UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program in Digital Systems Security

Master’s Thesis

Process Injection Techniques and Detection using the Volatility Framework

Sotiria Balaoura
(sbalaoura@sch.gr)
MTE1623

Supervisor: Dr. Christoforos Ntantogian, University of Piraeus

Athens, November 2018

Contents

List of Figures and Tables

List of Abbreviations

API	Application Programming Interface
DLL	Dynamic Link Library
IAT	Import Address Table
OS	Operating System
PE	Portable Executable
PEB	Process Environment Block
RVA	Relative virtual Address
SID	Security Identifier
VAD	Virtual Address Descriptor

Abstract

Malware usually incorporate mechanisms to avoid their detection. Process Injection is a technique that causes malicious code execution by injecting the code into a remote running process and forcing the process to execute it, in such a way that is concealed from the user. The program that performs the injection is called injector.

The purpose of this thesis is to propose methodologies to detect malware in memory. Regarding the malware type, it focuses on two different process injection techniques: Hollow process and Classic DLL (Dynamic Link Library) or otherwise called, Remote DLL. Various injectors are used. The malwares are executed on Windows 10 VMware virtual machines and their memory is acquired. Dynamic malware analysis is performed using the Volatility Framework.

The Hollow process injection technique is presented in detail and applied producing various testing memory images. A complete methodology of detection using the Volatility Framework is proposed that reveals and detects the anomalies that hollow process injection causes to the memory. This methodology has incorporated and organized in distinct steps most of the current literature, relevant articles on the web and research on the subject. The described steps are performed on the test images and the results are confirmed.

The Remote DLL injection is analyzed and injections are performed in various systems resulting various test memory images. A completely new methodology of detection is proposed, verified, implemented and tested. The whole idea is implemented in a python script of approximately 200 lines of code that has to be executed inside Volatility's volshell plugin environment. The results of the script executed on 12 distinct memory images, presented in the relative table, indicate that the script works satisfactory.

1 Introduction

1.1 Malware Analysis

The European Network and Information Security Agency (ENISA), in its annual Threat Landscape Report of 2017, classifies **Malware as the most frequently encountered cyberthreat** among the Top 15 Cyber Threats in 2017, exactly in the same position as it was in 2016 too. In 2017 some Anti-Virus (AV) vendors detected more than 4 million samples of threats per day [1][2]. This statistic demonstrates how critical the malware analysis is for anyone who responds to computer security incidents.

Malware analysis is the art of dissecting malware to understand how it works, how to identify it, as well as how to defeat or eliminate it. In [3] and [4], the malware analysis approaches are categorized as **static** or **dynamic**. The purpose of static analysis is to detect the malware before its execution. It consists of examining the executable file without viewing the actual instructions or reverse-engineering the malware's internals. Dynamic analysis techniques attempt to detect malicious behavior during or after the malware execution. It involves executing the malicious code, in an isolated environment, and observing its behavior as well as the anomalies or inconsistencies it causes. Off course, prerequisite for anomalies' recognition is the knowledge of what is normal.

The next issue is where to look for these anomalies, which can also be considered as evidences (of the way the malware works). The possible locations can be categorized in the same way as the digital forensics: in volatile and non-volatile computer storage [5]. Non-volatile electronic evidence is found on hard disks, USB flash drives, and removable media. They contain files, system and network logs, corrupted files and maybe even the malicious code which could be analyzed. The disk drive's file system can lead to the recovery of deleted files, which may contain further evidence. The analysis of data captured from hard disk is also called **disk forensics**. Volatile media, that is the main memory or RAM (Random Access Memory) contain information about each running process and thread, open files, deleted files, Windows registry keys and event logs, established network connections, recently executed commands, URLs, IP addresses, executing code, including malware. It is important to notice that while a malicious program is being executed, it cannot be erased from memory, unlike the hard disk. In other words, every function performed by an operating system or application, results in specific modifications to the computer's memory, which can often persist a long time after the action, essentially preserving them [6]. The analysis of data captured from memory is also called **memory forensics**.

For a malware analyst it is ideal to be able to perform both disk and memory forensics, but it is obvious that memory analysis has advantages relative to disk analysis. It seems that memory is the best place to identify malicious software activity. An analyst can study the system configuration and identify inconsistencies in it, bypass packers, rootkits and other hiding tools. Recent activity can be tracked and analyzed. Evidences that cannot be found anywhere else can be collected, such as memory-only malware [7]. Memory analysis can be done on the live system, but it can also be done on a dump of the volatile memory. A memory dump (also known as a core dump or system dump) is a snapshot capture of computer memory data from a specific instant. For a no longer live system, crash dumps and hibernation files can provide information about it. A hibernation file (hiberfil.sys) contains a compressed copy of memory that the system dumps to disk during the hibernation process.

There are commercial **memory analysis tools**, such as WindowsSCOPE Cyber Forensics [13], F-Response [14], Windows Memory Forensic Toolkit (WMFT) as well as open source tools such as the Volatility Framework. For a relatively complete list of open source memory analysis tools, see [15].

1.2 The Volatility Framework

The Volatility Framework is a completely open collection of tools, implemented in Python under the GNU General Public License, for the **extraction of digital artifacts from volatile memory (RAM) samples** or memory images as they are otherwise called. The extraction techniques are performed completely independent of the system being investigated but offer unprecedented visibility into the runtime state of the system. The framework is intended to introduce people to the techniques and complexities associated with extracting digital artifacts from volatile memory samples and provide a platform for further work into this exciting area of research. The Volatility Framework is maintained and promoted by The Volatility Foundation which is an independent non-profit organization [17].

Volatility supports memory dumps from all major 32- and 64-bit Windows versions. Whether the memory dump is in raw format, a Microsoft crash dump, hibernation file, or virtual machine snapshot, Volatility is able to work with it. It also supports Linux memory dumps in raw or LiME format and include 35+ plugins for analyzing Linux kernels. It supports 38 versions of Mac OSX memory dumps. Android phones with ARM processors are also supported [16].

The amount of Volatility's tools, the fact that it continuously maintained, so it supports Windows 10 Virtual Machines, the easiness for plugins' creations and the rich documentation, practically make it the most prudent choice as a memory analysis tool [18][19][20][21].

1.3 Virtualization

For malware analysis, virtualization can be used to create the **isolated environment** mentioned above. Virtualization refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage or network. In this case, operating system-level virtualization is used, meaning running multiple operating systems on a single piece of hardware. Virtualization technology involves separating the physical hardware and software by emulating hardware using software. When a different operating system is operating on top of the primary one, by means of virtualization, it is referred to as a virtual machine [8].

Currently the popular virtual products in the market are VMware Workstation player or pro [9], VirtualBox [10], Hyper-V [11] and Parallels [12]. These products can give us a snap shot of memory and page file which is an identical to original data of the virtual machine. For example, when we suspend a VMware virtual machine, the whole activities on it are stopped and the contents of the physical memory are contained in a .vmem file which is a raw memory image.

1.4 What is process injection

Researchers classify the many types of malware in several different ways: The delivery method or attack methodology, the specific type of vulnerability that the malware exploits, the objective of the malware, the persistence mechanisms, etc. An interesting classification the one referring to the methods that malware authors use to **avoid detection**, called covert launching techniques and mostly refer to the malware loader. Loader (also known as a launcher) is a program that is responsible for launching the malware itself in such a way that is concealed from the user [3].

Process Injection is a loader technique that causes malicious code execution by injecting the code into a remote running process and forcing the process to execute it, resulting in a different behavior than the expected one. It is a widespread defense evasion technique employed often within malware. The program that performs the injection is called **injector**. Direct injection refers to allocating and inserting code into the memory space of a remote process. DLL (Dynamic Link Library) injection is a form of process injection where the injected item is a DLL that is loaded within the context of the remote process. There are many process injection techniques, but the most common are:

- Hollow process injection or process replacement
- Remote DLL injection or otherwise called Classic DLL injection
- Portable Executable injection (PE injection)
- Thread execution hijacking
- Hook injection
- APC (Asynchronous Procedure Calls) injection and Atom Bombing

[22][6].

1.5 Prerequisites

In order to follow the concept of this thesis, some basic knowledge of the Windows operating system is required on these topics:

- Window memory management: Virtual memory, physical memory, paging, shared memory, Kernel and User Mode [23][24][26]
- Processes and the corresponding _EPROCESS data structure, threads, process memory layout [25], Virtual Address Descriptors (VADs) [6][26]
- Portable Executable (PE) File format [27][28][29][30][31]
- Handles [32]

It is not among the goals of this work to present the above issues, so the corresponding useful bibliography is cited, for those who believe that they need to refresh their knowledge.

1.6 Problem definition and thesis' scope

The purpose of this thesis is to propose methodologies to detect malware in memory and regarding the malware type, focuses on two different process injection techniques: **Hollow process injection** and **Classic DLL Injection**. To perform the injection, various **injectors** are chosen and used as described in the following sections. The malwares are executed on **Windows 10** VMware **virtual machines** and their memory are acquired. **Dynamic malware analysis** is performed using the **Volatility Framework**. Windows 10 is chosen as it is the latest windows versions and because one of the chosen injectors is tested only on this operating system.

For the detection of Hollow process injection, a **complete methodology**, based on current literature is presented and tested. The knowledge gained through this process, led to the creation of a completely **new methodology** for DLL Injection detection which is also presented in detail and tested.

1.7 Organization of this document

In the following chapter 2 is described how the hollow process injection technique works. After that, a complete methodology of detection is presented using the Volatility Framework, based on current literature. The methodology is explained in detail and applied producing various test memory images. The configuration of the testing environment, in which injections are performed, is described in detail.

Chapter 3 analyzes remote DLL injection and a new approach of detection is proposed, tested and evaluated. This approach is implemented by a python script that can be executed using the Volatility framework. The testing environment is described and the results of the conducted test are presented and evaluated.

2 Hollow process or process replacement

This chapter presents the code injection technique called hollow process or process replacement: the main concept of it as well as some of its main variations. A methodology is proposed that relevels and detects the anomalies that hollow process injection causes to the memory. This injection technique is performed in a testing environment, which is also described in detail, using two different malware-injectors and producing memory images on which the methodology of detection is applied. Alternative memory images available are also used. Each step of the methodology is explained and its results on the memory image are analyzed.

2.1 Introduction and hollow process description

Hollow Process Injection (or Process Hollowing, or Process Replacement, or Dynamic forking as it is otherwise called) is used when the malware needs to be disguised as a legitimate process, without the risk of crashing.

Hollow Process Injection is a code injection technique in which the hollowing process starts a new instance of a legitimate process in suspended state. The executable section of the legitimate process in the memory is swapped out (hollowed) and is replaced with malicious code, mostly malicious executable. After that, the (no longer) legitimate process is resumed and it executes the malicious code for the remainder of its process lifetime within its legitimate context of the process. The PEB (Process Environment Block) still points to the legitimate path and the processes' data structures remain the same, the malware has the same privileges as the process that is replacing. So, the user cannot distinguish the hollowed process from a legitimate one. For example, if the svchost.exe is replaced, the user would see a process named svchost.exe running from C:\Windows\System32 with normal characteristics.

2.2 Steps to hollow a process

Three components are involved in process hollowing: Let's assume that the process that performs the hollowing e.g. the **injector is called hollow.exe**, the legitimate process that it creates in order to get hollowed is **legitimate.exe (host process or remote process)** and the **malicious code (payload)** to be injected is malicious.exe.

Although there are various techniques that can be used for hollow process injection, in the most common variant the **hollowing process typically follows the following steps:**

1. Creates a new instance the legitimate process (for example C:\Windows\explorer.exe, C:\windows\system32\lsass.exe, C:\Windows\system32\svchost.exe), but with its first thread suspended.
   Result:
   The executable section legitimate.exe is loaded in the memory, but is not yet executed and its memory space can be modified. Its PEB (Process Environment Block) members such as ImagePathName and ImageBaseAddress, have the corresponding values.

How this can be done:
Create the legitimate process using CreateProcess with CREATE_SUSPENDED option and keep the provided handle for the subsequent function calls that modify its memory space.

2. Acquires the malicious code (mostly executable) to inject. This code can be originated from anywhere: from a file on the disk/over the network or the resource section of the hollowing process. The malicious PE (Portable Executable) file is parsed. Attributes like PE header, PE header size, sections and corresponding sizes of the malicious executable are extracted.

3. Reads the legitimates' processes entry point and image base address (which holds the memory address where the executable is loaded), by reading the PEB, i.e. PEB.ImageBaseAddress.
   After that, it frees or unmaps the containing memory section.
   Result
   The hollow is created. The legitimate process is just an empty container (the DLLs, heaps, stacks and open handles are still intact, but no process executable exists).
   How this can be done:
   Get the base address of the legitimate PEB using NtQueryProcessInformation or GetThreadContext function and then read PEB.ImageBaseAddress in process memory using ReadProcessMemory. After that, the NtUnmapViewOfSection function is utilized to unmap the section. The above functions are kernel functions, so hollow.exe usually resolves them at runtime using GetProcAddress.

4. Allocates a new memory region within the virtual address space of the legitimate process. The size of the memory region allocated is determined by the size of the malicious code. The starting address of this region depends on the technique variation used. If needed, hollow.exe updates the malicious PE header and calculates the difference between the two images' (legitimate and malicious) base addresses (delta) which is used in rebasing the malicious image. For further details, see Process Hollowing Variations.
   Result:
   The memory needed for the replacement is allocated and contains zeros.
   How this can be done:
   Use VirtualAllocEx with PAGE_EXECUTE_READWRITE permissions (for simplicity reasons) and allocation type MEM_COMMIT | MEM_RESERVE

5. Copies the malicious PE header as well as each PE section (.text, .rdata, .data, etc) into the hollow created inside the legitimates' process memory.
   Result:
   The injection is completed.
   How this can be done:

Use WriteProcessMemory to write the malicious image into the allocated space inside the legitimate process.

6. Updates injected processes' structures so that the malicious code is going to be executed.
   Result:
   The injected processes' PEB ImageBaseAddress points as the malicious image. The thread context of the suspended thread is set so that its first instruction (Entry Point) points at the injected executable.
   How this can be done:
   Calculate the new entry point. Call GetThreadContext function, update eax register and then call SetThreadContext

7. Resumes the suspended thread. At this point, the malicious code starts executing within the container created for the legitimate.exe.
   Result
   The injected (and no longer) legitimate process is executing looking legitimate on the outside but being malicious on the inside.
   How this can be done:
   Hollow.exe simply resumes the suspended process using ResumeThread function.

[6][22][33][34][35][36][37][43]

The advantage of hollow process injection is that the user cannot distinguish the injected process from the legit one using conventional tools. The PEB is unchanged so it still preserves valid looking fields in important structures, like ImagePathName, ImageBaseAddress, etc. It looks normal and not suspicious. The only think that has changed is the actual code that is executing.

In the following figure is shown, at a very abstract level, how the legitimates' Process Address Space changes during the steps described above.

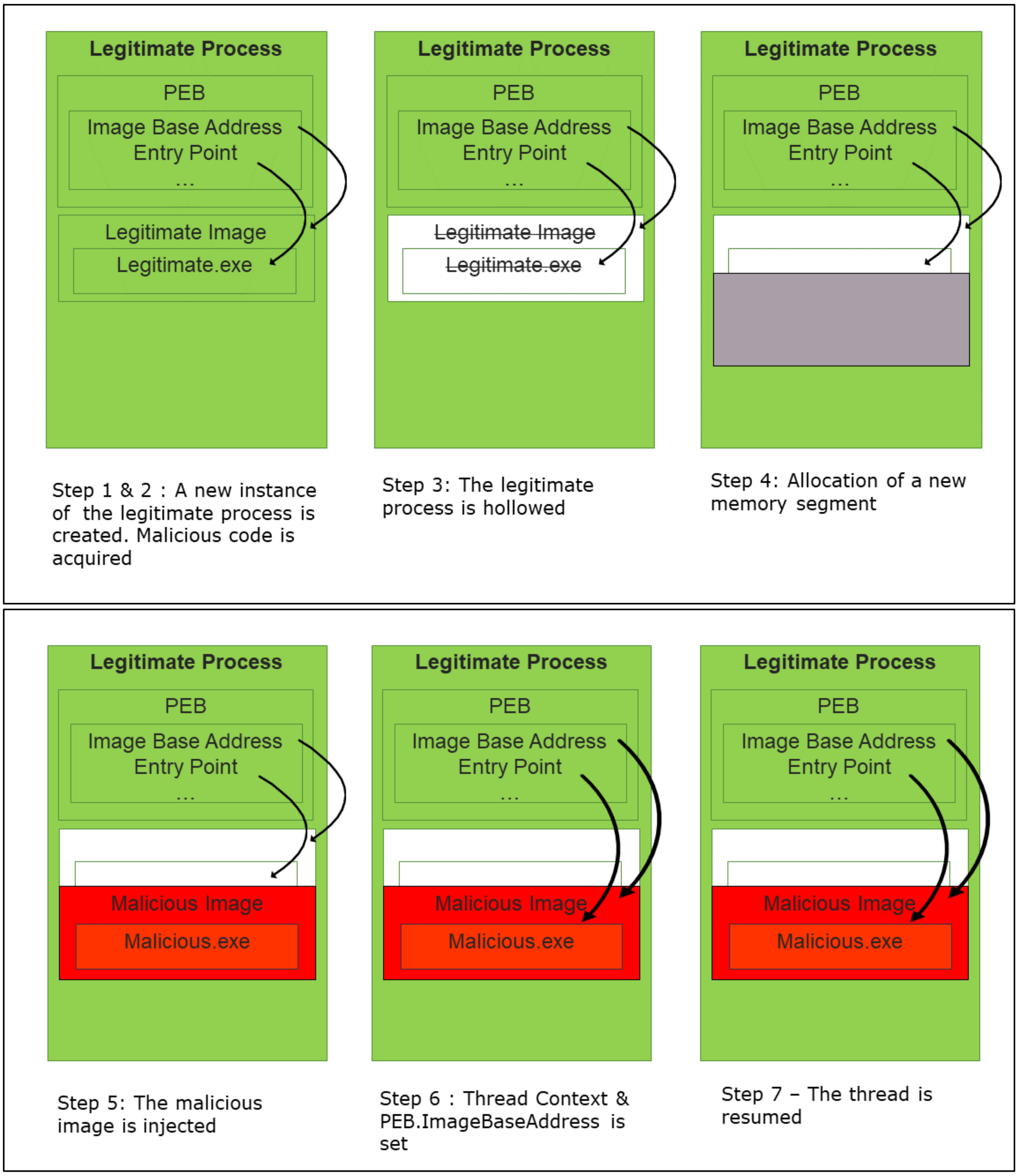

Legitimate Process
PEB
Image Base Address
Entry Point
...
Legitimate Image
Legitimate.exe
Step 1 & 2 : A new instance of the legitimate process is created. Malicious code is acquired
~~Legitimate Image~~
~~Legitimate.exe~~
Step 3: The legitimate process is hollowed
Step 4: Allocation of a new memory segment
Malicious Image
Malicious.exe
Step 5: The malicious image is injected
Step 6 : Thread Context & PEB.ImageBaseAddress is set
Step 7 – The thread is resumed

Figure 1 - Process Memory Space during hollowing

Malwares that use Process Hollowing exclusively or as a malware loader, are Stuxnet and Careto [6], DarkComet [38], Dridex [39], Skeeyah [35]. Various malware examples are also listed in [40].

