Vietnam National University
Ho Chi Minh city University of Techonology
Faculty of Computer Science and Engineering

Windows Memory Forensics

Finding hidden processes in a running machine

By

NGUYEN Anh Khoa - 1611617

A thesis proposal submitted to the Ho Chi Minh City University of Technology in accordance with the requirements of the Degree of Engineer in Computer Science.

Instructors: NGUYEN An Khuong
NGUYEN Le Thanh
NGUYEN Quoc Bao
Opponent: Dr. LE Thanh Sach

Ho Chi Minh City, December 2019

Abstract

Computers have become a tool that humans around the world use daily on for their study, work and entertain. With the capability to solve complex problems, it has become a necessity in our life. The widespread of computers also bring crime as hackers have been trying to damage the system or collect confidential information. Malware and attacks happen almost every day, threating our data. While anti-virus software and firewall cannot prevent new malware bringing new attack vectors, digital forensics act as a post-event investigation to collect and examine the attack. Memory forensics, a branch of digital forensics, dig into an infected machine finding hidden malware to analyze their behaviour. The job is done exclusively by experts and not available to the average user. If a regular user can know about the malware infecting his computer, we will be able to detect and prevent the possible spread of malware before the attack. Thus, we analyzed memory forensics techniques and proposed an implementation of a tool for finding hidden processes in a running machine, in the hope that this tool will run automatically and available to regular users as well.

TABLE OF CONTENTS

List of Figures

List of Tables

Listings

List of Abbreviations

OS	Operating System
I/O	Input and Output
CLI	Command Line Interface

CHAPTER 1. Introduction

1.1. Motivation

Throughout the years of computer development, computers have become a standard method for humans around the world to study, work and entertain. Most activities of our daily lives involve computers. Individuals across the globe have created systems running on computers to assist them in doing common and complex tasks. However, this somehow influenced other people to commit harmful activities, these people often refer to as *hacker*. Hackers have been creating software to cause damage and steal confidential or private information, these software are malware. Furthermore, lately along with the trend of cryptocurrency, while ordinary people commit their investment by using crypto mining machines, hackers, on the other hand, create sophisticated software (cryptojacking malware) that stealthily installed on a victim machine and mine cryptocurrency without the victim's acknowledgement.

Security researchers have been struggling to find ways to mitigate the gaining rate of attacks. However, it was never close to perfection. Security researchers uses file checksum and unique bytes sequences in file to create a file signature and combines a list of malware signature to a database, for example yararules [3]. Relying on file signature database for filtering file often miss out new one, thus, it is highly vulnerable to the newer class of malware. To counterattack these new malware when an attack happens, digital forensics is performed. Digital forensics which as described by The Forensics Research Workshop I [21]:

"The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate

unauthorized actions shown to be disruptive to planned operations."

Digital forensics includes many different aspects, however, the most intrigued part of digital forensics is memory forensics, which "provides unprecedented visibility into the runtime state of the system, such as which processes were running, open network connections, and recently executed commands" as stated in the book The Art of Memory Forensics [12]. Memory forensics provide a frame of the computer state, from which one can extract files and processes. Researchers often extract suspicious files from these memory samples, then reverse engineer it, if the file is a malware, they will create the file signature and add to the database. Digital forensics and memory forensics is very important to gain insight on the cyber attacks made by hackers. However, prevention is always better than fixing, and with the rise of the more steathy malware, the computer security industry is facing a major problem.

Software will not be considered as malware until they are found doing malicious behaviours. Therefore hackers create malware hidden from the Task Manager and other process listing tools. From the early days of 1990s, they have been improving ways to hide malware. In 2017, a small report [7] have revised on hiding techniques in malware over the years in the Windows OS. Most of these techniques is either preventable or mitigable. However, even in 2019, we can still observe incidences where malware hide itself so effectively. For example, Android malware that hid themselves on user mobile and was only found September 2019 after 5 months on the Google Play Store [17], or the *Titanium* backdoor[1] on Windows 10 disclosed November 2019 [32], or the macOS malware, named *unioncrypto*, that downloads and runs a hidden process in memory to mine cryptocurrency was only discovered December 2019 [35]. Researchers collected and analyzed these malware, but a normal user would not be able to do that. For a normal user, if the anti-virus software failed to flag the file as mallicious, his computer will be infected. To know whether a system is having a hidden malware running, one must send an investigator a memory extraction. Such process is complex, long and costly, a user must know how to extract the memory and hire

[1]Programs that receives remote connections

an investigator to analyze. If there were a tool to do memory forensics live, while the system is running, and extract the hidden malware automatically, it would be easier for a normal user. Looking in live memory forensics, Shuaibur Rahman and Khan [22] has a review of live forensics analysis techniques in 2015, and in the review there is one work that can extract running, finished, and cached processes [1]. However, the work is only restricted to the Linux OS and requires an experienced investigator to do. Besides that, there are projects that implements memory forensics techniques but is limited to dump file analysis. We have researched for live memory analysis and have come up with a method for finding hidden processes.

