack, which helped improve the accuracy and time of detection significantly.

3. The Forensic Report

The produced forensic report contains valuable detailed forensic information about suspicious behavior detected inside the memory to locate the threat and prevent its impact. The report includes the suspicious process name, PID, process path, suspicious module, thread, etc. Tables 12 and 13 present an example of two forensic reports
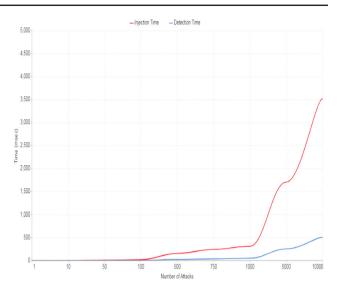
Table 10 Comparison of detection time (msec) with Volatility Plugins

Number of malware samples	FindDLLInj	Malfind	Threadmap	Hollowfind	Proposed approach
1	2.12	4.28	3.21	4.76	0.73
10	4.01	8.17	6.02	8.52	1.14
50	8.87	17.14	13.26	17.35	2.91
100	14.93	28.97	22.11	30.80	5.30
500	79.02	157.66	117.99	161.71	26.18
750	114.51	228.51	170.41	244.09	38.49
1000	155.60	309.92	231.41	331.82	52.10
5000	778.92	1754.51	1166.75	1613.04	260.27
10000	1556.11	3607.44	2329.07	3423.05	520.09

Fig. 16 Comparison of detection time with volatility plugins

Table 11 Agents' responsibilities in the detected techniques

Agent number	Process injection techniques
1	Classic DLL injection, Reflective DLL injection, Hook injection
2	Classic DLL injection, Reflective DLL injection, .NET DLL injection
3	Classic DLL injection, Reflective (Shellcode) DLL Injection, .NET DLL injection
4	Classic DLL injection, Registry modifications (Persistence)
5	Process hollowing

highest successful detection rate of 99.93% and the lowest false-positive detection rate of 0.068%.

While it is acknowledged that threats cannot be completely eradicated, their impact can be significantly mitigated through ongoing solutions development. This research serves as an initial step toward devising novel methods for detecting an expanded range of process injection attacks. The approach involves searching for malware-induced artifacts rather than directly analyzing the malware itself, primarily focusing on maintaining high detection accuracy and minimizing false-positive rates. Additionally, it is imperative to extend the implementation and testing of the proposed approach to diverse operating systems such as Linux, Mac, and mobile platforms. This extension should account for technological variations, memory layouts, privileges, and other system-specific factors. Furthermore, exploring semi-fileless malware, characterized by generating unintelligible registry keys using non-ASCII characters containing obfuscated malicious code, is an intriguing avenue. In these cases, the malicious code resides on the disk in the form of registry keys, deviating from the conventional structure of traditional malware. Another type stores some files involved in the attack on the disk, such as encrypted payloads and loaders, but the original payload can only be analyzed by intercepting the main script that started the execution, which is difficult.

It is also interesting that machine learning research yielded satisfactory results in detecting certain types of process injection attacks, including DLL injections and Registry modifications. To enhance the efficacy of detection mechanisms, a combined approach can be developed, leveraging machine learning algorithms to comprehensively examine and analyze system memory. This integration aims to create a robust model capable of detecting a wider spectrum of process injection attacks, thereby fortifying the security posture against evolving threats.

Table 12 Example of classic DLL injection detection report

Data name	Value
Process ID	6332
Process name	Notepad.exe
Process path	C:windowssystem32notepad.exe
Process command line	"C:windowssystem32notepad.exe"
Process number of threads	3
Suspicious module name	createremotethread.dll
Suspicious module path	C:UsersMohammedSGNDesktop...createremotethread.dll
Suspicious thread ID	11772
Suspicious thread priority	0
Suspicious thread base address	$0x000002AA59860037$
Suspicious memory protection	PAGE_EXECUTE_READWRITE
Suspicious memory Alloc. protection	PAGE_EXECUTE_READWRITE
Suspicious memory state	MEM_COMMIT
Suspicious memory type	MEM_PRIVATE
TOKEN_USER.User-Sid	DESKTOP-B90A38TMohammedSGN
Attributes	$0x00000000$
TOKEN_OWNER.Owner-Sid	DESKTOP-B90A38TMohammedSGN
TOKEN_PRIMARY_GROUP...	DESKTOP-B90A38TMohammedSGN
TOKEN_TYPE	Token Primary
...	...
Number of detected Attacks	1
Detection time	0.738 seconds