2.3 Process Hollowing Variations

As mentioned before, there are some variations in the above steps and the most common ones regard the following:

Address of the hollow and address of the allocated memory

- Memory unmapping and then memory allocation is done at the same address where the legitimate executable was previously loaded (PEB.ImageBaseAddress) and the malicious' PE header is updated so that its ImageBase equals to PEB.ImageBaseAddress of the legitimate image [41][6][33]. However, before setting it, the difference between the two images' (legitimate and malicious) base addresses must be calculated (delta) for use in rebasing the malicious image, if needed.

- The memory unmapping is done at PEB.ImageBaseAddress. The allocation is done at the address of the malicious' PE ImageBase (found in the optional header), so no changes are to be done in malicious PE headers. However, after writing PE header and sections, the PEB.ImageBaseAddress of the legitimate process must be updated to point to the address that came up from the memory allocation. This way, PEB points to the injected executable [35].
  This is convenient, because is more than likely that absolute addresses are involved within the code which is entirely dependent on its location in memory. Since many executables share common base addresses (usually `0x400000`), it is not uncommon that the hollowed process's own executable image exists at the same address [37][35].

- The memory allocation is not done at a specific address. This is a more general case and the following steps must be performed:
  a. Calculate the difference between the legitimates' PEB.ImageBaseAddress and PE.OptionalHeader.ImageBase. This delta value is used for rebasing.
  b. Update PE header of the malicious image so that its ImageBase equals to PEB.ImageBaseAddress of the legitimate image
  c. If the delta calculated in the prior step is not zero, rebase the malicious image. This involves recalculating every absolute address and modifying the code to use the new values: delta + preferred address.
  d. Calculate the new entry point as the sum of PEB.ImageBasesAddress (of the legitimate process) + Malicious PE Header-> OptionalHeader.AddressOfEntryPoint
  [42][34]

Memory protection for the region of pages at memory allocation step

The memory protection for the region of pages allocated could be for the sake of simplicity PAGE_EXECUTE_READWRITE, but this could be improved upon by changing each PE section with the appropriate permissions based on the characteristics specified in the section header, for example using VirtualProtectEx function. In this case, the hollowing should be harder to be detected [43].

No memory umapping

In this case, there is not any hollowing out of the legitimate executable because the Unmaping step is not performed. A new memory allocation is done and the malicious code is copied there. Then PEB.ImageBase and Entrypoint is updated as already described [35].

Modifying the legitimate code

In this case, after creating the legitimate process in suspended mode, the hollowing process creates a section in its own address space, copies the malicious code in it and finally maps a view of it in the legitimate process with PAGE_EXECUTE_READWRITE permissions. So, it injects the malicious code. After that is creates a second section, copies there the legitimate.exe and changes it so that it jumps to the address of the previously mapped view. This requires that 7 bytes of address of entry point are modified. Then it unmaps the legitimate.exe from the legitimate process and maps the section of the changed code at the same address. The "legitimate" process is resumed, so it executes the malicious code [35].

2.4 Testing Environment

The code injection is performed only in memory, so it is best detected using memory forensics. As already mentioned, the open source Volatility Framework is used [17].

There is a complete reference of the commands and plugins used in [18]. In https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples, there is a list of publicly available memory samples for testing purposes. Also, there are samples in https://www.memoryanalysis.net/amf, hyperlink "All memory images".

For the investigation, KALI Linux 2017.1 was used (downloaded file name kali-linux-2017.1-amd64.iso from https://www.kali.org/downloads/) which includes Volatility's Foundation Volatility Framework 2.6. KALI and was opened with Oracle VirtualBox version 5.2.12 r122591 (Qt5.6.2) [10].

The proposed methodology is applied in the memory images:

- Stuxnet.vmem which is a memory sample that came from a virtual machine infected with Stuxnet and is downloaded from https://volatility-labs.blogspot.gr/2016/08/automating-detection-of-known-malware.html.
- Three different test memory mages that are created in the context of this thesis.

These test images came from a windows 10 virtual machine (Enterprise Evaluation Build 14393.rs1_release.180209-1727, downloaded from https://az792536.vo.msecnd.net/vms/VMBuild_20160802/VMWare/MSEdge/MSEdge.Win10_RS1.VMWare.zip, which was opened with VMware® Workstation 12 Pro software, version 12.5.2 build-4638234 [9].

To perform the process replacement, the source code of two different projects was used: InjectProc [41] and Process-Hollowing [34]. The source code was compiled using Microsoft Visual Studio Community 2017, version 15.5.6. In both cases, the "malicious" executable just pops up a window with title "pwnd" displaying "Injected" or displaying "Hello World" respectively. After the injection, the memory image was retrieved by suspending the virtual machine and copying the corresponding .vmss and .vmem files.

The memory images that are hollowed using InjectProc project are MSEdge - Win10_preview-eaaa27c2-onrepl.vmem and MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem. In both images, the hollowed process is notepad.exe. Using Process-Hollowing project, which hollows svchost.exe, the memory image created is MSEdge - Win10_OnHollow-m0.vmem. For details on how the injection is performed, see Hollow process injection and detection examples section. In the specific section is also demonstrated how needed information is retrieved in order to verify a) that the injection is performed and b) that the results of the proposed methodology are correct.

In the following sections, the methodology is presented using MSEdge - Win10_preview-eaaa27c2-onrepl.vmem. The Volatility's profile parameter used is `--profile="Win10x64_14393"` which corresponds to the windows version of the virtual machine.

2.5 Methodology of Detection

Before analyzing the giveaways of process hollowing, some issues regarding Windows' architecture, must be reminded:

The executive process structure EPROCESS exists in system address space (kernel memory) except the PEB (Process Environment Block) structure which exists in process address space (process memory) and thus can be modified, as described in process hollowing steps.

The Virtual Address Descriptors (VADs) are data structures that the memory manager uses to keep track of the virtual addresses the process is using and is an excellent forensic resource because when a process allocates memory using VirtualAlloc, the memory manager creates an entry in the VAD tree. A process' VAD tree describes the layout of its memory segments at a slightly higher level than the page tables and is maintained by the operating system, in kernel memory. The VADs contain information such as the starting and ending addresses of the allocated memory block, mapped file name, the initial protection (read, write, execute) and several other characteristics that are objects of interest for the detection method [6][26].

In the following sections, the methodology is presented in two ways: first, presenting the Volatiliy command using a hypothetically memory image called mem.bin and secondly executing the corresponding command using MSEdge - Win10_preview-eaaa27c2-onrepl.vmem, which was introduced in the previous paragraph. After that, the results are analyzed.

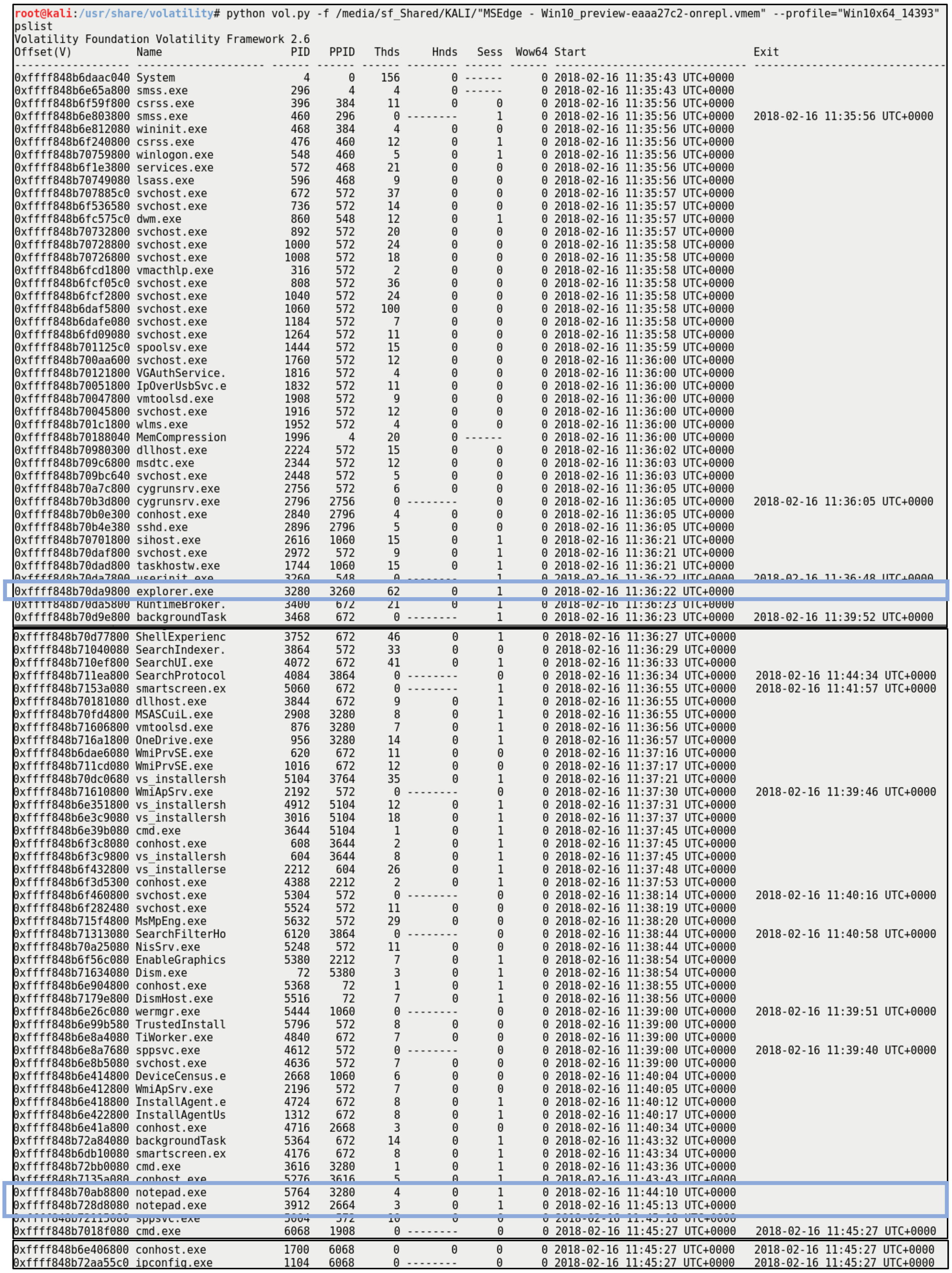
The steps to detect Process Hollowing are:

2.5.1 Process Listing

The purpose of this step is to **determine suspicious processes**, meaning processes which are not started from the corresponding parent process or parent process that is terminated, although it should be present. The last case is not a sufficient proof, rather than a simple indication. But the first case is a strong indication that something is wrong. For example, there should be only one instance of lsass.exe running from the system32 directory and its parent should be winlogon.exe on pre-Vista machines, or wininit.exe on Vista and later systems. Stuxnet for example, creates two fake copies of lsass.exe, and their parent is not winlogon.exe. Similarly, services.exe should be the parent for any svchost.exe instances. As already mentioned, another point to be taken into account is whether the number of instances of a specific process is the right one [6][35][43]. To list the processes of the system, volatility's pslist plugin is used.

```
python vol.py – f mem.bin pslist
```

In any step, it is a good practice to compare the data regarding the suspicious process with the data regarding corresponding legitimate.



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
------------------ -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ------------------------------
0xffff848b6daac040 System                    4      0    156        0 ------      0 2018-02-16 11:35:43 UTC+0000
0xffff848b6e65a800 smss.exe                296      4      4        0 ------      0 2018-02-16 11:35:43 UTC+0000
0xffff848b6f59f800 csrss.exe               396    384     11        0      0      0 2018-02-16 11:35:56 UTC+0000
0xffff848b6e803800 smss.exe                460    296      0 --------      1      0 2018-02-16 11:35:56 UTC+0000   2018-02-16 11:35:56 UTC+0000
0xffff848b6e812080 wininit.exe             468    384      4        0      0      0 2018-02-16 11:35:56 UTC+0000
0xffff848b6f240800 csrss.exe               476    460     12        0      1      0 2018-02-16 11:35:56 UTC+0000
0xffff848b70759800 winlogon.exe            548    460      5        0      1      0 2018-02-16 11:35:56 UTC+0000
0xffff848b6f1e3800 services.exe            572    468     21        0      0      0 2018-02-16 11:35:56 UTC+0000
0xffff848b70749080 lsass.exe               596    468      9        0      0      0 2018-02-16 11:35:56 UTC+0000
0xffff848b707885c0 svchost.exe             672    572     37        0      0      0 2018-02-16 11:35:57 UTC+0000
0xffff848b6f536580 svchost.exe             736    572     14        0      0      0 2018-02-16 11:35:57 UTC+0000
0xffff848b6fc575c0 dwm.exe                 860    548     12        0      1      0 2018-02-16 11:35:57 UTC+0000
0xffff848b70732800 svchost.exe             892    572     20        0      0      0 2018-02-16 11:35:57 UTC+0000
0xffff848b70728800 svchost.exe            1000    572     24        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b70726800 svchost.exe            1008    572     18        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6fcd1800 vmacthlp.exe            316    572      2        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6fcf05c0 svchost.exe             808    572     36        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6fcf2800 svchost.exe            1040    572     24        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6daf5800 svchost.exe            1060    572    100        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6dafe080 svchost.exe            1184    572      7        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b6fd09080 svchost.exe            1264    572     11        0      0      0 2018-02-16 11:35:58 UTC+0000
0xffff848b701125c0 spoolsv.exe            1444    572     15        0      0      0 2018-02-16 11:35:59 UTC+0000
0xffff848b700aa600 svchost.exe            1760    572     12        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70121800 VGAuthService.         1816    572      4        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70051800 IpOverUsbSvc.e         1832    572     11        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70047800 vmtoolsd.exe           1908    572      9        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70045800 svchost.exe            1916    572     12        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b701c1800 wlms.exe               1952    572      4        0      0      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70188040 MemCompression         1996      4     20        0 ------      0 2018-02-16 11:36:00 UTC+0000
0xffff848b70980300 dllhost.exe            2224    572     15        0      0      0 2018-02-16 11:36:02 UTC+0000
0xffff848b709c6800 msdtc.exe              2344    572     12        0      0      0 2018-02-16 11:36:03 UTC+0000
0xffff848b709bc640 svchost.exe            2448    572      5        0      0      0 2018-02-16 11:36:03 UTC+0000
0xffff848b70a7c800 cygrunsrv.exe          2756    572      6        0      0      0 2018-02-16 11:36:05 UTC+0000
0xffff848b70b3d800 cygrunsrv.exe          2796   2756      0 --------      0      0 2018-02-16 11:36:05 UTC+0000   2018-02-16 11:36:05 UTC+0000
0xffff848b70b0e300 conhost.exe            2840   2796      4        0      0      0 2018-02-16 11:36:05 UTC+0000
0xffff848b70b4e380 sshd.exe               2896   2796      5        0      0      0 2018-02-16 11:36:05 UTC+0000
0xffff848b70701800 sihost.exe             2616   1060     15        0      1      0 2018-02-16 11:36:21 UTC+0000
0xffff848b70daf800 svchost.exe            2972    572      9        0      1      0 2018-02-16 11:36:21 UTC+0000
0xffff848b70dad800 taskhostw.exe          1744   1060     15        0      1      0 2018-02-16 11:36:21 UTC+0000
0xffff848b70da7800 userinit.exe           3260    548      0 --------      1      0 2018-02-16 11:36:22 UTC+0000   2018-02-16 11:36:48 UTC+0000
0xffff848b70da9800 explorer.exe           3280   3260     62        0      1      0 2018-02-16 11:36:22 UTC+0000
0xffff848b70da5800 RuntimeBroker.         3400    672     21        0      1      0 2018-02-16 11:36:23 UTC+0000
0xffff848b70d9e800 backgroundTask         3468    672      0 --------      1      0 2018-02-16 11:36:23 UTC+0000   2018-02-16 11:39:52 UTC+0000