Within the scope of the thesis, we will create a tool finding hidden processes aims at normal users. The tool could also submit the binary along with the process memory to the remote server. And with the market share of Windows over others OS is more than 75% [19], within the Windows OS, version 10 is now dominating with more than 60% [36], this tool will focus on Windows 10 live memory analysis.

1.2. Objectives

In the scope of this thesis, we wish to:

- Understand the basics concept of OS that supports memory forensics,
- Understand to some extent of Windows internals,
- Understand some techniques often used to do Windows memory forensics,
- Analyze some already existed tools doing memory forensics,
- Propose a method to find hidden running processes in a running Windows machine,
- Implement the method in a demonstrating version and perform evaluation.

1.3. Structure

- Chapter 1 gives an overview, the goals and challenges of this thesis proposal,
- Chapter 2 presents the background on OS, Windows Internal and Digital forensics techniques,
- Chapter 3 introduces some products doing digital forensics,
- Chapter 4 suggests a method to perform live memory forensics on Windows 10,
- Chapter 5 states some challenges the proposed method might encounter and gives directions for implementation.

1.4. Timeline

In the next stage of the thesis, we will finish the followings works:

- January to Feburary 2019: Implementation of the tool base on proposed method,
- March 2019: Perform testing on Windows 10 and older version,
- April 2019: Resolve the addressed challenges.

CHAPTER 2. Background

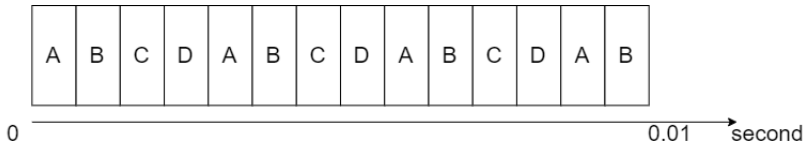
2.1. Operating system concepts

In this section, we will revise some concepts of OS on multiprogramming system. Multiprogramming defines a computer where it can run several programs simultaneously. To enable multiprogramming, the OS must be capable of managing memory region for processes, scheduling running time for processes, and many more crucial tasks. We will briefly review the two aspects of multiprogramming.

2.1.1. Scheduling

The first aspect of reviewing is scheduling. Cooperative multitasking, or so-called non-preemptive multitasking, is a design for multiprogramming scheduling where it does not limit a process's runtime. A process will actively run until it stops/idles or waits for an I/O operation, whence the operating system will change (switch context) to another process. In contrast with cooperative multitasking, preemptive multitasking limits the process's runtime and initiate a context switch when either the time allotted is used or the process waits for an I/O operation, Figure 2.1. Both schemes have been used in many operating systems. However, cooperative multitasking schema is not suitable for a multitasking OS, and thus in modern operating systems for general usage (Windows, macOS, and most Linux distributions), preemptive multitasking schema is chosen.

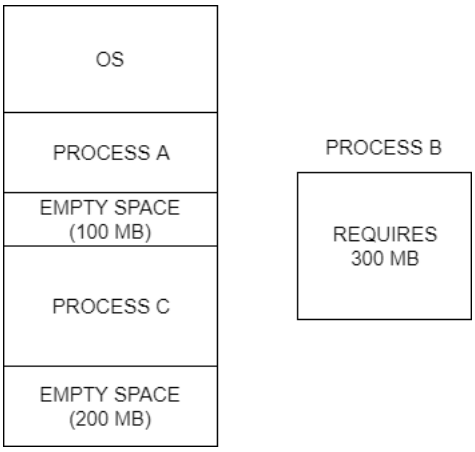
Figure 2.1: *Preemptive scheme*



2.1.2. Memory management

The second aspect of OS in discusses is memory management. In the early days, memory management was so simple that it would contain the whole process consecutively in memory. However, doing this leads to external fragmentation, where the memory is empty but not fit consecutively for a process, Figure 2.2. OS developers soon realized storing the whole process in memory is not optimize when programs only need a part of memory at a specific time when running. They came up with ways to load only the necessary parts in memory. After many years, people have agreed upon splitting a process to equal parts called a *page*. The OS splits the physical memory to pages and load processes data to those pages, Figure 2.3. When a process needs more space or needs parts that are not on the memory, the OS will find an empty/unused page to load in. If there is no empty page, one least use page will be moved (swapped out or page out) to disk to make space. The OS tracks what pages a process is using and swap those pages in and out by process need.