Table 13 Example of Process Hollowing detection report

Data name	Value
Process ID	11964
Process name	Notepad.exe
Process path	C:windowssystem32notepad.exe
Process command line	"C:windowssystem32notepad.exe"
Process number of threads	1
Suspicious module name	processhollowing.exe
Suspicious module path	C:UsersMohammedSGNDesktop...processhollowing.exe
Image size on disk	$0x0000000000038100$
Image size on memory	$0x000000000001D000$
Headers size on disk	$0x0000000000000600$
Headers size on memory	$0x0000000000000600$
DLL characteristics on disk	$0x000000000000D280$
DLL characteristics on memory	$0x0000000000008280$
Imports size on disk	$0x000000000000001E$
Imports size on memory	$0x0000000000000012$
Sections size on disk	$0x0000000000000010$
Sections size on memory	$0x0000000000000010$
Number of detected attacks	1
Detection time	0.987 seconds

Name	Explanation
Pid	6108
Description	Suspicious process notepad.exe
ImageFileName	notepad.exe
DLL	c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll
DllLoadTime	2018-06-04 14:23:27 (UTC+0000)
TimeDifference from previously loaded Dll in sec	113
ThreadPid	6108
ThreadTid	5304
ThreadLoadTime	2018-06-04 14:23:27 UTC+0000
ThreadExitTime	
ThreadHandlePid	5248

Fig. 17 Example of FindDLLInj forensic report [8]

Fig. 18 Example of Malfind forensic report

Fig. 19 Example of Threadmap forensic report

Fig. 20 Example of Hollowfind forensic report

7 Appendix: main tools and experiments

7.1 Main tools

Three main tools were programmed as follows: The first is an integrated project called "Injection Tool" for process injection attacks, where each injection technique is a complete project with sequential steps and lines of code extracted from its sources (Fig. 21).

Secondly, the "Automated Injection Tool" was developed to implement a larger number of attacks at the same time, in addition to implementing more than one type of attack at the same time. It works the same as the Manual Injection Tool but with more options (Fig. 22).

Third, C/C++ was used to develop the "Automated Detection Tool," which enumerates the processes and organizes the work of the agents used in detection (Fig. 23).

7.2 Process injection attacks: execution and detection

This part explains some experiments conducted on the most common process injection attacks to evaluate the performance of the proposed approach. For each experiment, we first analyze the memory state manually using the ProcessHacker 2 software before injection. Then, during the attack, we check the results of the Automated Detection Tool that was developed from the proposed approach in conjunction with monitoring the memory changes using ProcessHacker during and after the attack.
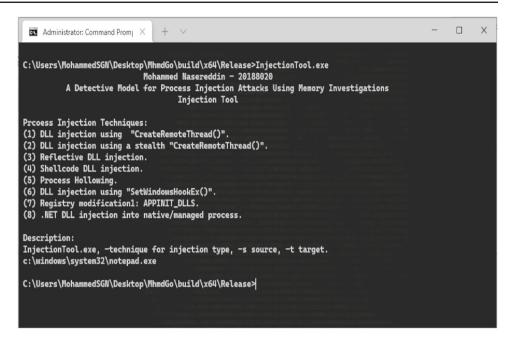
Fig. 21 Manual injection tool

Fig. 22 Automated injection tool

– The first experiment: The attack was carried out on one process using four different attack techniques: DLL Injection via "CreateRemoteThread()", Reflective DLL Injection, DLL Injection via "SetWindowsHook(), and Process Hollowing. The attacks were launched at separate intervals, and each time the tool was able to accurately detect the attack in a time ranging $0.738 - 1.592 Seconds$.