0xffff848b70d77800 ShellExperienc         3752    672     46        0      1      0 2018-02-16 11:36:27 UTC+0000
0xffff848b71040080 SearchIndexer.         3864    572     33        0      0      0 2018-02-16 11:36:29 UTC+0000
0xffff848b710ef800 SearchUI.exe           4072    672     41        0      1      0 2018-02-16 11:36:33 UTC+0000
0xffff848b711ea800 SearchProtocol         4084   3864      0 --------      0      0 2018-02-16 11:36:34 UTC+0000   2018-02-16 11:44:34 UTC+0000
0xffff848b7153a080 smartscreen.ex         5060    672      0 --------      1      0 2018-02-16 11:36:55 UTC+0000   2018-02-16 11:41:57 UTC+0000
0xffff848b70181080 dllhost.exe            3844    672      9        0      1      0 2018-02-16 11:36:55 UTC+0000
0xffff848b70fd4800 MSASCuiL.exe           2908   3280      8        0      1      0 2018-02-16 11:36:55 UTC+0000
0xffff848b71606800 vmtoolsd.exe            876   3280      7        0      1      0 2018-02-16 11:36:56 UTC+0000
0xffff848b716a1800 OneDrive.exe            956   3280     14        0      1      0 2018-02-16 11:36:57 UTC+0000
0xffff848b6dae6080 WmiPrvSE.exe            620    672     11        0      0      0 2018-02-16 11:37:16 UTC+0000
0xffff848b711cd080 WmiPrvSE.exe           1016    672     12        0      0      0 2018-02-16 11:37:17 UTC+0000
0xffff848b70dc0680 vs_installersh         5104   3764     35        0      1      0 2018-02-16 11:37:21 UTC+0000
0xffff848b71610800 WmiApSrv.exe           2192    572      0 --------      0      0 2018-02-16 11:37:30 UTC+0000   2018-02-16 11:39:46 UTC+0000
0xffff848b6e351800 vs_installersh         4912   5104     12        0      1      0 2018-02-16 11:37:31 UTC+0000
0xffff848b6e3c9080 vs_installersh         3016   5104     18        0      1      0 2018-02-16 11:37:37 UTC+0000
0xffff848b6e39b080 cmd.exe                3644   5104      1        0      1      0 2018-02-16 11:37:45 UTC+0000
0xffff848b6f3c8080 conhost.exe             608   3644      2        0      1      0 2018-02-16 11:37:45 UTC+0000
0xffff848b6f3c9800 vs_installersh          604   3644      8        0      1      0 2018-02-16 11:37:45 UTC+0000
0xffff848b6f432800 vs_installerse         2212    604     26        0      1      0 2018-02-16 11:37:48 UTC+0000
0xffff848b6f3d5300 conhost.exe            4388   2212      2        0      1      0 2018-02-16 11:37:53 UTC+0000
0xffff848b6f460800 svchost.exe            5304    572      0 --------      0      0 2018-02-16 11:38:14 UTC+0000   2018-02-16 11:40:16 UTC+0000
0xffff848b6f282480 svchost.exe            5524    572     11        0      0      0 2018-02-16 11:38:19 UTC+0000
0xffff848b715f4800 MsMpEng.exe            5632    572     29        0      0      0 2018-02-16 11:38:20 UTC+0000
0xffff848b71313080 SearchFilterHo         6120   3864      0 --------      0      0 2018-02-16 11:38:44 UTC+0000   2018-02-16 11:40:58 UTC+0000
0xffff848b70a25080 NisSrv.exe             5248    572     11        0      0      0 2018-02-16 11:38:44 UTC+0000
0xffff848b6f56c080 EnableGraphics         5380   2212      7        0      1      0 2018-02-16 11:38:54 UTC+0000
0xffff848b71634080 Dism.exe                 72   5380      3        0      1      0 2018-02-16 11:38:54 UTC+0000
0xffff848b6e904800 conhost.exe            5368     72      1        0      1      0 2018-02-16 11:38:55 UTC+0000
0xffff848b7179e800 DismHost.exe           5516     72      7        0      1      0 2018-02-16 11:38:56 UTC+0000
0xffff848b6e26c080 wermgr.exe             5444   1060      0 --------      0      0 2018-02-16 11:39:00 UTC+0000   2018-02-16 11:39:51 UTC+0000
0xffff848b6e99b580 TrustedInstall         5796    572      8        0      0      0 2018-02-16 11:39:00 UTC+0000
0xffff848b6e8a4080 TiWorker.exe           4840    672      7        0      0      0 2018-02-16 11:39:00 UTC+0000
0xffff848b6e8a7680 sppsvc.exe             4612    572      0 --------      0      0 2018-02-16 11:39:00 UTC+0000   2018-02-16 11:39:40 UTC+0000
0xffff848b6e8b5080 svchost.exe            4636    572      7        0      0      0 2018-02-16 11:39:00 UTC+0000
0xffff848b6e414800 DeviceCensus.e         2668   1060      6        0      0      0 2018-02-16 11:40:04 UTC+0000
0xffff848b6e412800 WmiApSrv.exe           2196    572      7        0      0      0 2018-02-16 11:40:05 UTC+0000
0xffff848b6e418800 InstallAgent.e         4724    672      8        0      1      0 2018-02-16 11:40:12 UTC+0000
0xffff848b6e422800 InstallAgentUs         1312    672      8        0      1      0 2018-02-16 11:40:17 UTC+0000
0xffff848b6e41a800 conhost.exe            4716   2668      3        0      0      0 2018-02-16 11:40:34 UTC+0000
0xffff848b72a84080 backgroundTask         5364    672     14        0      1      0 2018-02-16 11:43:32 UTC+0000
0xffff848b6db10080 smartscreen.ex         4176    672      8        0      1      0 2018-02-16 11:43:34 UTC+0000
0xffff848b72bb0080 cmd.exe                3616   3280      1        0      1      0 2018-02-16 11:43:36 UTC+0000
0xffff848b7135a080 conhost.exe            5276   3616      5        0      1      0 2018-02-16 11:43:43 UTC+0000
0xffff848b70ab8800 notepad.exe            5764   3280      4        0      1      0 2018-02-16 11:44:10 UTC+0000
0xffff848b728d8080 notepad.exe            3912   2664      3        0      1      0 2018-02-16 11:45:13 UTC+0000
0xffff848b72115080 sppsvc.exe             3004    572     10        0      0      0 2018-02-16 11:45:18 UTC+0000
0xffff848b7018f080 cmd.exe                6068   1908      0 --------      0      0 2018-02-16 11:45:27 UTC+0000   2018-02-16 11:45:27 UTC+0000

0xffff848b6e406800 conhost.exe            1700   6068      0        0      0      0 2018-02-16 11:45:27 UTC+0000   2018-02-16 11:45:27 UTC+0000
0xffff848b72aa55c0 ipconfig.exe           1104   6068      0 --------      0      0 2018-02-16 11:45:27 UTC+0000   2018-02-16 11:45:27 UTC+0000
```

Figure 2 - Process Listing in test image

It is observed that services.exe is the parent of all svchost.exe instances, as they should. Everything looks normal, but it is also observed that there are two instances of notepad.exe with pids 5764 and 3912. The parent process id (ppid) of process 5763 is 3280 that corresponds to explorer.exe, but the parent of 3912 (pid 2664) is not found in the process list.

2.5.2 Comparing kernel and process memory structures

The purpose of this step is to discover any discrepancies, indicating that a process is hollowed out, between the two primary data structures used by processes: the PEB (Process Environment Block) which exists in process memory and the VADs (Virtual Address Descriptors) which exist in kernel memory. In a noninfected system the results from the Volatility's plugins:

- **dllist** (which displays a process's loaded DLLs and retrieves the information by walking the load order list of PEB)
- **vadinfo** (which displays extended information about a process's VAD nodes and retrieves the information from the VAD structures)
- **ldrmodules** (which retrieves the information from the PEB and also correlates it with VADs)

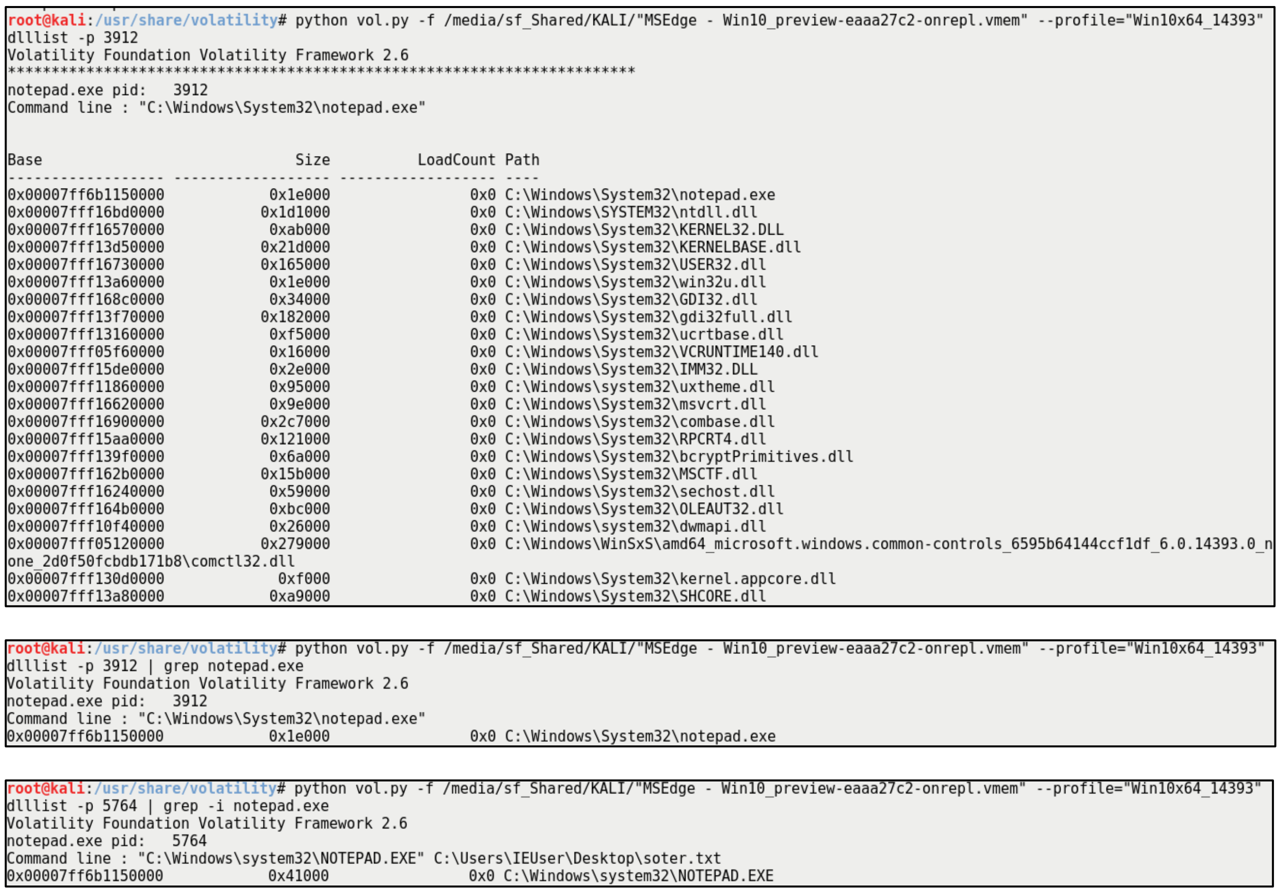
should be **compatible** [6][35][36].

Dynamic Link Library Listing

Since the data in PEB are initialized at process creation, dlllist may not provide any direct clue, but the columns **PATH** (full path) and **BASE** (Image Base) of the executables found should be used as a reference for the following commands. An interesting point is that all instances of a process should have the same executable size as explained in a next step.

Concentrating on the results from the previous step, the commands for this step are:

```
python vol.py – f mem.bin
dlllist -p <suspicious process_id> |grep -i <suspicious process_name>
```



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
dlllist -p 3912
Volatility Foundation Volatility Framework 2.6
************************************************************************
notepad.exe pid:   3912
Command line : "C:\Windows\System32\notepad.exe"


Base                             Size          LoadCount Path
------------------ ------------------ ------------------ ----
0x00007ff6b1150000            0x1e000                0x0 C:\Windows\System32\notepad.exe
0x00007fff16bd0000           0x1d1000                0x0 C:\Windows\SYSTEM32\ntdll.dll
0x00007fff16570000            0xab000                0x0 C:\Windows\System32\KERNEL32.DLL
0x00007fff13d50000           0x21d000                0x0 C:\Windows\System32\KERNELBASE.dll
0x00007fff16730000           0x165000                0x0 C:\Windows\System32\USER32.dll
0x00007fff13a60000            0x1e000                0x0 C:\Windows\System32\win32u.dll
0x00007fff168c0000            0x34000                0x0 C:\Windows\System32\GDI32.dll
0x00007fff13f70000           0x182000                0x0 C:\Windows\System32\gdi32full.dll
0x00007fff13160000            0xf5000                0x0 C:\Windows\System32\ucrtbase.dll
0x00007fff05f60000            0x16000                0x0 C:\Windows\System32\VCRUNTIME140.dll
0x00007fff15de0000            0x2e000                0x0 C:\Windows\System32\IMM32.DLL
0x00007fff11860000            0x95000                0x0 C:\Windows\system32\uxtheme.dll
0x00007fff16620000            0x9e000                0x0 C:\Windows\System32\msvcrt.dll
0x00007fff16900000           0x2c7000                0x0 C:\Windows\System32\combase.dll
0x00007fff15aa0000           0x121000                0x0 C:\Windows\System32\RPCRT4.dll
0x00007fff139f0000            0x6a000                0x0 C:\Windows\System32\bcryptPrimitives.dll
0x00007fff162b0000           0x15b000                0x0 C:\Windows\System32\MSCTF.dll
0x00007fff16240000            0x59000                0x0 C:\Windows\System32\sechost.dll
0x00007fff164b0000            0xbc000                0x0 C:\Windows\System32\OLEAUT32.dll
0x00007fff10f40000            0x26000                0x0 C:\Windows\system32\dwmapi.dll
0x00007fff05120000           0x279000                0x0 C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.14393.0_n
one_2d0f50fcbdb171b8\comctl32.dll
0x00007fff130d0000             0xf000                0x0 C:\Windows\System32\kernel.appcore.dll
0x00007fff13a80000            0xa9000                0x0 C:\Windows\System32\SHCORE.dll
```

```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
dlllist -p 3912 | grep notepad.exe
Volatility Foundation Volatility Framework 2.6
notepad.exe pid:   3912
Command line : "C:\Windows\System32\notepad.exe"
0x00007ff6b1150000            0x1e000                0x0 C:\Windows\System32\notepad.exe
```

```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
dlllist -p 5764 | grep -i notepad.exe
Volatility Foundation Volatility Framework 2.6
notepad.exe pid:   5764
Command line : "C:\Windows\system32\NOTEPAD.EXE" C:\Users\IEUser\Desktop\soter.txt
0x00007ff6b1150000            0x41000                0x0 C:\Windows\system32\NOTEPAD.EXE
```

Figure 3 – DLL listing in test image

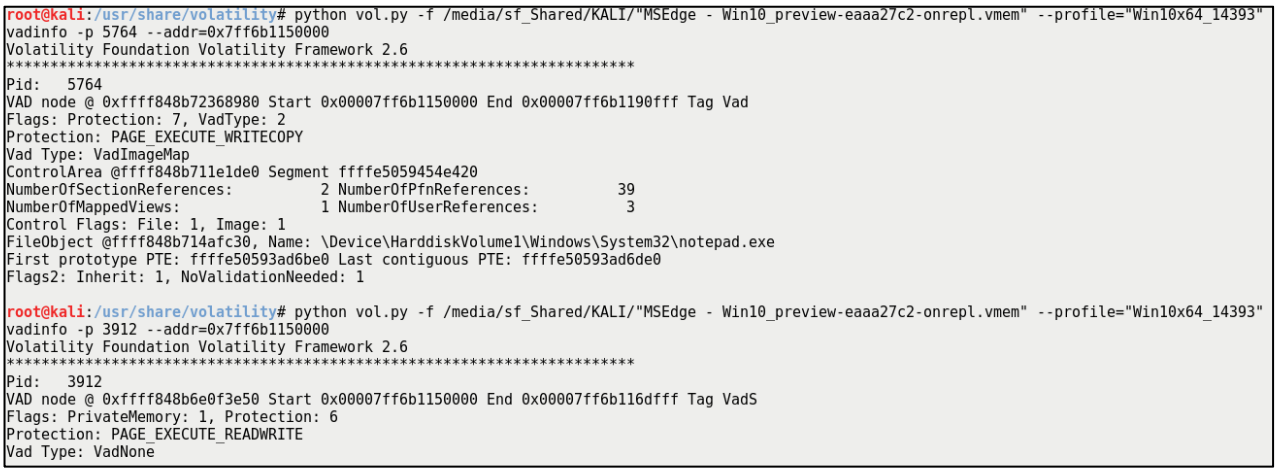
As seen above, both notepad processes (pid 3912 and 5764) have the same executables' full path and same image base, but not the same size. The last is an indication that something is wrong, as explained in a next step.