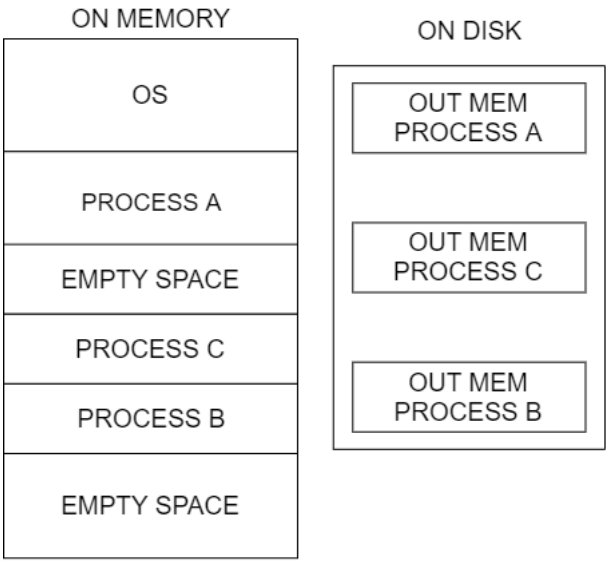
Figure 2.2: *External fragmentation*



So far, we have only discussed processes that have a limited amount of memory, but the OS allows programs to have more space by using the paging technique describe above. Because process data can be moved to disk and brought back when needed, a process can have as many spaces as the disk space available. However, by most OS implementation, a process should have a limited virtual address space. Because the

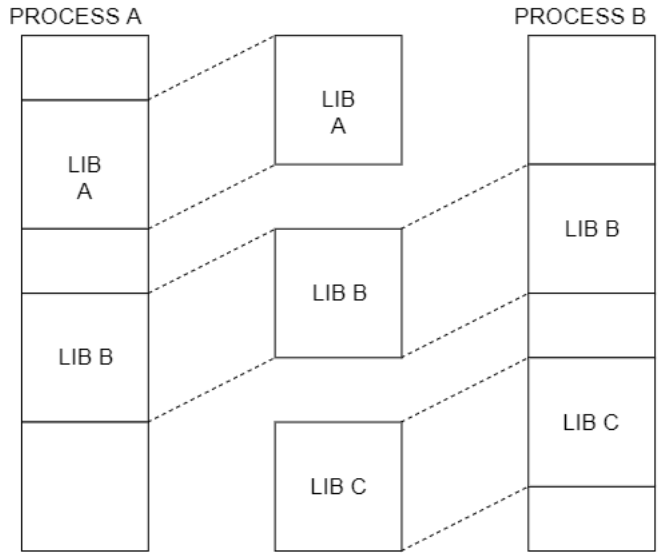
address is virtual to the process, every memory access will need the OS to change to the address in the physical memory (after page swapping). Changing address from virtual to physical is called *address translation*. By using virtual memory, the OS can map the same data to many processes and reuse data. For example, two processes use the same library in Figure 2.4.

Figure 2.3: *Splitting process*



Example of an address translation. Assuming the OS allows a process to have 2GB of (virtual) memory space (from 0 to $2^{32} - 1$), and a page will have the size of 4KB, and the system has 4GB of physical memory. A process loading at 0x40000000 will have all data from 0x40000000 to (0x40000000 + 4KB) loaded to physical memory at the address of 0x60000000 to (0x60000000 + 4KB).

Figure 2.4: *Two processes use the same library*



2.2. Windows Internals

To understand memory forensics on Windows, knowledge about Windows Internals is necessary, from the memory model to some kernel design. In this section, we will give the reader some technical details on Windows' design.

2.2.1. Memory model

We start by exploring the Windows memory model, and more importantly, the Windows kernel memory model. Windows by default use page size of 4KB, the virtual address space for each process is 2GB for the 32-bit version and 8TB for the 64-bit version. The system address space is 2GB for the 32-bit version and 248TB for the 64-bit version. For every process, when run, can see both its own address space and the system space, but access to the system space is restricted. A typical 32-bit process can have a 2GB space for itself and a 2GB space of the system, makes a total virtual memory space 4GB. The system space, refer to as the kernel space, contains the OS kernel objects and the currently running drivers and kernel application. The kernel space uses *pools* to manage objects allocation and deallocation. Windows has two

types of pool, one that is *pagable*, can swap out to disk, called *paged pool* and the other *non-paged pool*. The size of these two types of pools on Windows 10 can be referred through the Table 2.1 [14]. Because many parts of the OS is not used often, Windows can swap those pages out to make more space. However, some information will always remain in the non-paged pools.

Table 2.1: *Pool size on Windows 10*

Pool Type	**Limit on 32-bit**	**Limit on 64-bit**
Paged Pool	384 GB or system commit limit, whichever is smaller	384 GB or system commit limit, whichever is smaller
Non-paged Pool	75% of RAM or 2 GB, whichever is smaller.	RAM or 128 GB, whichever is smaller (address space is limited to 2 x RAM)

The memory pool is a bitmap, an array of bits. Windows will return the pointer and size when asked for space in the pool (chunk). Windows keeps track of chunks in the pool. At first, there is only one chunk (the pool itself). When the user asks for space, Windows will split the chunk into two, one for the user and one left unused, Windows will keep splitting the unused space in the pool and return to the user for any allocation requested. Upon deallocation, Windows will merge unused chunks into one. Consider the code in Listing 2.1 for a basic example of pool allocation.

Each chunk will have a `POOL_HEADER` field on the top to denote the content of the chunk. `POOL_HEADER` has a size field (`BlockSize`), a previous size field (`PreviousBlockSize`), and a tag (`POOL_TAG`). Tag is a four-byte character that Windows and Driver writer use to denote the data stored in the chunk. In Table 2.2, we listed some typical structure with its tag.