– The Second Experiment: The Automated Injection Tool was used to target a large number of processes simultaneously. In the first execution, 73 processes were injected using the Reflective DLL Injection technique, and the tool succeeded in detecting all of them within $4.156 Seconds$. While in the second execution, 445 processes were injected using the DLL Injection via Stealth "CreateRemoteThread()" technique, and it took $24.478 Seconds$ to detect 444 of the 445 injected processes.

– The Third Experiment: An attack was performed using two different techniques (.NET DLL Injection and Shellcode DLL Injection) on two different processes at the same time. The tool was able to detect them in $2.066 Seconds$.

Fig. 23 Automated detection tool

Declarations

Conflict of interest The authors assert that there is no potential conflicts of interest. The authors have no relevant financial or non-financial interests to disclose.

References

1. Afreen, A., Aslam, M., Ahmed, S.: Analysis of fileless malware and its evasive behavior. In: 2020 International Conference on Cyber Warfare and Security (ICCWS), Islamabad, Pakistan, 2020, pp. 1–8 (2020). https://doi.org/10.1109/ICCWS48432.2020.9292376.

2. Angelystor Process Injection Techniques used by Malware. Accessed July 10, 2022, (2020, June 24). from Medium: https://medium.com/csg-govtech/process-injection-techniques-used-by-malware-1a34c078612c

3. Aslan, Ö.A., Samet, R.: A comprehensive review on malware detection approaches. IEEE Access **8**, 6249–6271 (2020). https://doi.org/10.1109/ACCESS.2019.2963724

4. Attaallah, A., Alsuhabi, H., Shukla, S., Kumar, R., Gupta, B.K., Khan, R.A.: Analyzing the big data security through a unified decision-making approach. Intell. Autom. Soft Comput. **32**(2), 1071–1088 (2022)

5. Almulihi, A.H., Alassery, F., Khan, A.I., Shukla, S., Gupta, B.K., Kumar, R.: Analyzing the implications of healthcare data breaches through computational technique. Intell. Autom. Soft Comput. **32**(3), 1763–1779 (2022)

6. AV-TEST. Malware Statistics & Trends Report | AV-TEST. Accessed May 13, 2023, (2023)from AV-TEST: https://www.av-test.org/en/statistics/malware/

7. AVTEST. The IT Security Status at a Glance: The AV-TEST Security Report 2016/2017. Accessed November 02, 2022, (2017, July 05) from Tech. Rep.: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf

8. Balaoura, S.: Process injection techniques and detection using the Volatility Framework. Master's thesis, University of Piraeus, Piraeus, Greece (2018)

9. Blaam, M.: Great explanation of Process Hollowing (a Technique often used in Malware). Accessed November 2, 2022, from GitHub: https://github.com/m0n0ph1/Process-Hollowing (2021, August 21)

10. Block, F., Dewald, A.: Windows memory forensics: detecting (un) intentionally hidden injected code by examining page table entries. Digit. Investig. **29**, S3–S12 (2019). https://doi.org/10.1016/j.diin.2019.04.008

11. Bridge, K., Abram, N., Kennedy, J., Batchelor, D., Coulter, D., Krell, J., LeBLanc, M.: PE Format. MS Docs. Accessed November 25, 2022 (2021a, November 8)

12. Bridge, K., Sharkey, K., Coulter, D., Jacobs, M., Satran, M.: About event tracing. MS Docs. Accessed December 20, 2022 (2021b, January 7)

13. Bridge, K., Sharkey, K., Coulter, D., Batchelor, D., Satran, M.: Thread handles and identifiers. MS Docs. Accessed November 8, 2022 (2021c, January 7)

14. Chang, T.: Detecting Malware with DLL Injection And PE Infection. Master's thesis, National Sun Yat-sen University, Taiwan (2016)

15. Chen, C., Lai, G., Cai, Z., Chang, T., Lee, B.: Detecting pe-infection based malware. Int. J. Secur. Netw. **16**(3), 191–199 (2021). https://doi.org/10.1504/IJSN.2021.117871