Virtual Address Descriptor information

As mentioned in earlier sections, VAD structure resides in kernel memory and contains information about contiguous process virtual address space allocation and in case there is an executable loaded, the VAD node contains information about start address, end address, permissions and the full path to the executable. Vadinfo is used to extract this information and a process is considered hollowed in case its VAD characteristics (for the specific **Image Base found with dlllist**) are different than its PEB characteristics.

```
python vol.py – f mem.bin
vadinfo -p <suspicious process_id> --addr=<Image Base>
```

One possible anomaly of process replacement is that the name field of **FileObject**, which contains the memory mapped file name, does not exist. This is the result of process hollowing because when memory unmap is done, the name entry is removed from VAD structure and it is no longer associated with the specific memory region, while the PEB structure is not affected by unmapping [6]. The fact that FileObject pointer is none/NULL, means that the memory isn't backed by a file. It would be backed by a file if the PE was loaded via LoadLibrary [45].



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 5764 --addr=0x7ff6b1150000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:   5764
VAD node @ 0xffff848b72368980 Start 0x00007ff6b1150000 End 0x00007ff6b1190fff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffff848b711e1de0 Segment ffffe5059454e420
NumberOfSectionReferences:          2 NumberOfPfnReferences:          39
NumberOfMappedViews:                1 NumberOfUserReferences:          3
Control Flags: File: 1, Image: 1
FileObject @ffff848b714afc30, Name: \Device\HarddiskVolume1\Windows\System32\notepad.exe
First prototype PTE: ffffe50593ad6be0 Last contiguous PTE: ffffe50593ad6de0
Flags2: Inherit: 1, NoValidationNeeded: 1

root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 3912 --addr=0x7ff6b1150000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:   3912
VAD node @ 0xffff848b6e0f3e50 Start 0x00007ff6b1150000 End 0x00007ff6b116dfff Tag VadS
Flags: PrivateMemory: 1, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
Vad Type: VadNone
```

Figure 4 – VAD information in test image

The legitimates' process (pid 5764) VAD information is correct (name field of FileObject contains the executable's full path) while the corresponding entry for the hollowed process (pid 3912) does not exists, meaning it is null. The output of vadinfo plugin is used for other forensics reasons too, as it will be shown in the following sections.

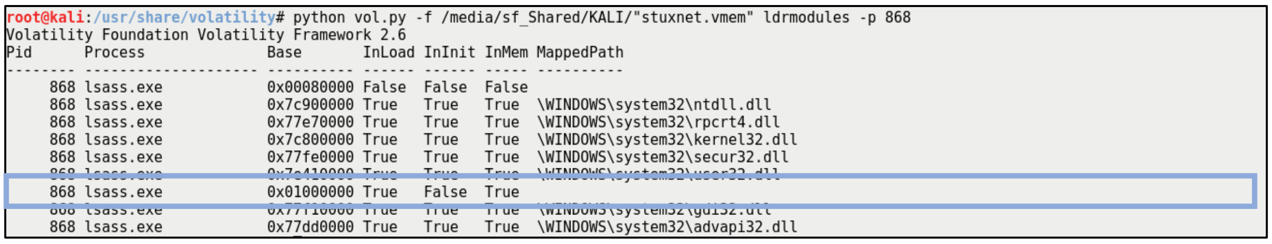
Module linked lists and VAD

When the process is loaded, the process executable is added to the load order, init order and memory order module lists in the PEB. The plugin ldrmodules cross-references this information with the memory-mapped files in the VAD. Specifically, it looks for large nodes with PAGE_EXECUTE_WRITECOPY protections, a VadImageMap type, and the Image control flag set [6].

```
python vol.py – f mem.bin
ldrmodules –p <suspicious process_id> | grep <Image Base>
```

It is suspicious when the **MappedPath** column (the name field of the memory mapped file) is blank for the specific Image Base Address or there isn't any output for the specific Image base address. These are indications that most likely there is code injected via VirtualAlloc(Ex) and WriteProcessMemory. If the same full path exists in both dlllist and ldrmodules but in a different base, is also suspicious. That might happen for example in case of "No memory unmapping" variation.

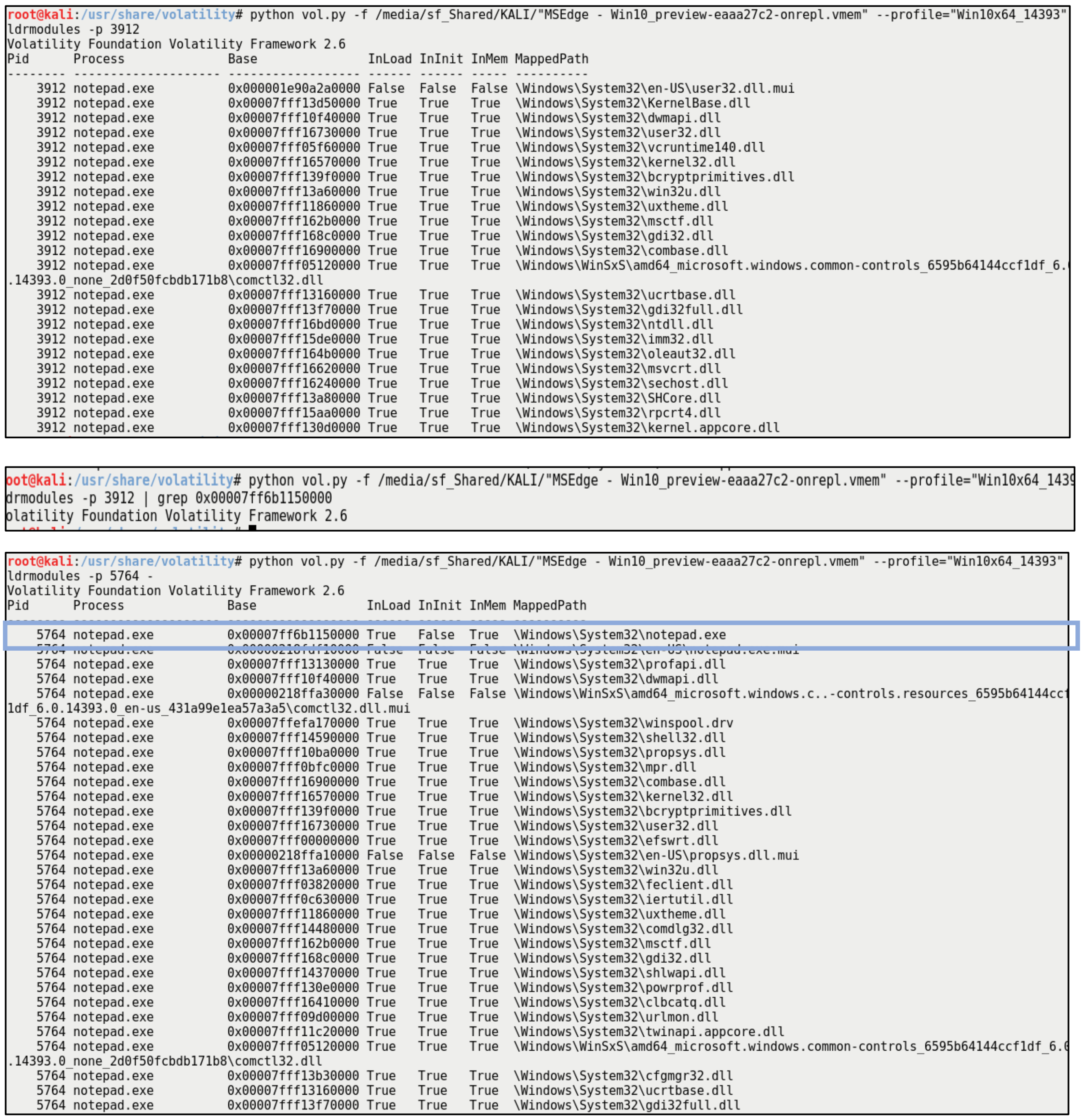
It follows the corresponding output using Stuxnet.vmem, which shows that the MappedPath column is blank for a hollowed process:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"stuxnet.vmem" ldrmodules -p 868
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base       InLoad InInit InMem MappedPath
-------- -------------------- ---------- ------ ------ ----- ----------
     868 lsass.exe            0x00080000 False  False  False
     868 lsass.exe            0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
     868 lsass.exe            0x77e70000 True   True   True  \WINDOWS\system32\rpcrt4.dll
     868 lsass.exe            0x7c800000 True   True   True  \WINDOWS\system32\kernel32.dll
     868 lsass.exe            0x77fe0000 True   True   True  \WINDOWS\system32\secur32.dll
     868 lsass.exe            0x7e410000 True   True   True  \WINDOWS\system32\user32.dll
     868 lsass.exe            0x01000000 True   False  True
     868 lsass.exe            0x77f10000 True   True   True  \WINDOWS\system32\gdi32.dll
     868 lsass.exe            0x77dd0000 True   True   True  \WINDOWS\system32\advapi32.dll
```

Figure 5 – Comparing module linked list and VAD information in Stuxnet.vmem

But using the demonstration test image, there is no output for the hollowed process's (pid 3912) Image Base, compared to the legitimate one (pid 5764):



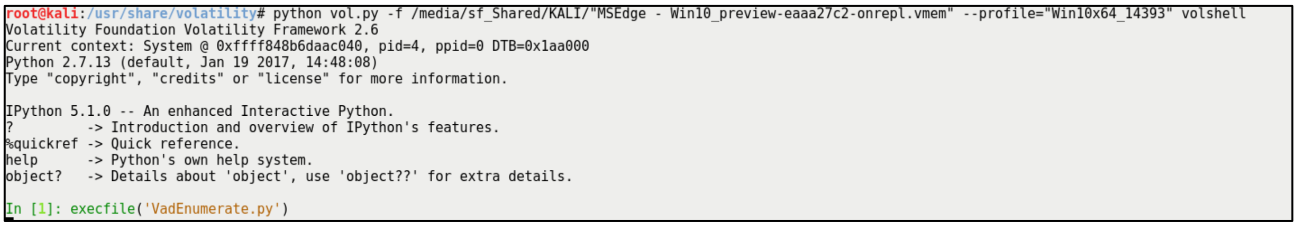
```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
ldrmodules -p 3912
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    3912 notepad.exe          0x000001e90a2a0000 False  False  False \Windows\System32\en-US\user32.dll.mui
    3912 notepad.exe          0x00007fff13d50000 True   True   True  \Windows\System32\KernelBase.dll
    3912 notepad.exe          0x00007fff10f40000 True   True   True  \Windows\System32\dwmapi.dll
    3912 notepad.exe          0x00007fff16730000 True   True   True  \Windows\System32\user32.dll
    3912 notepad.exe          0x00007fff05f60000 True   True   True  \Windows\System32\vcruntime140.dll
    3912 notepad.exe          0x00007fff16570000 True   True   True  \Windows\System32\kernel32.dll
    3912 notepad.exe          0x00007fff139f0000 True   True   True  \Windows\System32\bcryptprimitives.dll
    3912 notepad.exe          0x00007fff13a60000 True   True   True  \Windows\System32\win32u.dll
    3912 notepad.exe          0x00007fff11860000 True   True   True  \Windows\System32\uxtheme.dll
    3912 notepad.exe          0x00007fff162b0000 True   True   True  \Windows\System32\msctf.dll
    3912 notepad.exe          0x00007fff168c0000 True   True   True  \Windows\System32\gdi32.dll
    3912 notepad.exe          0x00007fff16900000 True   True   True  \Windows\System32\combase.dll
    3912 notepad.exe          0x00007fff05120000 True   True   True  \Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0
.14393.0_none_2d0f50fcbdb171b8\comctl32.dll
    3912 notepad.exe          0x00007fff13160000 True   True   True  \Windows\System32\ucrtbase.dll
    3912 notepad.exe          0x00007fff13f70000 True   True   True  \Windows\System32\gdi32full.dll
    3912 notepad.exe          0x00007fff16bd0000 True   True   True  \Windows\System32\ntdll.dll
    3912 notepad.exe          0x00007fff15de0000 True   True   True  \Windows\System32\imm32.dll
    3912 notepad.exe          0x00007fff164b0000 True   True   True  \Windows\System32\oleaut32.dll
    3912 notepad.exe          0x00007fff16620000 True   True   True  \Windows\System32\msvcrt.dll
    3912 notepad.exe          0x00007fff16240000 True   True   True  \Windows\System32\sechost.dll
    3912 notepad.exe          0x00007fff13a80000 True   True   True  \Windows\System32\SHCore.dll
    3912 notepad.exe          0x00007fff15aa0000 True   True   True  \Windows\System32\rpcrt4.dll
    3912 notepad.exe          0x00007fff130d0000 True   True   True  \Windows\System32\kernel.appcore.dll
```

```
oot@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_1439
drmodules -p 3912 | grep 0x00007ff6b1150000
olatility Foundation Volatility Framework 2.6
```

```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
ldrmodules -p 5764 -
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    5764 notepad.exe          0x00007ff6b1150000 True   False  True  \Windows\System32\notepad.exe
    5764 notepad.exe          0x00000218fdf10000 False  False  False \Windows\System32\en-US\notepad.exe.mui
    5764 notepad.exe          0x00007fff13130000 True   True   True  \Windows\System32\profapi.dll
    5764 notepad.exe          0x00007fff10f40000 True   True   True  \Windows\System32\dwmapi.dll
    5764 notepad.exe          0x00000218ffa30000 False  False  False \Windows\WinSxS\amd64_microsoft.windows.c..-controls.resources_6595b64144cc
1df_6.0.14393.0_en-us_431a99e1ea57a3a5\comctl32.dll.mui
    5764 notepad.exe          0x00007ffefa170000 True   True   True  \Windows\System32\winspool.drv
    5764 notepad.exe          0x00007fff14590000 True   True   True  \Windows\System32\shell32.dll
    5764 notepad.exe          0x00007fff10ba0000 True   True   True  \Windows\System32\propsys.dll
    5764 notepad.exe          0x00007fff0bfc0000 True   True   True  \Windows\System32\mpr.dll
    5764 notepad.exe          0x00007fff16900000 True   True   True  \Windows\System32\combase.dll
    5764 notepad.exe          0x00007fff16570000 True   True   True  \Windows\System32\kernel32.dll
    5764 notepad.exe          0x00007fff139f0000 True   True   True  \Windows\System32\bcryptprimitives.dll
    5764 notepad.exe          0x00007fff16730000 True   True   True  \Windows\System32\user32.dll
    5764 notepad.exe          0x00007fff00000000 True   True   True  \Windows\System32\efswrt.dll
    5764 notepad.exe          0x00000218ffa10000 False  False  False \Windows\System32\en-US\propsys.dll.mui
    5764 notepad.exe          0x00007fff13a60000 True   True   True  \Windows\System32\win32u.dll
    5764 notepad.exe          0x00007fff03820000 True   True   True  \Windows\System32\feclient.dll
    5764 notepad.exe          0x00007fff0c630000 True   True   True  \Windows\System32\iertutil.dll
    5764 notepad.exe          0x00007fff11860000 True   True   True  \Windows\System32\uxtheme.dll
    5764 notepad.exe          0x00007fff14480000 True   True   True  \Windows\System32\comdlg32.dll
    5764 notepad.exe          0x00007fff162b0000 True   True   True  \Windows\System32\msctf.dll
    5764 notepad.exe          0x00007fff168c0000 True   True   True  \Windows\System32\gdi32.dll
    5764 notepad.exe          0x00007fff14370000 True   True   True  \Windows\System32\shlwapi.dll
    5764 notepad.exe          0x00007fff130e0000 True   True   True  \Windows\System32\powrprof.dll
    5764 notepad.exe          0x00007fff16410000 True   True   True  \Windows\System32\clbcatq.dll
    5764 notepad.exe          0x00007fff09d00000 True   True   True  \Windows\System32\urlmon.dll
    5764 notepad.exe          0x00007fff11c20000 True   True   True  \Windows\System32\twinapi.appcore.dll
    5764 notepad.exe          0x00007fff05120000 True   True   True  \Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0
.14393.0_none_2d0f50fcbdb171b8\comctl32.dll
    5764 notepad.exe          0x00007fff13b30000 True   True   True  \Windows\System32\cfgmgr32.dll
    5764 notepad.exe          0x00007fff13160000 True   True   True  \Windows\System32\ucrtbase.dll
    5764 notepad.exe          0x00007fff13f70000 True   True   True  \Windows\System32\gdi32full.dll
```

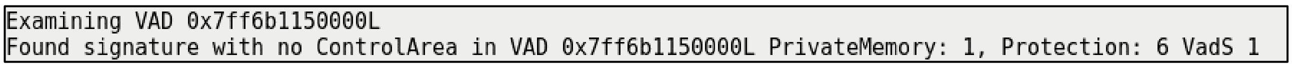
Figure 6 – Comparing module linked list and VAD information in test images