Listing 2.1 *Basic Pool Algorithm*

```
#include<vector>
using std::vector;
typedef struct POOL {
  int size;
  char* const location;
  int32_t tag;
  int previousSize;
} _POOL, *PPOOL;

constexpr unsigned int MAX_POOL = 1000;
char POOL_BITMAP[MAX_POOL];
vector<PPOOL> CHUNKS;

void write(PPOOL p, unsigned int size, void* data) {
  char* d = (char*)data;
  if (size > p->size) return;
  for (int i = 0; i < size; i++) p->location[i] = d[i];
}

void writePoolHeader(PPOOL p) {
  write(p, sizeof(POOL), (void*)p);
}

PPOOL alloc(unsigned int size, int32_t tag) {
  unsigned int actual_size = size + sizeof(POOL);
  if (chunks.empty()) {
    if (size <= MAX_POOL) {
      PPOOL p = new _POOL{actual_size, &POOL_BITMAP[0], tag};
      writePoolHeader(p);
      CHUNKS.push_back(p);
      PPOOL pp = new _POOL{MAX_POOL - actual_size,
                           &POOL_BITMAP[size],
                           0, size};
      CHUNKS.push_back(pp);
      return p;
    }
    return nullptr;
  }
  PPOOL lastChunk = CHUNKS.back();
  PPOOL newChunk = new _POOL{lastChunk->size - actual_size,
                             lastChunk->location + size,
                             0, size};
  lastChunk->size = actual_size;
  writePoolHeader(lastChunk);
  CHUNKS.push_back(newChunk);
  return lastChunk;
}
```

Table 2.2: *Some pool tags and their corresponding structure*

Structure	Structure Name	Pool Tag
Driver Object	_DRIVER_OBJECT	Driv
File Object	_FILE_OBJECT	File
Process	_EPROCESS	Proc
TCP endpoint		TcpE
TCP listener		TcpL
Thread	_ETHREAD	Thre
UDP endpoint		UdpA

2.2.2. EPROCESS and ETHREAD

Windows has a special structure that contains process information called `_EPROCESS`. This structure is created every time when a new process spawns and located inside a non-paged pool with tag *Proc*. If we have a reference to one `_EPROCESS` we might follows `LIST_ENTRY ActiveProcessLinks`, a doubly-linked list pointer, to find other `_EPROCESS` of different process Figure 2.5. However, because a normal user can access this structure by calling `PsLookupProcessByProcessId` one may edit the doubly linked list to remove itself from the list chain as can be seen in Figure 2.6.

Listing 2.2 *LIST_ENTRY*

```
1 typedef struct _LIST_ENTRY {
2       struct _LIST_ENTRY *Flink;
3       struct _LIST_ENTRY *Blink;
4 } LIST_ENTRY, *PLIST_ENTRY, PRLIST_ENTRY;
```

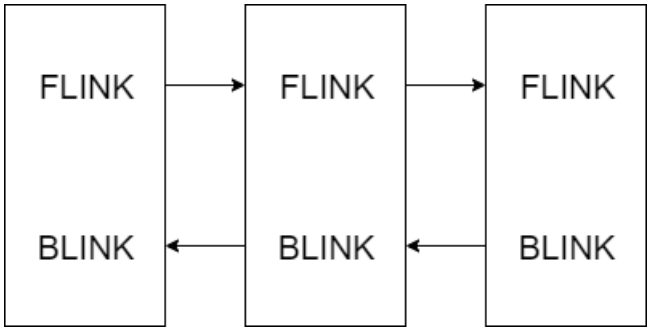


Figure 2.5: *EPROCESS linking*

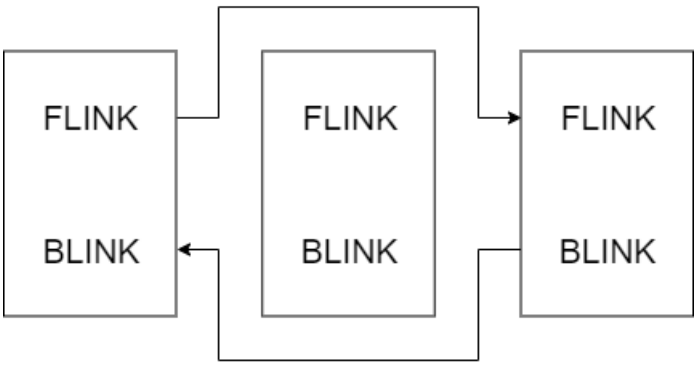


Figure 2.6: *EPROCESS linking modified*

A process in Windows is composed of threads. For example, a process loading a library will have the library run as a thread. Every thread will have a kernel object inside the non-paged pool called `_ETHREAD`. From an `_ETHREAD` by calling `IoThreadToProcess`, Listing 2.3, one can get the parent `_EPROCESS`. An `_EPROCESS` has a doubly-linked list of `_ETHREAD` in `LIST_ENTRY ThreadListHead` to list threads of a process.