16. Cooper, S.: Fileless malware attacks explained (with examples). Accessed May 18, 2022, (2021, May 14). from Comparitech: https://www.comparitech.com/blog/information-security/fileless-malware-attacks/

17. Cruz, M., de la Pena Perona, M., Rivera, B., Ang, K.: Washington, DC: U.S. Patent and Trademark Office Patent No. 8,572,739 (2013)

18. Dai, Y., Li, H., Qian, Y., Lu, X.: A malware classification method based on memory dump grayscale image. Digit. Investig. **27**, 30–37 (2018). https://doi.org/10.1016/j.diin.2018.09.006

19. Das, S., Mathew, M., Vijayaraghavan, P.: An Approach for optimal feature subset selection using a new term weighting Scheme and mutual information. In: Proceeding of the International Conference on Advanced Science, Engineering and Information Technology, pp. 273–278. Academia, Putrajaya, Malaysia (2011)

20. Duan, Y., Fu, X., Luo, B., Wang, Z., Shi, J., Du, X.: Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs. In: 2015 IEEE International Conference on Communications (ICC), pp. 5691–5696. London, UK, IEEE (2015)

21. Dubyk, M.: Leveraging the PE Rich Header for Static Malware Detection and Linking. SANS Institute, Bethesda, Maryland, United States (2019)

22. Fewer, S.: ReflectiveDLLInjection. Accessed October 26, 2022, (2013, September 5). from GitHub: https://github.com/stephenfewer/ReflectiveDLLInjection

23. Firch, J.: 2021 Cyber security statistics: the ultimate list of stats, data & trends. Accessed September 10, 2021, (2021). from Purplesec: https://purplesec.us/resources/cyber-security-statistics/

24. GitHub, & OpenAI. Your AI pair programmer. Accessed October 22, 2022, (2021). from GitHub Copilot: https://copilot.github.com/

25. Github-milkdevil. injectAllTheThings. Accessed October 29, 2022, (2017, July 21). from GitHub: https://github.com/milkdevil/injectAllTheThings

26. Gorelik, M., Moshailov, R.: Fileless Malware: Attack Trend Exposed. Morphisec Ltd. (2017)

27. Gorelik, M.: Machine learning can't protect you from fileless attacks. Accessed August 27, 2022, (2020, May 13). from SecurityBoulevard: https://securityboulevard.com/2020/05/machine-learning-cant-protect-you-from-fileless-attacks/

28. Hasherezade. Process Doppelganging meets Process Hollowing in Osiris dropper. Accessed September 20, 2022, (2018, September 25). from Malwarebytes Labs: https://blog.malwarebytes.com/threat-analysis/2018/08/process-doppelganging-meets-process-hollowing_osiris/

29. Hosseini, A.: Ten process injection techniques: A technical survey of common and trending process injection techniques. Accessed September 3, 2022, (2017). from Elastic: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

30. Javaheri, D., Hosseinzadeh, M.: A Solution for Early Detection and Negation of Code and DLL Injection Attacks of Malwares. J. Adv. Def. Sci. Technol. 10(4), 393–406 (2020)

31. Javeed, D., Khan, M., Ahmad, I., Iqbal, T., Badamasi, U., Ndubuisi, C., Umar, A.: An efficient approach of threat hunting using memory forensics. Int. J. Comput. Netw. Commun. Secur. 8(5), 37–45 (2020)

32. Khasaia, L.: InjectProc - Process Injection Techniques. (2019, February 10). Accessed October 25, 2022, from GitHub: https://github.com/secrary/InjectProc

33. KSLGroup. Threadmap Volatility Plugin. Accessed November 02, 2022, (2021, August 23) from GitHub: https://github.com/kslgroup/threadmap

34. Li, Y., Li, W., Jiang, C.: A survey of virtual machine system: Current technology and future trends. In: 2010 Third International Symposium on Electronic Commerce and Security, pp. 332–336. Nanchang, China, IEEE (2010)