The fact that ldrmodules gives not any output for the hollowed process at Image Base Address made impression and triggered an in-depth analysis of ldrmodules plugin. It turned out that ldrmodules filters the VADs with mapped_file_filter, meaning Private memory flag equals to zero and vad.ControlArea is set. As shown in Figure 4 – VAD information in test image, ControlArea is not set. Also, ldrmodules looks for large nodes with PAGE_EXECUTE_WRITECOPY protections [6]. For reasons of more profound research, a python script is written to enumerate all the Vads' characteristics of the specific process from the volatility's volshell environment and the output of it is (regarding the specific Image Base Address):



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393" volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0xffff848b6daac040, pid=4, ppid=0 DTB=0x1aa000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: execfile('VadEnumerate.py')
```

[snip]

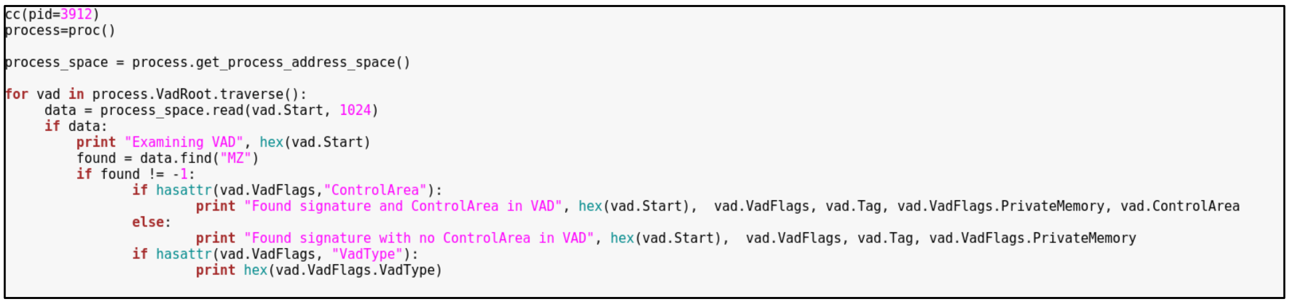


```
Examining VAD 0x7ff6b1150000L
Found signature with no ControlArea in VAD 0x7ff6b1150000L PrivateMemory: 1, Protection: 6 VadS 1
```

[snip]

The above is also confirmed from the vadinfo output in the previous step.

The code of VadEnumerate.py is:



```python
cc(pid=3912)
process=proc()

process_space = process.get_process_address_space()

for vad in process.VadRoot.traverse():
     data = process_space.read(vad.Start, 1024)
     if data:
         print "Examining VAD", hex(vad.Start)
         found = data.find("MZ")
         if found != -1:
                if hasattr(vad.VadFlags,"ControlArea"):
                        print "Found signature and ControlArea in VAD", hex(vad.Start),  vad.VadFlags, vad.Tag, vad.VadFlags.PrivateMemory, vad.ControlArea
                else:
                        print "Found signature with no ControlArea in VAD", hex(vad.Start),  vad.VadFlags, vad.Tag, vad.VadFlags.PrivateMemory
                if hasattr(vad.VadFlags, "VadType"):
                        print hex(vad.VadFlags.VadType)
```

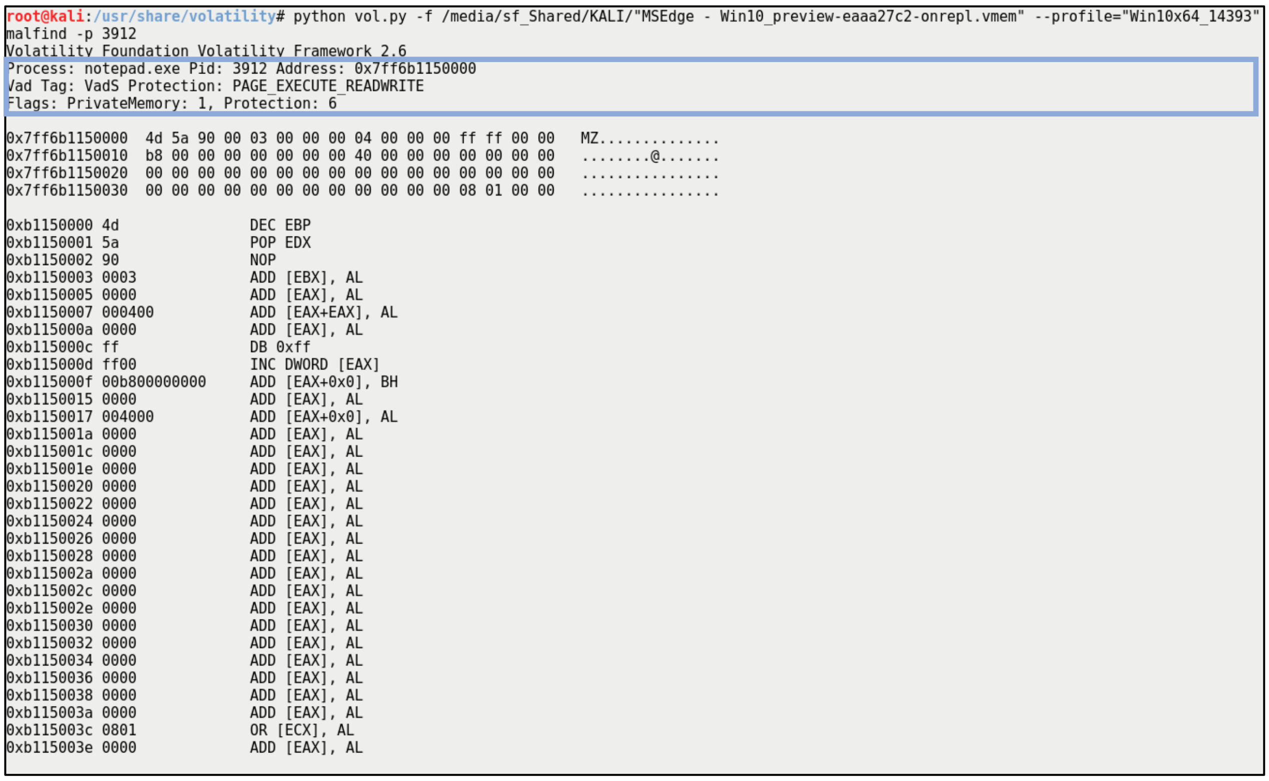
2.5.3 Checking memory permissions

As described in the 4th step of hollowing a process, the hollowing process allocates a new memory region with PAGE_EXECUTE_READWRITE permissions, while an executable normally loaded has PAGE_EXECUTE_WRITECOPY permission. This difference is spotted at **Protection** field in vadinfo output: Protection equals to 6 (PAGE_EXECUTE_READWRITE) instead of 7 (PAGE_EXECUTE_WRITECOPY).

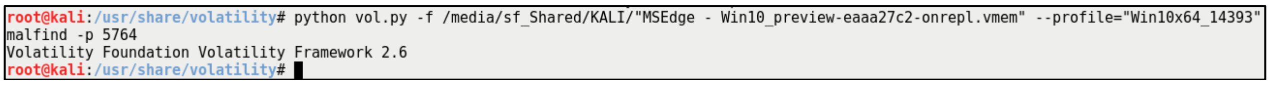
The malfind plugin is designed to hunt down remote code injections. The concept is that there will be a readable, writeable, and executable private memory region (that is, no file mapping) with all pages committed. The region will contain a PE header and/or valid CPU instructions [6], so it is used:

```
python vol.py – f mem.bin
malfind –p <suspicious process_id>
```

Malfind plugin spotted the suspicious memory permissions in the suspicious process (pid 3912), while it gives no output for the legitimates one (pid 5764):



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
malfind -p 3912
Volatility Foundation Volatility Framework 2.6
Process: notepad.exe Pid: 3912 Address: 0x7ff6b1150000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

0x7ff6b1150000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff6b1150010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff6b1150020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff6b1150030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

0xb1150000 4d              DEC EBP
0xb1150001 5a              POP EDX
0xb1150002 90              NOP
0xb1150003 0003            ADD [EBX], AL
0xb1150005 0000            ADD [EAX], AL
0xb1150007 000400          ADD [EAX+EAX], AL
0xb115000a 0000            ADD [EAX], AL
0xb115000c ff              DB 0xff
0xb115000d ff00            INC DWORD [EAX]
0xb115000f 00b800000000    ADD [EAX+0x0], BH
0xb1150015 0000            ADD [EAX], AL
0xb1150017 004000          ADD [EAX+0x0], AL
0xb115001a 0000            ADD [EAX], AL
0xb115001c 0000            ADD [EAX], AL
0xb115001e 0000            ADD [EAX], AL
0xb1150020 0000            ADD [EAX], AL
0xb1150022 0000            ADD [EAX], AL
0xb1150024 0000            ADD [EAX], AL
0xb1150026 0000            ADD [EAX], AL
0xb1150028 0000            ADD [EAX], AL
0xb115002a 0000            ADD [EAX], AL
0xb115002c 0000            ADD [EAX], AL
0xb115002e 0000            ADD [EAX], AL
0xb1150030 0000            ADD [EAX], AL
0xb1150032 0000            ADD [EAX], AL
0xb1150034 0000            ADD [EAX], AL
0xb1150036 0000            ADD [EAX], AL
0xb1150038 0000            ADD [EAX], AL
0xb115003a 0000            ADD [EAX], AL
0xb115003c 0801            OR [ECX], AL
0xb115003e 0000            ADD [EAX], AL
```



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
malfind -p 5764
Volatility Foundation Volatility Framework 2.6
root@kali:/usr/share/volatility# 
```

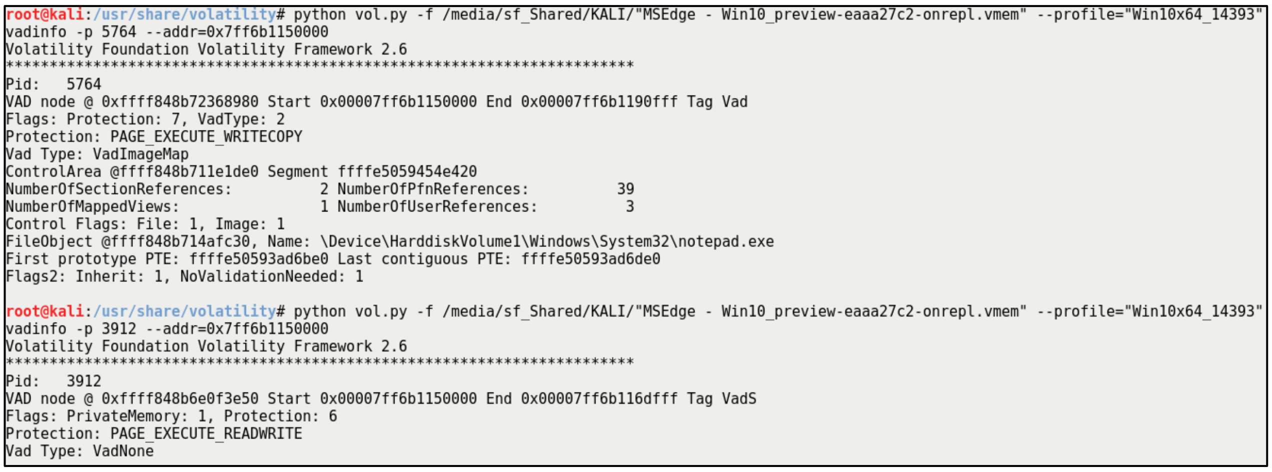
Figure 7 – Checking memory permissions in test image

For the legitimate process, there should not be any output *, but we observe that the suspicious one has Protection: PAGE_EXECUTE_READWRITE at addr=<Image Base> and the hex dump begins with an MZ signature, indicating that there is an executable at the specific address.

* Legitimate programs may allocate executable private memory for legitimate reasons. So, the malfind plugin may produce false positive results. One way to distinguish the false positive is to look for MZ signature in hex dump.

2.5.4 Vadinfo – Looking Deeper

For now, vadinfo volatility plugin is used for comparing kernel and process memory structures, but there are some extra points that need attention. One can compare the vadinfo output for the legitimate (pid 5764) and suspicious process (pid 3912), to make these points more comprehensible.



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 5764 --addr=0x7ff6b1150000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:   5764
VAD node @ 0xffff848b72368980 Start 0x00007ff6b1150000 End 0x00007ff6b1190fff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffff848b711e1de0 Segment ffffe5059454e420
NumberOfSectionReferences:          2 NumberOfPfnReferences:          39
NumberOfMappedViews:                1 NumberOfUserReferences:          3
Control Flags: File: 1, Image: 1
FileObject @ffff848b714afc30, Name: \Device\HarddiskVolume1\Windows\System32\notepad.exe
First prototype PTE: ffffe50593ad6be0 Last contiguous PTE: ffffe50593ad6de0
Flags2: Inherit: 1, NoValidationNeeded: 1

root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 3912 --addr=0x7ff6b1150000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:   3912
VAD node @ 0xffff848b6e0f3e50 Start 0x00007ff6b1150000 End 0x00007ff6b116dfff Tag VadS
Flags: PrivateMemory: 1, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
Vad Type: VadNone
```

Figure 8 – Comparing VAD of legitimate and suspicious process

Besides **File Object's Name** and **Protection** fields that have been already mentioned, it is observed that the VADs' flags have the **following characteristics** that could be used to reduce the false positive results or confirm the injection: [6][35][33][36]

- The hollowed process's VAD is marked as **private memory** (PrivateMemory :1 in flags output line), meaning it is not shared with or inherited by other processes. Executables and DLLs can be shared with other processes. A process' memory ranges allocated with VirtualAlloc or VirtualAllocEx are usually marked as private. Thus, if the PrivateMemory bit is set for a memory region is a factor when looking for injection.
- The hollowed process's VAD has a **VadS** tag which means there is no memory mapped file already occupying the space. Dynamically allocated memory pages created via VirtualAllocEx /WriteProcessMemory are of type _MMVAD_SHORT (VadS). Regular loaded libraries in the address space of a process are of type _MMVAD (Vad) or _MMVAD_LONG (VadL), meaning it is representing a memory mapped file.
- Only the legitimate process has a copy of its executable into the region, meaning that the output fields **VadImageMap** (in Vad Type output line) and Image (in Control Flag output line) have value 1.
- The hollow process variation that changes each PE section with the appropriate permissions, described previous section, is covered by the way vadinfo works:
  It displays the original protection specified for all pages in the range when they were first reserved or committed [18] and because the initial memory allocation is made with protection PAGE_EXECUTE_READWRITE, this info is revealed by this plugin.
  If the hollowing process used VirtualAlloc to reserve for example ten pages with PAGE_NOACCESS and later it commits three of the pages as PAGE_EXECUTE_READWRITE and four others as PAGE_READONLY, the Protection field still contains PAGE_NOACCESS. Older Zeus samples from 2006 used this technique, so its injected memory regions appeared as PAGE_NOACCESS [6]

- In some cases, there might exist a File **Object's Name and should be confirmed that is the same as the one stated in ImagePathName** of the PEB (Process Environment Block), displayed by pslist plugin.