Listing 2.3 *IoThreadToProcess*

```
1 PEPROCESS IoThreadToProcess(
2   PETHREAD Thread
3 );
```

2.2.3. KDBG

KDBG short for `Kernel Debugger Block` is one of the commonly used structure when analyzing a memory artefact of Windows. It was created to easy debugging process for developers when writing OS or kernel drivers as this structure contains pointers to other structures. One pointer contained in KDBG is `PsActiveProcessHead` which is a pointer to an `_EPROCESS`. As previously mentioned, we can walk the list of `_EPROCESS` to enumerate all processes.

2.2.4. Windows dump file

Windows provides a dump file to contains compressed memory data. This file is created by Windows when an error in kernel occur. Third-party tools can dump the

memory on running machine, for example, DumpIt, FTK Imager. This file structure is not documented but Schuster [27] has written a guide to the file structure. Another dump file type is minidump file which is documented by Joachim [15]. These dump files are commonly used to preserve the system memory state for analysis.

2.2.5. Process Injection

Process injection is a technique to inject a process to another running process. In 2019, Amit Klein and Itzik Kotler gave a talk at the US Blackhat conference about the topic [2]. The research is done on Windows 10 x64, and they tried to systemize the process injection techniques. Some techniques could still be used although Windows 10 has better security than previous Windows version. A process injected will be more stealthy since the system can only see the parent process. This method is often used by malware to make their process hidden from the system. For any process spawn, a thread with `_ETHREAD` is always created, and investigators often rely on this to detect hidden threads.

2.3. Digital Forensics and memory forensics techniques

Digital forensics describes the process includes the collection and analyzation of evidence; information synthesization; and sometimes unknown binaries and files analysis. Digital forensics has become significant in the industry. People will turn into digital forensics companies when they notice or doubt one computer contains illegal files or malicious applications. Without digital forensics, we cannot traceback when an attack happens, and without prior learning of historical events, we can not build a more reliable system to detect and prevent future replicate attacks. We learn and build filters based on files signature [3]. These signatures can be unknown from a Zero-day attack, and even a One-day attack can still harm the system. Digital forensics is gaining more and more important when the next generation of malware do not intend to break the system but to exploit the system computing power for profit or collect

sensitive information. Most notably are Cryptocurrency Malware and Botnet; these types of malicious applications will run inside the system and uses little computations. Backdoors and trojans are another concern when they silently collect user's information, files and activities. We can only know of their existence when the attack had happened.

2.3.1. Evidence gathering

Once we have identified the corrupted system, we must try to disconnect the machine from the internet. However, if we disconnect the system from the internet, the malware will drop all connection, and we will lose all traces. Hence, we must quickly dump the memory of the system and disconnect the machine from the internet. An investigator can collect the information below from a memory dump.

- Currently running processes
- Currently opening files (file descriptor)
- Currently opening sockets

After memory dump file acquisition, we use forensics tools and begin analysis. In the worst condition, where we must shut down the system, we can try to collect the whole physical memory by using *cold boot attack* [10]. This technique relies on memory data dissolve slower on low temperature, enable us to shut down and collect RAM data given physical access. Full RAM dump can easily be converted to dump file format to use with tools. After raw memory is gathered, we will start analyzing. The two go-to tools for memory forensics are Google's Rekall platform and the open-source Volatility project. Most digital forensics individuals and teams use these two tools. Next, we will describe some techniques on memory forensics.

2.3.2. KDBG scanning

Scanning KDBG will provide some small but important piece of information. As described in Section 2.2.3, KDBG has a member `PsActiveProcessHead`, which is a

pointer to the doubly linked list of _EPROCESS. Volatility used KDBG scanning to characterize Windows version before Windows 8. From Windows 8 forward, this structure is encoded and made a hindrance to Volatility users [33].

"An encoded KDBG can have a hugely negative effect on your ability to perform memory forensics. This structure contains a lot of critical details about the system, including the pointers to the start of the lists of active processes and loaded kernel modules, the address of the PspCid handle table, the ranges for the paged and non-paged pools, etc. If all of these fields are encoded, your day becomes that much more difficult."

While users of Volatility will suffer a bad day, Rekall's users do not as Rekall uses another method without the knowledge of KDBG structure [24]. However, KDBG is still a critical Windows structure that provides kernel information and still being used to get general information of Windows out of a dump file.

2.3.3. Pool tag scanning and quick scanning

Pool tag scanning was first introduced by Schuster [28] in 2006. By utilizing the pool layout with POOL_HEADER's POOL_TAG, scanning for tags that contain a certain structure is possible. With the larger physical memory in these recent years, scanning the whole address space will soon unsuitable. In 2016, Sylve, Marziale and Richard [30] has improved the algorithm for 64-bit Windows by using a global kernel variable indicating the start of dynamic allocation of the pool MiNonPagedPoolStartAligned and a pointer to the allocation bitmap MiDynamicBitMapNonPagedPool. Their work has been proven to be much faster and work effectively on huge memory. Even today, pool tag scanning is still being used often to identify structure not found by the kernel structure indexing. Thus, pool tag scanning is used to search for hidden processes by finding _EPROCESS that are not found by traversing the list of _EPROCESS.