35. Liang, H., Rugerio, D., Chen, L., Xu, S.: What is a DLL. MS Docs. Accessed February 11, 2023 (2022, January 23)

36. Lim, S., Im, E.: Proposal of process hollowing attack detection using process virtual memory data similarity. J. Korea Inst. Inf. Secur. Cryptol. 29(2), 431–438 (2019). https://doi.org/10.13089/JKIISC.2019.29.2.431

37. Liu, W., Steven, G.: A free, powerful, multi-purpose tool that helps you monitor system resources, debug software and detect malware. Accessed October 2, 2022, (2021). from Process Hacker: https://processhacker.sourceforge.io/

38. Microsoft Developer. Download a Windows 10 virtual machine. Accessed September 22, 2022, (2021). from Microsoft Developer: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

39. Mikben Batchelor, D., Sharkey, K., Coulter, D., Kennedy, J., Satran, M.: Memory Protection Constants. MS Docs. Accessed October 11, 2022 (2021, March 22)

40. Mikben, Sharkey, K., Satran, M.: About Memory Management. MS Docs. Accessed November 8, 2022 (2021, January 7)

41. Mohd Yusof, M., Mokhtar, M.: A review of predictive analytic applications of bayesian network. Int. J. Adv. Sci. Eng. Inf. Technol. 6(6), 857–867 (2016). https://doi.org/10.18517/ijaseit.6.6.1382

42. Monnappa, K.: Detecting deceptive process hollowing techniques using hollowfind volatility plugin. Accessed August 25, 2022, (2016a, September 22). from Cysinfo: https://cysinfo.com/detecting-deceptive-hollowing-techniques/

43. Monnappa, K.: Hollowfind Volatility Plugin. Accessed August 25, 2022, (2016b, September 24). from GitHub: https://github.com/monnappa22/HollowFind

44. Monnappa, K.: Psinfo Volatility Plugin. Accessed August 25, 2022, (2016c, September 24). from GitHub: https://github.com/monnappa22/Psinfo

45. Mosli, R., Li, R., Yuan, B., Pan, Y.: A behavior-based approach for malware detection. In: IFIP International Conference on Digital Forensics, pp. 187–201. Springer, Cham, Orlando, FL, USA (2017)

46. Mosli, R., Li, R., Yuan, B., Pan, Y.: Automated malware detection using artifacts in forensic memory images. In: 2016 IEEE Symposium on Technologies for Homeland Security (HST), pp. 1–6. IEEE, Waltham, MA, USA (2016)

47. Otsuki, Y., Kawakoya, Y., Iwamura, M., Miyoshi, J., Faires, J., Lillard, T.: Toward the analysis of distributed code injection in postmortem forensics. In: 14th International Workshop on Security, IWSEC 2019. 11689, pp. 391–409. Tokyo, Japan: Springer, Cham (2019)

48. Pingios, A., Beek, C., Becwar, R.: Process injection, technique T1055 - enterprise. Accessed November 8, 2022, (2017, May 31). from MITRE ATT&CK: https://attack.mitre.org/techniques/T1055/

49. Rathnayaka, C., Jamdagni, A.: An efficient approach for advanced malware analysis using memory forensic technique. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 1145–1150. IEEE, Sydney, NSW, Australia (2017)

50. Red Teaming Experiments. Code & Process Injection. Accessed November 5, 2022, (2021). from ired.team: https://www.ired.team/offensive-security/code-injection-process-injection

51. Sahu, K., Srivastava, R.K.: Needs and importance of reliability prediction: an industrial perspective. Inf. Sci. Lett. 9(1), 33–37 (2020)

52. Sahu, K., Srivastava, R.K.: Predicting software bugs of newly and large datasets through a unified neuro-fuzzy approach: reliability perspective. Adv. Math. Sci. J. 10(1), 543–555 (2021)

53. Sahu, K., Srivastava, R.K., Kumar, S., Saxena, M., Gupta, B.K., Verma, R.P.: Integrated hesitant fuzzy-based decision-making framework for evaluating sustainable and renewable energy. Int. J. Data Sci. Anal. 16(3), 371–390 (2023)