2.5.5 Hollowfind plugin

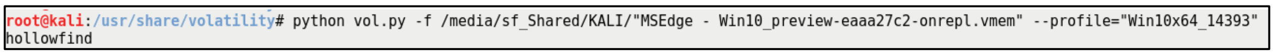
HollowFind is a Volatility plugin created to detect different types of process hollowing techniques created by Monnappa K.A. [56][35], who won the 2016 Volatility Plugin [21].

```
python vol.py – f mem.bin hollowfind
```

HollowFind works (in general) as follows:

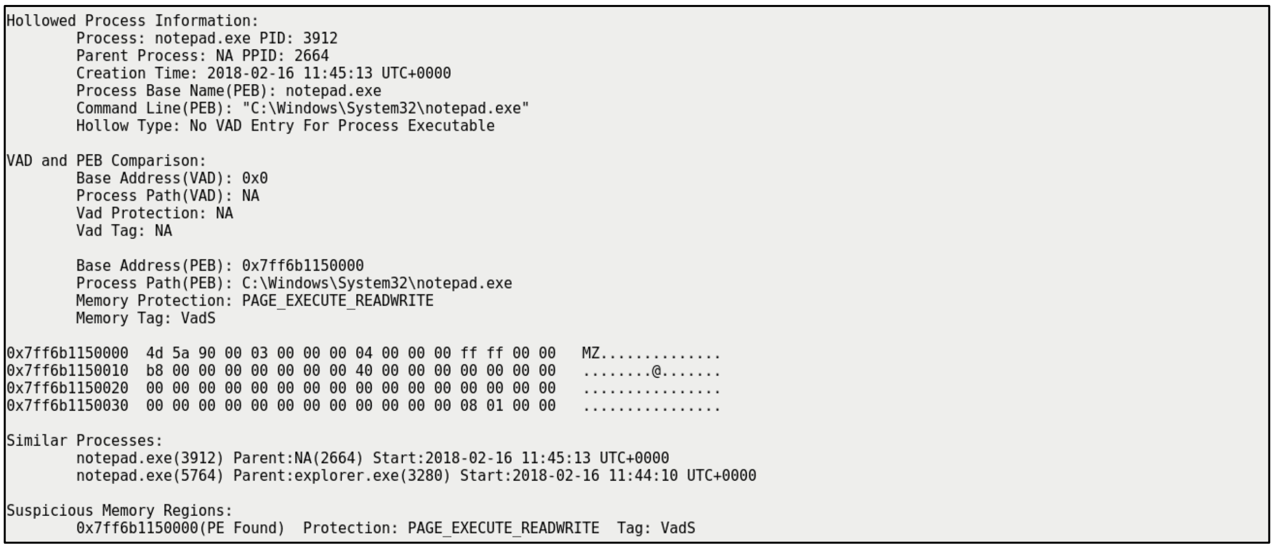
1. Creates a list of processes using pslist plugin (unless it is used with a -p pid parameter)
2. For each process, it retrieves information from the process structure, PEB and the VAD that is pointed by PEB.ImageBaseAddress and checks whether there is an executable in the processes' s VADs
3. Detects whether the process is hollowed. The criteria used are:
   - VAD's Protection equals to PAGE_EXECUTE_READWRITE or
   - There is no VAD entry corresponding to PEB.ImageBaseAddress
   - There is an executable in processes' VADs that PEB.ImageBaseAddress does not point to and has PAGE_EXECUTE_WRITECOPY protection
4. For each suspect process, it displays similar processes' information

Hollowfind detected the hollowed process in test image, as shown below:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
hollowfind
```

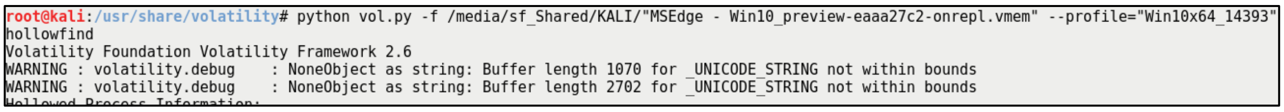
[snip]



```
Hollowed Process Information:
        Process: notepad.exe PID: 3912
        Parent Process: NA PPID: 2664
        Creation Time: 2018-02-16 11:45:13 UTC+0000
        Process Base Name(PEB): notepad.exe
        Command Line(PEB): "C:\Windows\System32\notepad.exe"
        Hollow Type: No VAD Entry For Process Executable

VAD and PEB Comparison:
        Base Address(VAD): 0x0
        Process Path(VAD): NA
        Vad Protection: NA
        Vad Tag: NA

        Base Address(PEB): 0x7ff6b1150000
        Process Path(PEB): C:\Windows\System32\notepad.exe
        Memory Protection: PAGE_EXECUTE_READWRITE
        Memory Tag: VadS

0x7ff6b1150000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff6b1150010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff6b1150020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff6b1150030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

Similar Processes:
        notepad.exe(3912) Parent:NA(2664) Start:2018-02-16 11:45:13 UTC+0000
        notepad.exe(5764) Parent:explorer.exe(3280) Start:2018-02-16 11:44:10 UTC+0000

Suspicious Memory Regions:
        0x7ff6b1150000(PE Found)  Protection: PAGE_EXECUTE_READWRITE  Tag: VadS
```

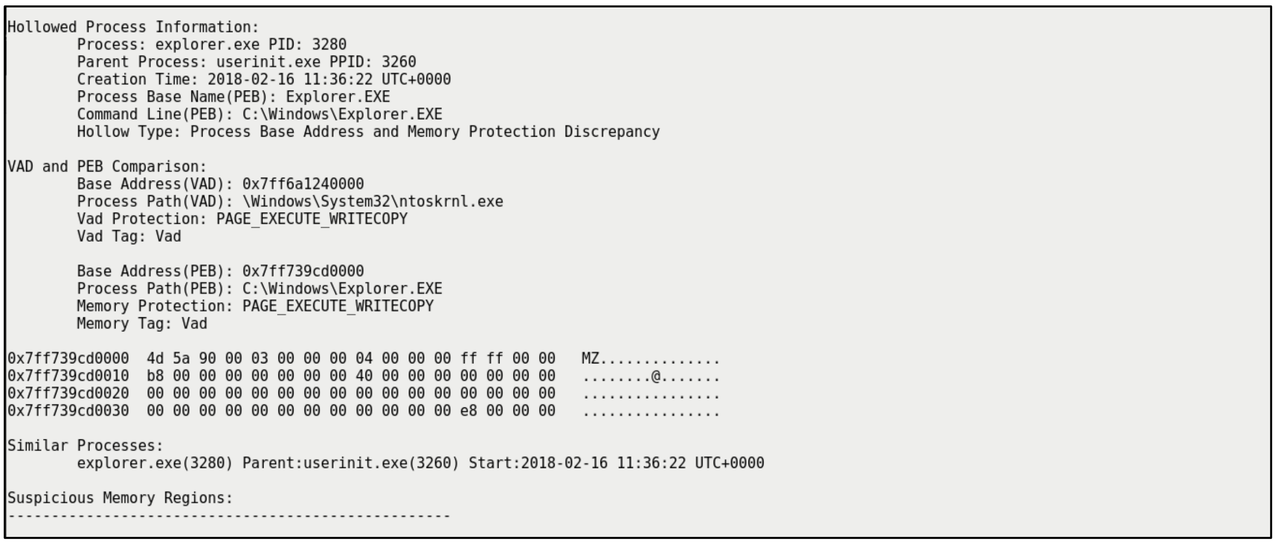
Figure 9 – Hollowfind plugin in test image

But it produced some false positive results also, like the following, regarding process explorer.exe, with pid 3280:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
hollowfind
Volatility Foundation Volatility Framework 2.6
WARNING : volatility.debug    : NoneObject as string: Buffer length 1070 for _UNICODE_STRING not within bounds
WARNING : volatility.debug    : NoneObject as string: Buffer length 2702 for _UNICODE_STRING not within bounds
```

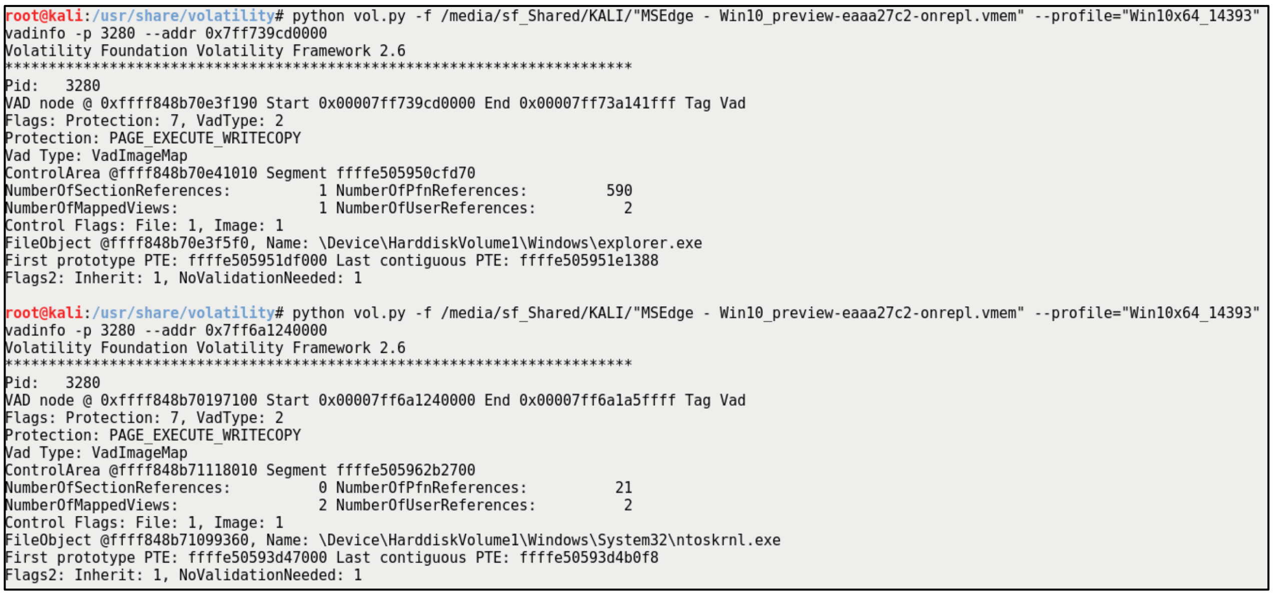
[snip]



```
Hollowed Process Information:
        Process: explorer.exe PID: 3280
        Parent Process: userinit.exe PPID: 3260
        Creation Time: 2018-02-16 11:36:22 UTC+0000
        Process Base Name(PEB): Explorer.EXE
        Command Line(PEB): C:\Windows\Explorer.EXE
        Hollow Type: Process Base Address and Memory Protection Discrepancy

VAD and PEB Comparison:
        Base Address(VAD): 0x7ff6a1240000
        Process Path(VAD): \Windows\System32\ntoskrnl.exe
        Vad Protection: PAGE_EXECUTE_WRITECOPY
        Vad Tag: Vad

        Base Address(PEB): 0x7ff739cd0000
        Process Path(PEB): C:\Windows\Explorer.EXE
        Memory Protection: PAGE_EXECUTE_WRITECOPY
        Memory Tag: Vad

0x7ff739cd0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff739cd0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff739cd0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff739cd0030  00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00   ................

Similar Processes:
        explorer.exe(3280) Parent:userinit.exe(3260) Start:2018-02-16 11:36:22 UTC+0000

Suspicious Memory Regions:
---------------------------------------------------
```

Figure 10 – Hollowfind plugin false positives

Analyzing the corresponding VADs' information, it is confirmed that it is indeed a false positive alarm:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 3280 --addr 0x7ff739cd0000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:    3280
VAD node @ 0xffff848b70e3f190 Start 0x00007ff739cd0000 End 0x00007ff73a141fff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffff848b70e41010 Segment ffffe505950cfd70
NumberOfSectionReferences:          1 NumberOfPfnReferences:        590
NumberOfMappedViews:                1 NumberOfUserReferences:         2
Control Flags: File: 1, Image: 1
FileObject @ffff848b70e3f5f0, Name: \Device\HarddiskVolume1\Windows\explorer.exe
First prototype PTE: ffffe505951df000 Last contiguous PTE: ffffe505951e1388
Flags2: Inherit: 1, NoValidationNeeded: 1

root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
vadinfo -p 3280 --addr 0x7ff6a1240000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:    3280
VAD node @ 0xffff848b70197100 Start 0x00007ff6a1240000 End 0x00007ff6a1a5ffff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffff848b71118010 Segment ffffe505962b2700
NumberOfSectionReferences:          0 NumberOfPfnReferences:         21
NumberOfMappedViews:                2 NumberOfUserReferences:         2
Control Flags: File: 1, Image: 1
FileObject @ffff848b71099360, Name: \Device\HarddiskVolume1\Windows\System32\ntoskrnl.exe
First prototype PTE: ffffe50593d47000 Last contiguous PTE: ffffe50593d4b0f8
Flags2: Inherit: 1, NoValidationNeeded: 1
```

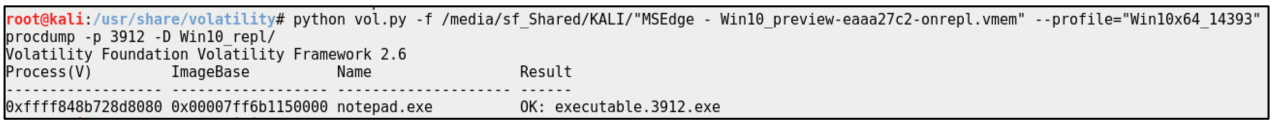
A good strategy could be to run hollowfind plugin and then follow the steps described in Vadinfo – Looking Deeper.

2.5.6 Extracting executables

The volatility plugin procdump [18] is used to dump a process's executable to disk.

```
mkdir dump
python vol.py – f mem.bin
procdump –p <suspicious process_id> -D dump/
ls -a -l dump
```

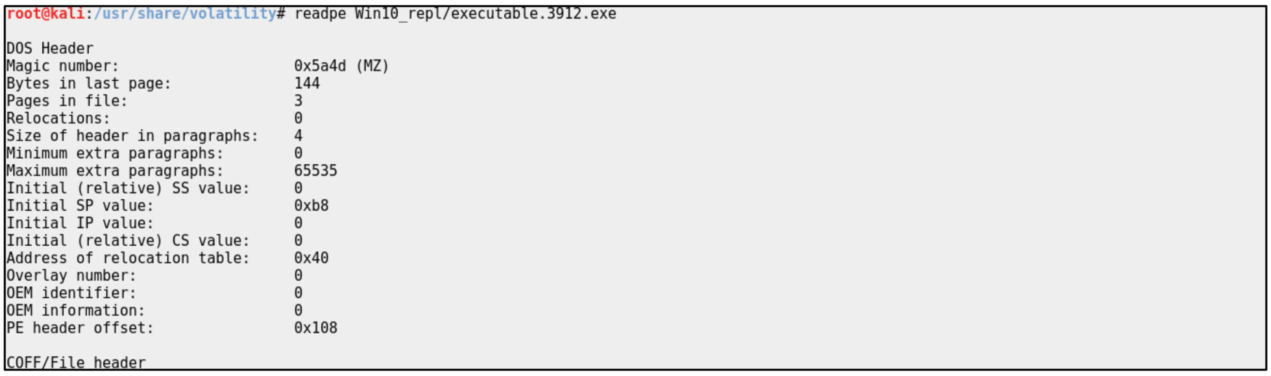
Similar plugins that could be used are dlldump, vaddump or memdump. The difference between memdump, procexedump and procmemdump is analyzed in [57].



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
procdump -p 3912 -D Win10_repl/
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffff848b728d8080 0x00007ff6b1150000 notepad.exe          OK: executable.3912.exe
```

Figure 11 – Extracting malicious executable in test image

The executables are now available for further static analysis using file or strings commands, ssdeep [https://ssdeep-project.github.io/ssdeep/], IDA Pro, readpe or a hex editor, YARA and other tools. YARA detects hidden and injected code and provides a framework for general-purpose signature-based memory scanning [33][43]. This is helpful especially in case there are any IOC (Indicators Of Compromise). For example, in case there is a suspicion that the malicious executable references libraries such as LoadLibrary and GetProcAddress, someone could look for them. As another example, using the string command on executable.3912.exe, "MessageBoxW" string is found which probably is the command that pops the "malicious" window. The executables could also be submitted for analysis, for example to virustotal or using the volatility's vscan plugin. Static analysis is outside the scope of this document, however here is an example of using readpe:



```
root@kali:/usr/share/volatility# readpe Win10_repl/executable.3912.exe

DOS Header
Magic number:                    0x5a4d (MZ)
Bytes in last page:              144
Pages in file:                   3
Relocations:                     0
Size of header in paragraphs:    4
Minimum extra paragraphs:        0
Maximum extra paragraphs:        65535
Initial (relative) SS value:     0
Initial SP value:                0xb8
Initial IP value:                0
Initial (relative) CS value:     0
Address of relocation table:     0x40
Overlay number:                  0
OEM identifier:                  0
OEM information:                 0
PE header offset:                0x108

COFF/File header
```

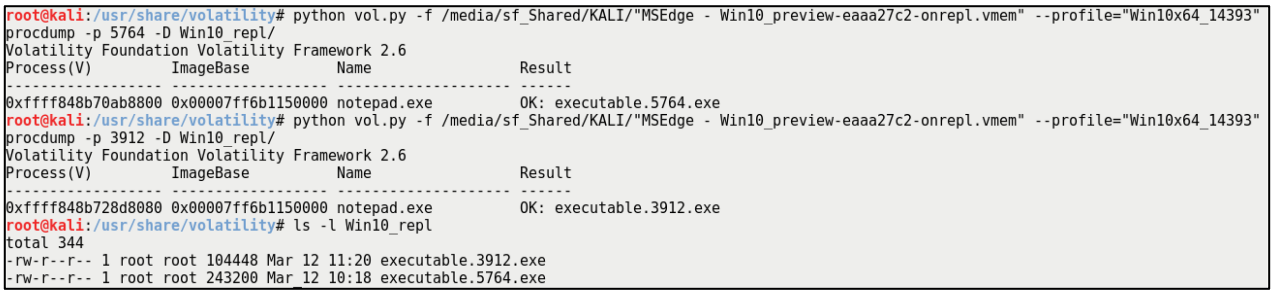
Figure 12 – Extracting malicious executable in test image

2.5.7 Comparing executables sizes

Now that there is a strong indication about which process is injected and which is the legitimate one, it is useful to compare their executable sizes to evaluate whether the executables are different. Alternatively, diff command could be used.

Another way to compare the executables is to retrieve the percentage of similarity between them using fuzzy hashing. The industry standard tool for fuzzy hashing is called 'ssdeep' [43].

As already mentioned and seen in the next figure, the aforementioned executables' sizes are different:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
procdump -p 5764 -D Win10_repl/
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffff848b70ab8800 0x00007ff6b1150000 notepad.exe          OK: executable.5764.exe
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem" --profile="Win10x64_14393"
procdump -p 3912 -D Win10_repl/
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffff848b728d8080 0x00007ff6b1150000 notepad.exe          OK: executable.3912.exe
root@kali:/usr/share/volatility# ls -l Win10_repl
total 344
-rw-r--r-- 1 root root 104448 Mar 12 11:20 executable.3912.exe
-rw-r--r-- 1 root root 243200 Mar_12 10:18 executable.5764.exe
```

Figure 13 – Comparing malicious and legit executable in test image

2.5.8 Determining the owner of the hollowed process

Security Identifiers (SID) identify user, group and computer accounts. Every account has a unique SID. Each process in Windows has an associated token that describes which Security Identifier (SID) owns the process and what kind of privileges have been granted to it [44]. Viewing the SID associated with a process is useful only when the hollowed process belongs to specific users, like winlogon.exe, so it can be compared with the legitimates' process information. Either way, it can help identifying processes which have maliciously escalated privileges.