CHAPTER 3. Related works

3.1. Volatility

Volatility [8] is an open-sourced forensic tool first developed by Aaron Walters and Petroni. They created a tool called Volatool in 2000, later changed its name to Volatility, and by now, it has gained much popularity in the digital forensics world. This work combines years of digital research into a convenient and versatile tool. The Volatility organization also has a contest every year to help improve the tool. This tool allows forensics investigators to analyze a memory dump file of Windows, macOS and Linux. It is designed by plugins of different version of OS, including major, minor and build version. These plugins specify constant numbers, kernel structure definitions and other OS-related information.

When provided a dump file and a command as an input, Volatility will parse the dump file and operate the command. For finding processes Volatility provides these command *pslist*, *psscan*, *psxview*. *pslist* will walk through the `PsActiveProcessHead` doubly linked list found in KDBG, while *psscan* will do pool tag scanning. *psxview* will compare the result of multiple commands with *pslist* to find processes tries to hide from the system.

3.2. Rekall

Rekall [9] is an "advanced forensic and incident response framework" created by Google. Rekall implements techniques in the field and also maintain as an open-source project. Rekall also supports Windows, macOS and Linux memory forensics. One part different from Volatility is that Rekall can do live analysis. Rekall can dump the current memory or load a driver to run analysis in the current system. It is a common practice

for an investigator to connect to a machine remotely and do forensics using Rekall. Unlike the Volatility plugins system, Rekall uses debug information of OS vendor to find the correct data structure without having to write out and also works with rare OS version manually.

For finding processes Rekall has commands for listing processes and scanning the pool, *pslist* and *psscan*. Rekall can do live analysis when we provide `--live` to its CLI argument. With this option enabled, Rekall will run a driver inside the system, and we can apply the command to analyze.

3.3. Memtriage

Memtriage [11] is a tool combining Rekall driver and Volatility tool to query live RAM artefacts. This tool works for a live machine. However, it may fail on Windows 10. The author of this tool has recorded the blue screen errors on some Windows 10 builds.

3.4. Windows SysInternal suite

The Windows SysInternals [25] suite is a set of software for internal Windows monitoring and inspection. The suite not made for forensics, but some software inside can help investigators inspect the system to find malware. The two software *process explorer* and *process monitor* are used often. Process explorer will give a list of processes, for each process, the software list threads running inside. *process explorer* also sends file checksum to VirusTotal for checking signature online.

We were not able to find any evidence whether *process explorer* can find hidden process. However, *process monitor* will log activities such as the file I/O, networking. If a hidden process does any activities that is logged by *process monitor*, we can see the process in the log.

3.5. Overview

Even there are many related works, all of which require people with experience in forensics to use. For Volatility, even with a versatile toolset, it cannot work live out of the box. Rekall project can do live forensics, but the project is stagnant. There has been no commitment to the Rekall project since March 2019 despite having around 180 issues opened in December 2019. Memtriage does live memory forensics but fail to work on Windows 10.

As the target we set is a tool for the average user and work on Windows 10, none of the above works meets our target. However, these projects influenced our research. In the next chapter, we will discuss the proposed method.

CHAPTER 4. The proposed method

With the pool tag scanning described in Section 2.3.3, we proposed a way to do it live. The steps are listed below for clarity:

Step 1. Run a controlled driver in kernel space

Step 2. Perform pool tag scanning

Step 3. Deserialize the bytes to get processes information

Step 4. Find potential hidden process and send to the server

Our program should have two parts, part that runs inside the kernel space, `K`, and part that run in user space, `U`. We will perform pool tag scanning with `K` and for every successful findings, `K` will send the structure as bytes to `U`. From that `U` will try to extract information by deserialize the bytes to a defined structure.

4.1. Run a controlled edriver in kernel space

Our tool needs a process, which is a kernel driver in this case, in order to gain access to the kernel space. The kernel driver can be loaded while booting the machine or by calling the undocumented Windows API – `NtLoadDriver(PUNICODE RegistryPath)`. Both ways need a key that is registered at a specific path in the Windows registry, `\registry\machine\SYSTEM\CurrentControlSet\Services`. The key must have three values:

- `ImagePath`: the path of the driver to be loaded,
- `Start`: an enumeration specifing when to start the driver, manually or on boot,
- `Type`: an enumeration specifing the type of the driver.

To load the driver, the `Start` value should be `2` for automatically start and `3` for manually start. The `Type` value should be `1` for specifying the driver is in the type

of kernel device. Once the driver is loaded and run, we have permission in the kernel space.

4.2. Perform pool tag scanning

Performing pool tag scanning for hidden processes requires a pointer to an address in the non-paged pool. Calling an allocation through `ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag)` will return a pointer to a chunk pool. By calling the function with `PoolType = NonPagedPool`, a pointer inside the non-paged pool will be returned. Using this pointer to scan the pool by the given algorithm, for each positive hit of `_EPROCESS`, the driver will send the bytes of the structure to the user space process `U`. Continue scanning to get all the structures in the pool.