54. Sahu, K., Alzahrani, F.A., Srivastava, R.K., Kumar, R.: Evaluating the impact of prediction techniques: software reliability perspective. Comput. Mater. Continua 67(2), 1471–1488 (2021)

55. Sahu, K., Alzahrani, F.A., Srivastava, R.K., Kumar, R.: Hesitant fuzzy sets based symmetrical model of decision-making for estimating the durability of web application. Symmetry 12(11), 1770 (2020)

56. Sahu, K., Srivastava, R.K.: Soft computing approach for prediction of software reliability. Neural Netw. 17, 19 (2018)

57. Salman, M., Husna, D., Viani, N.: Static Analysis Method on Portable Executable Files for REMNUX based Malware Identifi-

cation. In: 2019 IEEE 10th International Conference on Awareness Science and Technology (iCAST), pp. 1–6. IEEE, Morioka, Japan (2019)

58. Sihwail, R., Omar, K., Ariffin, K.: An effective memory analysis for malware detection and classification. CMC-Comput. Mater. Continua **67**(2), 2301–2320 (2021). https://doi.org/10.32604/cmc.2021.014510

59. Sihwail, R., Omar, K., Ariffin, K.: A survey on malware analysis techniques: static, dynamic, hybrid and memory analysis. Int. J. Adv. Sci. Eng. Inf. Technol. **8**(4–2), 1662–1671 (2018). https://doi.org/10.18517/ijaseit.8.4-2.6827

60. Srivastava, A., Jones, J.: Detecting code injection by cross-validating stack and VAD information in windows physical memory. In: 2017 IEEE Conference on Open Systems (ICOS), pp. 83–89. IEEE, Miri, Malaysia (2017)

61. Subedi, K., Budhathoki, D., Dasgupta, D.: Forensic analysis of ransomware families using static and dynamic analysis. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 180–185. IEEE, San Francisco, CA, USA (2018)

62. Teller, T., Hayon, A.: Enhancing automated malware analysis machines with memory analysis. London, England and Wales: BlackHat, InformaTech. Retrieved from https://www.blackhat.com/docs/us-14/materials/arsenal/us-14-Teller-Automated-Memory-Analysis-WP.pdf (2014)

63. Thompson, E.: Cybersecurity Incident Response: How to Contain, Eradicate, and Recover from Incidents, 1st edn. Apress, New York, USA (2018)

64. VMware Docs. VMware Workstation 15.5.1 Pro Release Notes. Accessed September 22, 2022, (2019, November 12). from VMware Docs: https://docs.vmware.com/en/VMware-Workstation-Pro/15.5/rn/VMware-Workstation-1551-Pro-Release-Notes.html

65. Volatility Foundation. The Volatility Foundation - Open-Source Memory Forensics. Accessed March 29, 2023, (2020). from VolatilityFoundation: https://www.volatilityfoundation.org/

66. Webb, M.: Evaluating tool based automated malware analysis through persistence mechanism detection. Doctoral dissertation, Kansas State University, Manhattan, USA (2018)

67. White, A., Schatz, B., Foo, E.: Integrity verification of user space code. Digit. Investig. **10**, S59–S68 (2013). https://doi.org/10.1016/j.diin.2013.06.007

68. Xiao, C., Zheng, C.: New IoT/Linux Malware Targets DVRs, Forms Botnet. Accessed September 19, 2022, (2017, April 6). from Paloaltonetworks: https://unit42.paloaltonetworks.com/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/

69. Yadav, A., Garg, M.: Docker containers versus virtual machine-based virtualization. In: Emerging Technologies in Data Mining and Information Security, pp. 141–150. Springer, Singapore (2019)

70. Yosifovich, P., Solomon, D., Ionescu, A.: Windows Internals, Part 1: System architecture, processes, threads, memory management, 7th edn. Microsoft Press, Redmond (2017)

71. Zadeh, L.: Fuzzy logic. Computer **21**(4), 83–93 (1988). https://doi.org/10.1109/2.53

72. Zhang, S., Hu, Y., Bian, G.: Research on string similarity algorithm based on Levenshtein Distance. In: 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), pp. 2247–2251. IEEE, Chongqing, China (2017)