```
python vol.py – f mem.bin getsids -p <suspicious process_id>, <legitimate process_id>
```

In our test image this does not apply, because the process notepad.exe doesn't belong to specific user. However, it follows an example taken from [33] that shows the difference between the SIDs for the legitimate winlogon.exe and a process which was started by a user from Explorer:

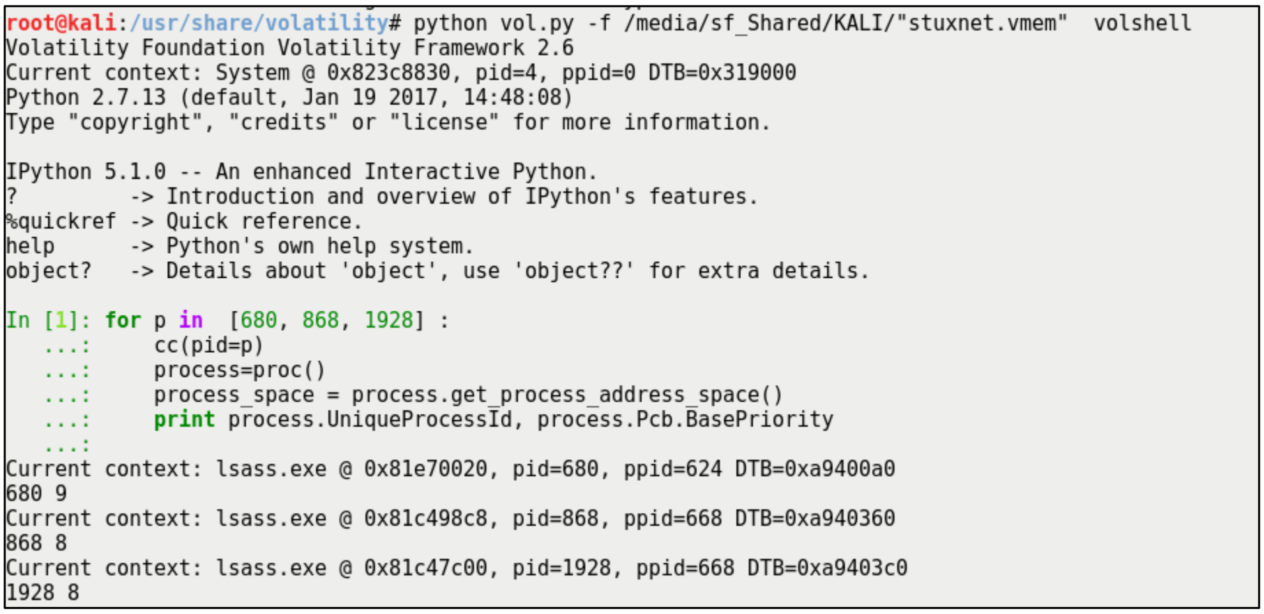
```
# This is a legitimate winlogon.exe
winlogon.exe (632): S-1-5-18 (Local System)
winlogon.exe (632): S-1-5-32-544 (Administrators)
winlogon.exe (632): S-1-1-0 (Everyone)
winlogon.exe (632): S-1-5-11 (Authenticated Users)
# This is a process started from Explorer by the user
aelas.exe (1984): S-1-5-21-1614895754-436374069-839522115-500 (Administrator)
aelas.exe (1984): S-1-5-21-1614895754-436374069-839522115-513 (Domain Users)
aelas.exe (1984): S-1-1-0 (Everyone)
aelas.exe (1984): S-1-5-32-544 (Administrators)
aelas.exe (1984): S-1-5-32-545 (Users)
aelas.exe (1984): S-1-5-4 (Interactive)
aelas.exe (1984): S-1-5-11 (Authenticated Users)
aelas.exe (1984): S-1-5-5-0-59917 (Logon Session)
aelas.exe (1984): S-1-2-0 (Users with the ability to log in locally)
```

Based on the output, one should know that if he/she ever sees a process named winlogon.exe that has SID owners like the aelas.exe process, then the winlogon.exe is probably not the real winlogon.exe.

2.5.9 Priority level

User applications and services start with a normal base priority, so their initial thread typically executes at priority level 8. However, some Windows system processes (such as the session manager, service control manager, and local security authentication process) have a base process priority slightly higher than the default that ensures that the threads in these processes will all start at a higher priority than the default value of 8 [26]. In some cases, the hollowed process is a process that is normally created by Windows system processes, meaning that their threads have priority level higher than 8. So, priority level check and comparison make sense. The process base priority is stored in EPROCESS.Pcb.BasePriority. A simple Volatility's volshell script is created to retrieve this information.

In the test image, the hollowed process is notepad.exe and it does not belong to this category. Thus, comparing the priority does not give any evidence. Instead, the Stuxnet memory image is used as an example. Process 680 is the legitimate one and processes 868 and 1928 are injected. As shown below, the legitimate one has a higher priority than the other ones:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"stuxnet.vmem"  volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0x823c8830, pid=4, ppid=0 DTB=0x319000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: for p in  [680, 868, 1928] :
   ...:     cc(pid=p)
   ...:     process=proc()
   ...:     process_space = process.get_process_address_space()
   ...:     print process.UniqueProcessId, process.Pcb.BasePriority
   ...:
Current context: lsass.exe @ 0x81e70020, pid=680, ppid=624 DTB=0xa9400a0
680 9
Current context: lsass.exe @ 0x81c498c8, pid=868, ppid=668 DTB=0xa940360
868 8
Current context: lsass.exe @ 0x81c47c00, pid=1928, ppid=668 DTB=0xa9403c0
1928 8
```

Figure 14 – Comparing legitimates' and injected process' priority

This isn't a strong artifact, because it is possible to change the priority by using SetPriorityClass. Also, since the base priority of threads is inherited from the base priority of the process which owns the thread (unless SetThreadPriority is called), then the differences should be visible using the threads plugin [45].

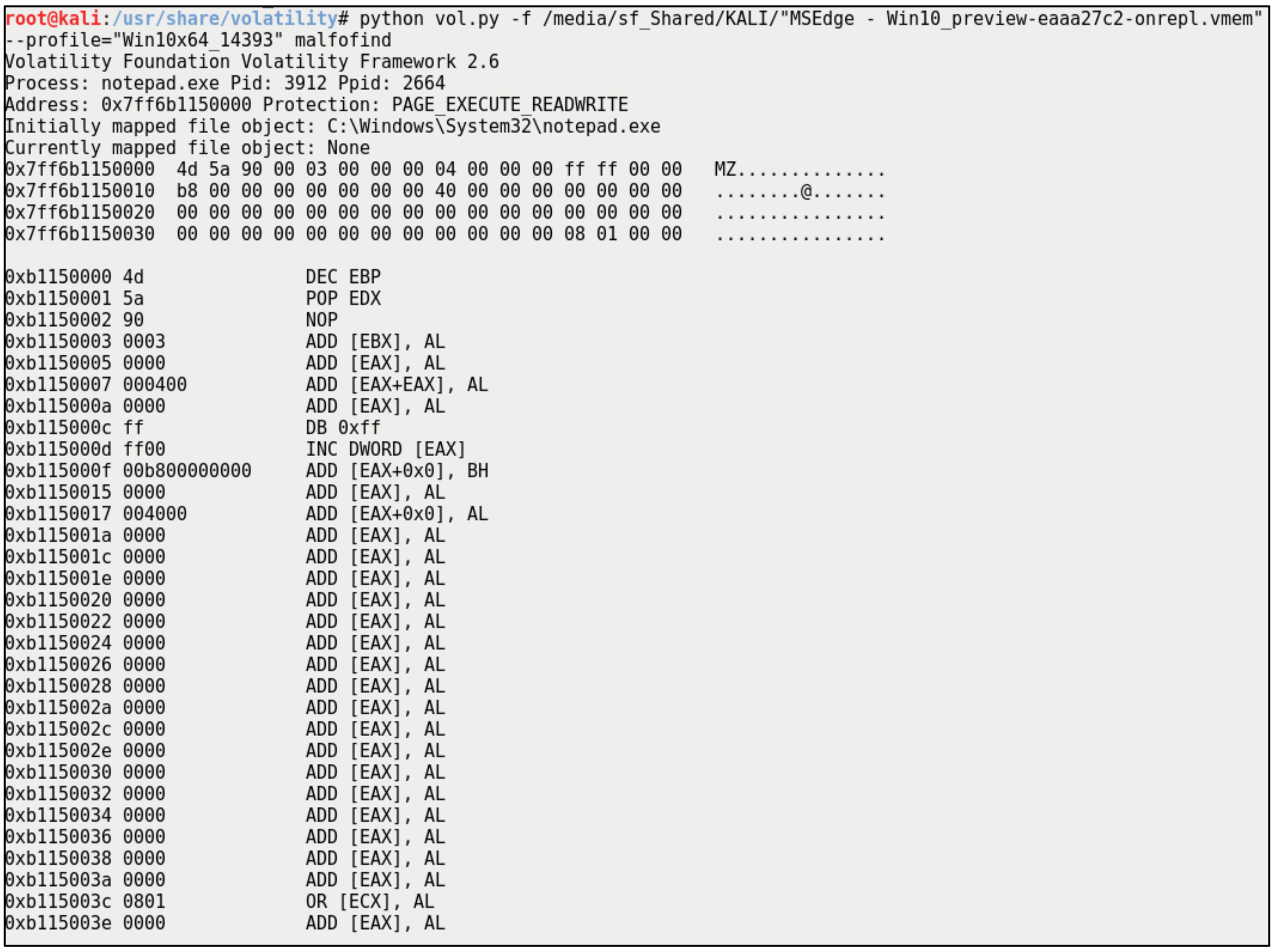
2.5.10 Malfofind plugin

At 2016 Volatility Plugin Contest, Dima Pshoul won the 3rd place with "Advanced Malware Hunter's Kit" [21] which includes Malfofind plugin, created to detect Process Hollowing.

As the author claims in his submission paper, "This plugin will scan currently loaded modules (using the VAD) for each process and will check if they are all accordingly mapped in the process' PEB. So why should this scanning technique work? If we examine the implementations of this technique we will notice how all the implementations contain the step of NtUnmapViewOfSection(ProcessHandle, ProcessImageBase) and then using VirtualAllocEx to map the injected executable into the same address. This will create an inconsistency between the VAD mapped images and the PEB linked images." [46].

```
python vol.py – f mem.bin malfofind
```

Malfofind was tested and successfully detected the hollowed process (pid 3912), as demonstrated in the next figure:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem"
--profile="Win10x64_14393" malfofind
Volatility Foundation Volatility Framework 2.6
Process: notepad.exe Pid: 3912 Ppid: 2664
Address: 0x7ff6b1150000 Protection: PAGE_EXECUTE_READWRITE
Initially mapped file object: C:\Windows\System32\notepad.exe
Currently mapped file object: None
0x7ff6b1150000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff6b1150010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff6b1150020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff6b1150030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

0xb1150000 4d               DEC EBP
0xb1150001 5a               POP EDX
0xb1150002 90               NOP
0xb1150003 0003             ADD [EBX], AL
0xb1150005 0000             ADD [EAX], AL
0xb1150007 000400           ADD [EAX+EAX], AL
0xb115000a 0000             ADD [EAX], AL
0xb115000c ff               DB 0xff
0xb115000d ff00             INC DWORD [EAX]
0xb115000f 00b800000000     ADD [EAX+0x0], BH
0xb1150015 0000             ADD [EAX], AL
0xb1150017 004000           ADD [EAX+0x0], AL
0xb115001a 0000             ADD [EAX], AL
0xb115001c 0000             ADD [EAX], AL
0xb115001e 0000             ADD [EAX], AL
0xb1150020 0000             ADD [EAX], AL
0xb1150022 0000             ADD [EAX], AL
0xb1150024 0000             ADD [EAX], AL
0xb1150026 0000             ADD [EAX], AL
0xb1150028 0000             ADD [EAX], AL
0xb115002a 0000             ADD [EAX], AL
0xb115002c 0000             ADD [EAX], AL
0xb115002e 0000             ADD [EAX], AL
0xb1150030 0000             ADD [EAX], AL
0xb1150032 0000             ADD [EAX], AL
0xb1150034 0000             ADD [EAX], AL
0xb1150036 0000             ADD [EAX], AL
0xb1150038 0000             ADD [EAX], AL
0xb115003a 0000             ADD [EAX], AL
0xb115003c 0801             OR [ECX], AL
0xb115003e 0000             ADD [EAX], AL
```

Figure 15 – Malfofind output for test image

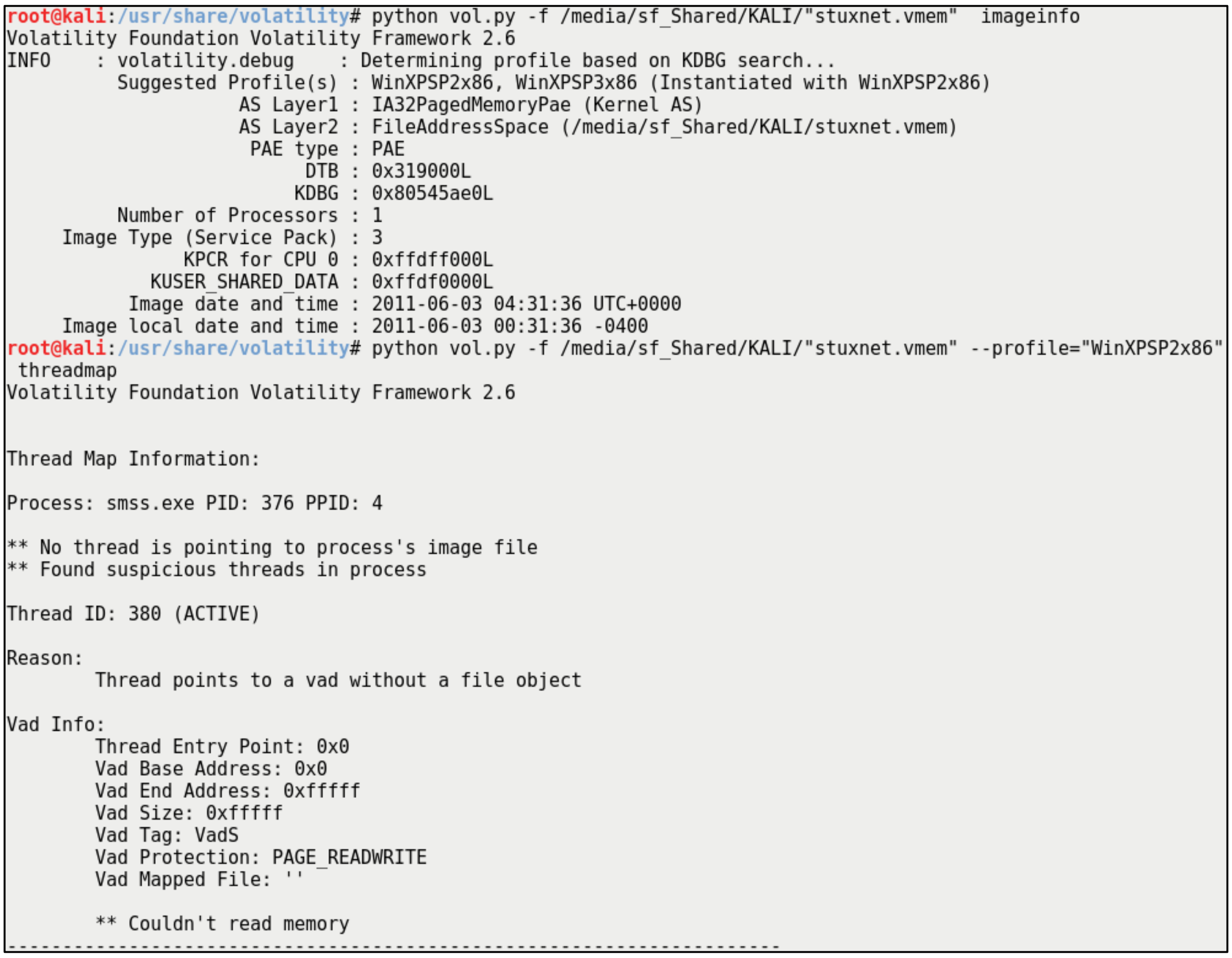
2.5.11 Threadmap plugin

Threadmap [47] is a plugin whose authors won the 2nd place in 2017 Volatility Plugin Contest for it [21]. The working team was based on John Leitch's Process Hollowing project [34] and created three different variations of the hollowing process. The first was not detected by malfind but was detected by hollowfind. The second variation was detected only by malfofind plugin and the third was detected only by threadmap plugin. Threadmap uses _ETHREAD structure information in order to relate a thread with a VAD. Then it compares the corresponding _EPROCESS structure to the VAD using the key points referenced above as well as some extra points needed for detecting the created variations.

This plugin was initially tested with the Memory Dump sample that the author used, meaning the file KSLSample.vmem, downloaded from [https://www.mediafire.com/file/jlmtbbinanuh6jr/KSLSample.rar] and the results were as it is expected, based on the Threadmap documentation [47]. In order the Threadmap to produce results and not any error messages, the parameter --profile="Win7SP1x64" was used for the specific memory image.

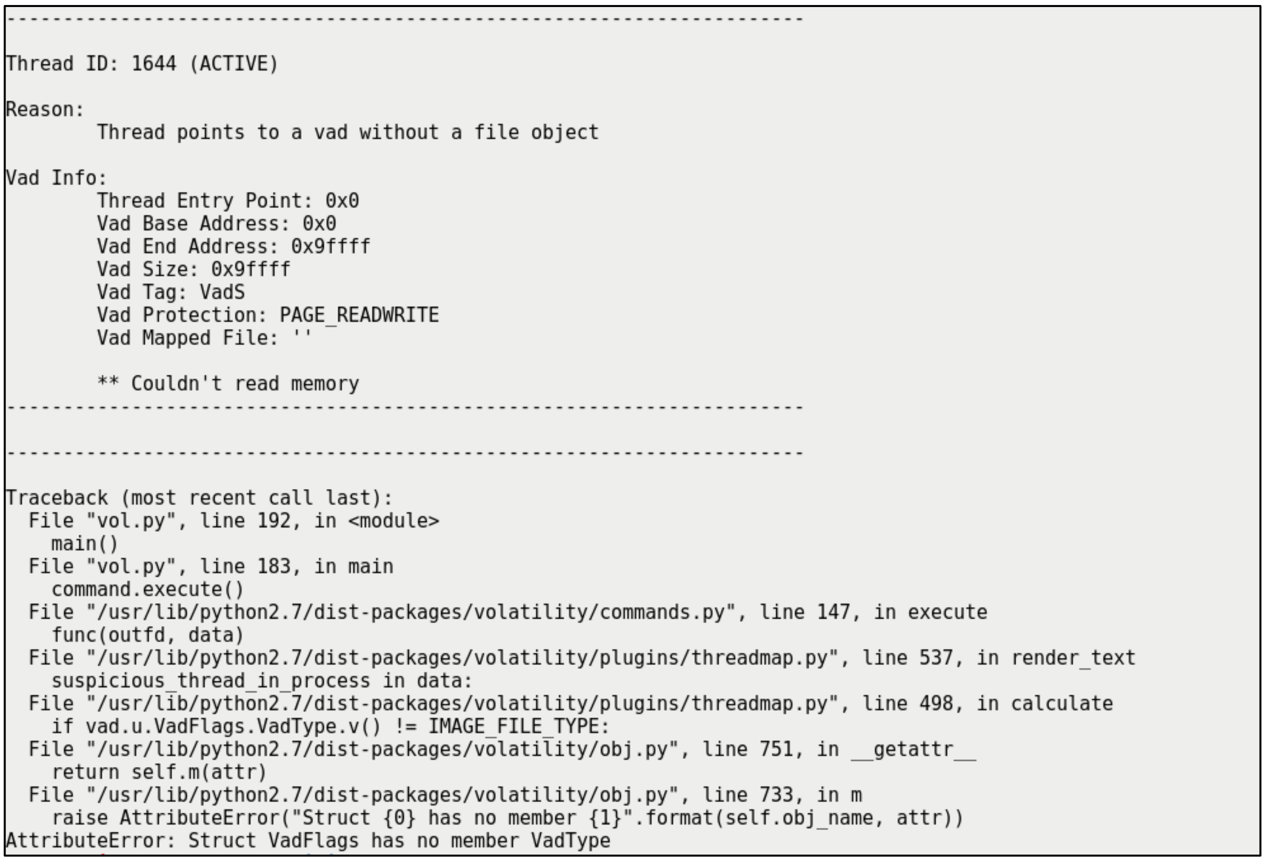
```
python vol.py – f mem.bin threadmap
```

For Stuxnet memory image, many profiles were tested. In all cases, the plugin produced many results and then errors occurred, as shown below:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"stuxnet.vmem"  imageinfo
Volatility Foundation Volatility Framework 2.6
INFO    : volatility.debug    : Determining profile based on KDBG search...
          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with WinXPSP2x86)
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/media/sf_Shared/KALI/stuxnet.vmem)
                      PAE type : PAE
                           DTB : 0x319000L
                          KDBG : 0x80545ae0L
          Number of Processors : 1
     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000L
             KUSER_SHARED_DATA : 0xffdf0000L
           Image date and time : 2011-06-03 04:31:36 UTC+0000
     Image local date and time : 2011-06-03 00:31:36 -0400
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"stuxnet.vmem" --profile="WinXPSP2x86"
 threadmap
Volatility Foundation Volatility Framework 2.6


Thread Map Information:

Process: smss.exe PID: 376 PPID: 4

** No thread is pointing to process's image file
** Found suspicious threads in process

Thread ID: 380 (ACTIVE)

Reason:
        Thread points to a vad without a file object

Vad Info:
        Thread Entry Point: 0x0
        Vad Base Address: 0x0
        Vad End Address: 0xfffff
        Vad Size: 0xfffff
        Vad Tag: VadS
        Vad Protection: PAGE_READWRITE
        Vad Mapped File: ''

        ** Couldn't read memory
----------------------------------------------------------------------
```