4.3. Deserialize the bytes to get processes information

After we have collected the raw bytes, they need to be deserialized to their correct structure based on Windows build. The Windows kernel structures change drastically over every builds [6]. Thus, `_EPROCESS` layout is not fixed across Windows builds. Kernel structure layout is important for driver developers, so Windows has a file to record these structures layout and distributed with every Windows builds. Those files are called PDB (program database). Windbg, the Microsoft debugger, and Visual Studio both use these file for debugging purposes. Microsoft hosts these files at `https://msdl.microsoft.com/download/symbols`. However, the download protocol is not HTTP. To download these files, one may need to use Windbg. Another option is to use *pdb-downloader* by Rajkumar Rangaraj [23], the code is open-source and written in *C#*. The protocol can be understood through the code and re-implement to fit our needs.

The PDB files are now available, but, the file is not parsable now. The file format is not officially documented, Microsoft only gave the community an incomplete repository containing the PDB file information [16]. Fortunately, community over time have

written many parsers, from these open-source parser, we can use to parse the PDB files and get `_EPROCESS` structure definition. With the correct `_EPROCESS` definition, the bytes will be deserialized in order to get information from the bytes. The information are the process identification number (process id), the process creation and exit time, and the file path. The information can be retrieved from these members of `_EPROCESS`:

- Process id: `PVOID UniqueProcessId`
- Process create time: `LARGE_INTEGER CreateTime`
- Process exit time: `LARGE_INTEGER ExitTime`
- File path: `UCHAR ImageFileName[16]`

`UniqueProcessId` is a pointer, in this case, the value must be dereferenced. The value is inside the kernel space, so the tool will request the driver to fetch the value. When all processes information is gathered, we have completed step 3.

4.4. Find potential hidden process

In step 4, with all the processes information collected from pool tag scanning, we will attempt to filter out normal processes. Normal processes are visible to the system and can be enumerated by traversing the `LIST_ENTRY ActiveProcessLinks` of any process. To get an `_EPROCESS`, we will call `PsLookupProcessByProcessId`, Listing 4.1, with the tool id. Filtering out process that is not on the list after traversing the `_EPROCESS` will yield a list of potentially hidden processes.

Listing 4.1 *PsLookupProcessByProcessId*

```
1 NTSTATUS PsLookupProcessByProcessId(
2   HANDLE    ProcessId,
3   PEPROCESS *Process
4 );
```

With the list of process id of malicious processes, we will dump the process memory and collect the binary file. For dumping a process from memory, we can use a tool created by Geoff McDonald [13]. The binary file can be collected through the file path string extracted. Zipping all the dump files and binaries, we send the zip file to the

server for further analysis. If the researchers find the file is a malware, they will write a script to stop and remove the malware from the computer and send back to the tool running for automatic removal. Ofcourse, there are time taken to analyze new sample, but we can notify the user through email that there is a malware running and ask the user to open the tool, enter the malware id, and receive automatic malware removal.

CHAPTER 5. Challenges and conclusion

5.1. Challenges

With the given proposed method above, we will name out some challenges the method might have and ways we can resolve those challenges.

Firstly, the proposed method is based on pool tag scanning, and thus, we should hit the same problem where pool tag scanning has. Pool tag scanning can scan deleted or freed process, where the pool chunks are merged but not overwritten. Pool tag scanning uses tag value, but any kernel driver can name these tag values, a hacker may write malware that allocates a false positive chunk with tag *Proc*.

Secondly, the tool only scans for `_EPROCESS` and may miss processes that were injected to another process as a thread. To resolve this, the tool could scan for `_ETHREAD` and find the parent process `_EPROCESS` using `IoThreadToProcess`. Then, from the given `_EPROCESS`, loop through the `ThreadListHead` to find whether the thread is recorded in the process or not.

Thirdly, the hidden process activities are not monitored to send with the binary and the process memory dump. Process activities are significant to learn about file behaviour. It would be better if the result is attached with a log of process activities including file I/O and networking request/response.

Lastly, the algorithm using is not quick scanning. For a larger memory, it could result in long scanning time. Pool tag quick scanning would be a considerable improvement, but we will focus on implementing the standard algorithm and try to improve with pool tag quick scanning.

5.2. Conclusion

In this proposal stage, we have researched thoroughly about memory forensics in Windows and have come up with a solution to perform live memory forensics for finding hidden processes running. Besides, our group has begun to start developing the prototype of the tool.

Within the next stage of the thesis, we will continue working on the implementation of the tool, perform testing and resolve the challenges mentioned above. We will also try to target lower Windows versions to test the usability of the tool.

Bibliography

[1] Amer Aljaedi et al. (2011), "Comparative Analysis of Volatile Memory Forensics: Live Response vs. Memory Imaging", pp. 1253–1258, DOI: 10.1109/PASSAT/SocialCom.2011.68.

[2] Itzik Kotler Amit Klein, *Windows Process Injection in 2019*, 2019, URL: https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf.

[3] Pieter Arntz, *Explained: YARA rules*, 2017, URL: https://blog.malwarebytes.com/security-world/technology/2017/09/explained-yara-rules/.

[4] Brian Carrier (2002), "Defining Digital Forensic Examination and Analysis Tools", *International Journal of Digital Evidence*, 1, p. 2003.

[5] Geoff Chappell, *Kernel-Mode Windows*, 2019, URL: https://www.geoffchappell.com/studies/windows/km/index.htm.

[6] Michael Cohen (2015), "Characterization of the windows kernel version variability for accurate memory analysis", *Digital Investigation*, 12, DOI: 10.1016/j.diin.2015.01.009.

[7] Sebastian Eresheim, Robert Luh, and Sebastian Schrittwieser (2017), "The Evolution of Process Hiding Techniques in Malware - Current Threats and Possible Countermeasures", *Journal of Information Processing*, 25, pp. 866–874, DOI: 10.2197/ipsjjip.25.866.

[8] Volatility Foundation, *Volatility*, 2018, URL: https://www.volatilityfoundation.org/.

[9] Google, *Rekall*, 2019, URL: http://www.rekall-forensic.com/.

[10] J. Halderman et al. (2008), "Lest We Remember: Cold Boot Attacks on Encryption Keys", pp. 45–60.

[11] Jamie Levy, *memtriage*, 2019, URL: https://github.com/gleeda/memtriage.

[12] Michael Hale Ligh et al. (2014), *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*, John Wiley & Sons.

[13] Geoff McDonald, *Process-Dump*.

[14] *Memory Limits for Windows and Windows Server Releases*, 2018, URL: https://docs.microsoft.com/en-us/windows/win32/memory/memory-limits-for-windows-releases.

[15] Joachim Metz, *Minidump (MDMP) format*, 2017, URL: https://github.com/libyal/libmdmp/blob/master/documentation/Minidump(MDMP)format.asciidoc.

[16] *microsoft-pdb*, 2016, URL: https://github.com/microsoft/microsoft-pdb.

[17] *More Hidden App Malware Found on Google Play with over 2.1 Million Downloads*, 2019, URL: https://www.symantec.com/blogs/threat-intelligence/hidden-adware-google-play.

[18] *Nirsoft Windows kernel struct*, 2019, URL: http://www.nirsoft.net/.

[19] *Operating system market share*, 2019, URL: https://gs.statcounter.com/os-market-share/desktop/worldwide.

[20] *Operating systems most affected by malware as of 1st quarter 2019*, 2019, URL: https://www.statista.com/statistics/680943/malware-os-distribution/.

[21] Gary Palmer, *A Road Map for Digital Forensic Research*, 2001, URL: https://www.dfrws.org/sites/default/files/session-files/a_road_map_for_digital_forensic_research.pdf.

[22] Shuaibur Rahman and M. Khan (2015), "Review of Live Forensic Analysis Techniques", *International Journal of Hybrid Information Technology*, 8, pp. 379–388, DOI: 10.14257/ijhit.2015.8.2.35.

[23] Rajkumar Rangaraj, *pdb-downloader*, 2016, URL: https://blogs.msdn.microsoft.com/webtopics/2016/03/07/pdb-downloader/.

[24] *ReKall on windows 8 encoded KDBG*, 2014, URL: http://web.rekall-innovations.com/posts/2014-02-21-do-we-need-kdbg.html.

[25] Mark Russinovich, *Sysinternals Suite*, 2019, URL: https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite.

[26] Mark E Russinovich, David A Solomon, and Alex Ionescu (2012), *Windows internals*, Pearson Education.

[27] Andreas Schuster, *DMP File Structure*, 2006, URL: https://computer.forensikblog.de/en/2006/03/dmp-file-structure.html.

[28] Andreas Schuster (2006), "Pool Allocations as an Information Source in Windows Memory Forensics", *IMF*, pp. 104–115.

[29] Svitlana Storchak and Sergey Podobry, *Vergilius project*, 2019, URL: https://www.vergiliusproject.com/.

[30] Joe T Sylve, Vico Marziale, and Golden G Richard III (2016), "Pool tag quick scanning for windows memory analysis", *Digital Investigation*, 16, S25–S32.

[31] ReactOS Team, *ReactOS source code*, 2019, URL: https://reactos.org/.

[32] *Titanium: the Platinum group strikes again*, 2019, URL: https://securelist.com/titanium-the-platinum-group-strikes-again/94961/.

[33] *Volatility on windows 8 encoded KDBG*, 2016, URL: https://volatility-labs.blogspot.com/2014/01/the-secret-to-64-bit-windows-8-and-2012.html.

[34] Aaron Walters and Nick Petroni (2007), "Volatools: Integrating volatile memory forensics into the digital investigation process", *Digital Investigation*, URL: https://www.blackhat.com/presentations/bh-dc-07/Walters/Paper/bh-dc-07-Walters-WP.pdf.