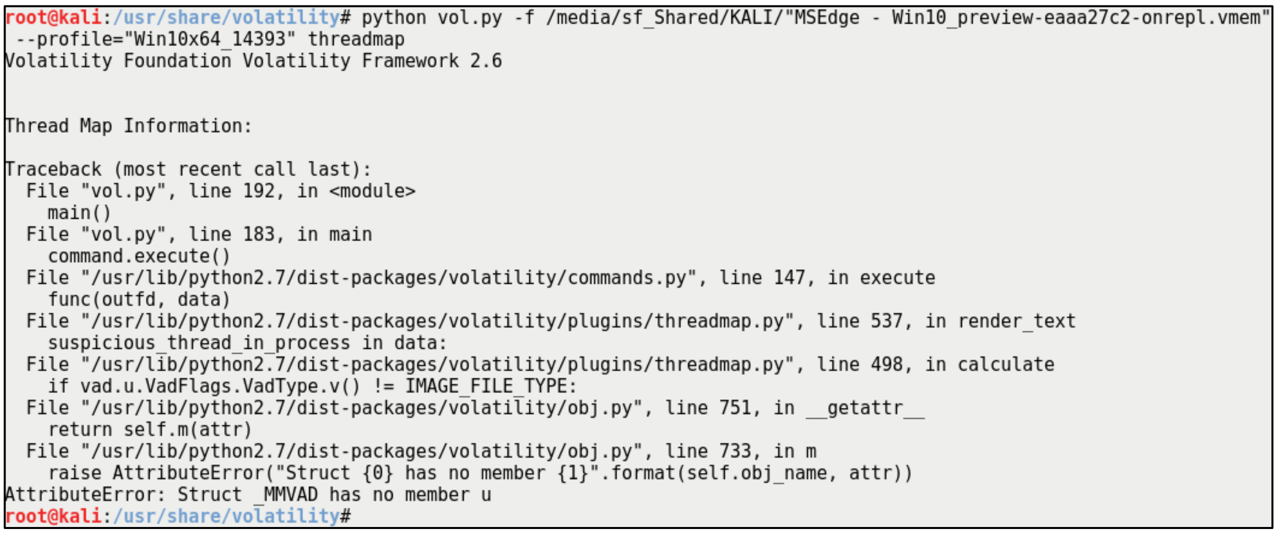
[snip]



```
----------------------------------------------------------------------

Thread ID: 1644 (ACTIVE)

Reason:
        Thread points to a vad without a file object

Vad Info:
        Thread Entry Point: 0x0
        Vad Base Address: 0x0
        Vad End Address: 0x9ffff
        Vad Size: 0x9ffff
        Vad Tag: VadS
        Vad Protection: PAGE_READWRITE
        Vad Mapped File: ''

        ** Couldn't read memory
----------------------------------------------------------------------

----------------------------------------------------------------------

Traceback (most recent call last):
  File "vol.py", line 192, in <module>
    main()
  File "vol.py", line 183, in main
    command.execute()
  File "/usr/lib/python2.7/dist-packages/volatility/commands.py", line 147, in execute
    func(outfd, data)
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 537, in render_text
    suspicious_thread_in_process in data:
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 498, in calculate
    if vad.u.VadFlags.VadType.v() != IMAGE_FILE_TYPE:
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 751, in __getattr__
    return self.m(attr)
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 733, in m
    raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct VadFlags has no member VadType
```

Figure 16 – ThreadMap output for Stuxnet memory image

Regarding the main testing image, ThreadMap produced nothing but errors:



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onrepl.vmem"
 --profile="Win10x64_14393" threadmap
Volatility Foundation Volatility Framework 2.6


Thread Map Information:

Traceback (most recent call last):
  File "vol.py", line 192, in <module>
    main()
  File "vol.py", line 183, in main
    command.execute()
  File "/usr/lib/python2.7/dist-packages/volatility/commands.py", line 147, in execute
    func(outfd, data)
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 537, in render_text
    suspicious_thread_in_process in data:
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 498, in calculate
    if vad.u.VadFlags.VadType.v() != IMAGE_FILE_TYPE:
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 751, in __getattr__
    return self.m(attr)
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 733, in m
    raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct _MMVAD has no member u
root@kali:/usr/share/volatility#
```

Figure 17 – ThreadMap output for test image

Probably ThreadMap works correctly for memory images of specific Windows versions, such as Win7SP1x64.

2.6 Summary

As seen in the previous paragraphs, the proposed methodology **detects the Hollow Process** Injection performed in a system and reveals the possible systems' memory anomalies. These steps were followed to detect the Injection in various memory images additionally to the main one demonstrated in the current section, as already mentioned. The good results of the methodology are verified in three created memory images as well as in two downloaded images.

The methodology requires that memory analysis is conducted using the Volatility Framework and more specific core Volatility's plugins, custom made scripts and plugins created for the specific purpose of Hollow Process Injection detection were used.

Since Process Hollowing is a commonly used Process injection technique, it has caused a great deal of concern to the analyst community. This is also clear from the number of dedicated plugins created and the number of related articles found on the World Wide Web.

3 DLL Injection

In this section, classic DLL (Dynamic Link Library) injection technique is presented and injections are performed in various systems resulting various test memory images. A new detection method is proposed, verified, implemented and applied on the test images. After that, the conclusions are extracted and presented.

Classic DLL injection, or remote DLL injection, is an injection technique that loads a malicious DLL inside the context of a running legitimate (target) process. The remote process is manipulated by functions such as CreateRemoteThread and its memory content is altered. Once the compromised process loads the malicious DLL, the OS (Operating System) automatically calls the DLL's DllMain function, which is defined by the author of the DLL. This function contains the malicious code and has as much access to the system as the process in which it is running [48][3]. The prerequisite is the existence of the malicious DLL in disk. It is considered as the easiest and simpler injection technique.

3.1 How DLL Injection works

The steps that the **malicious process**, or otherwise called the injector performs in this technique are:

1. Enables debug privilege (SE_DEBUG_PRIVILEGE) that gives it the right to read and write in other process' memory as if it were a debugger.

2. Finds the target process' ID using its name.
   This is usually done by creating a snapshot of all processes and searching in it for the specific process. CreateToolhelp32Snapshot, Process32First and Process32Next APIs (Application Program Interfaces) are used for this purpose.

3. Opens the target process with the desired access rights and gets a handle to it.
   This is done by calling OpenProcess function with at least PROCESS_CREATE_THREAD, PROCESS_VM_OPERATION and PROCESS_VM_WRITE access rights.
   If the caller has enabled the Debug Privilege, the requested access is granted regardless of the contents of the security descriptor.

4. Gets the full path and name of the DLL, which already exists on the disk.

5. Allocates a new memory region within the virtual address space of the target process of size as the length of the malicious DLL full name (including its path) and gets the address this memory region. VirtualAllocEx with PAGE_READWRITE protection is used.

6. Writes the DLLs' full name into the newly allocated memory using WriteProcessMemory at the address retrieved in the previous step.

7. Creates a new thread within the context of the legitimate process that executes the LoadLibrary function using as parameter the written DLL full name. This practically loads the malicious DLL and the injection is completed.
This step is performed as follows: First, a handle for kernel32.dll is fetched and the address of LoadLibrary function is retrieved from it. Then the thread is created by using APIs such as CreateRemoteThread, NtCreateThreadEx, or RtlCreateUserThread. The important parameters for these APIs are the handle on the target process (retrieved in step 3), the pointer to the address of LoadLibrary function and the pointer to the address of the memory region that holds the Dll name. This means that LoadLibrary loads the malicious DLL. The thread runs in the virtual address space of the target process immediately after its creation.

8. Cleans up by freeing the allocated memory and closing the handle on the target process and created thread, using VirtualFree and CloseHandle functions respectively.
Note: "The thread object remains in the system until the thread has terminated and all handles to it are closed through a call to CloseHandle" [58].

[6][22][23][49]

A malware that uses this technique is for example Poison Ivy.

The following figure presents the described steps.

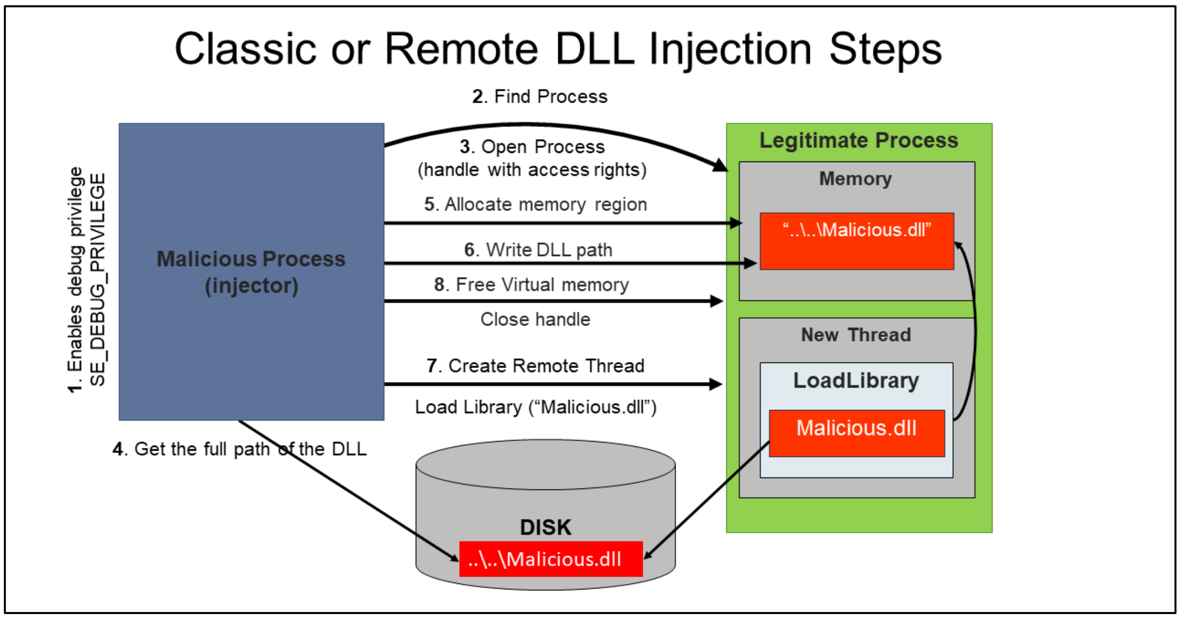

Classic or Remote DLL Injection Steps
1. Enables debug privilege SE_DEBUG_PRIVILEGE
Malicious Process (injector)
2. Find Process
3. Open Process (handle with access rights)
5. Allocate memory region
6. Write DLL path
8. Free Virtual memory
Close handle
7. Create Remote Thread
Load Library ("Malicious.dll")
4. Get the full path of the DLL
Legitimate Process
Memory
"..\..\Malicious.dll"
New Thread
LoadLibrary
Malicious.dll
DISK
..\..\Malicious.dll

Figure 18 – Remote DLL injection steps

Most legitimate processes should not need to use APIs such as **CreateRemoteThread**, so it is characterized as a very suspicious API and is detected by many security products which also may detect the malicious DLL on the disk [22]. These are the reasons this technique is considered as simple and is not frequently used by the attackers who are trying to evade defenses.

3.2 DLL Injection Detection: A different approach

The DLL Injection described in previous section uses LoadLibrary function to load the malicious DLL in the process, which is the perfectly normal way, so it cannot be distinguished from any other legitimately loaded DLL. The malicious DLL appears through Volatility's dlllist plugin and can be detected by its unexpected name paths. Anomalies in memory can be detected using Volatility in case a) the injected DLL is hidden (by unlinking its _LDR_DATA_TABLE_ENTRY from one or more of the ordered lists) using ldrmodules and b) the injected DLL is packed, and the unpacking procedure copies the decompressed code to a new memory region which can be located by malfind plugin. In any other case, an analyst must use typical analysis such as context analysis, Yara scan, etc. [6]

This thesis assumes that the **DLL is not hidden** and does not focus on standard DLL injection detection techniques (such as memory permissions, priority changes, analyzing the Process Environment Block), nor it analyzes the DLL itself for signatures for example. A different approach is used which focuses on the fact that **LoadLibrary** and **CreateRemoteThread** functions are used and on **time sequence of events** which is presented in the following timeline, along with the key points explained below.

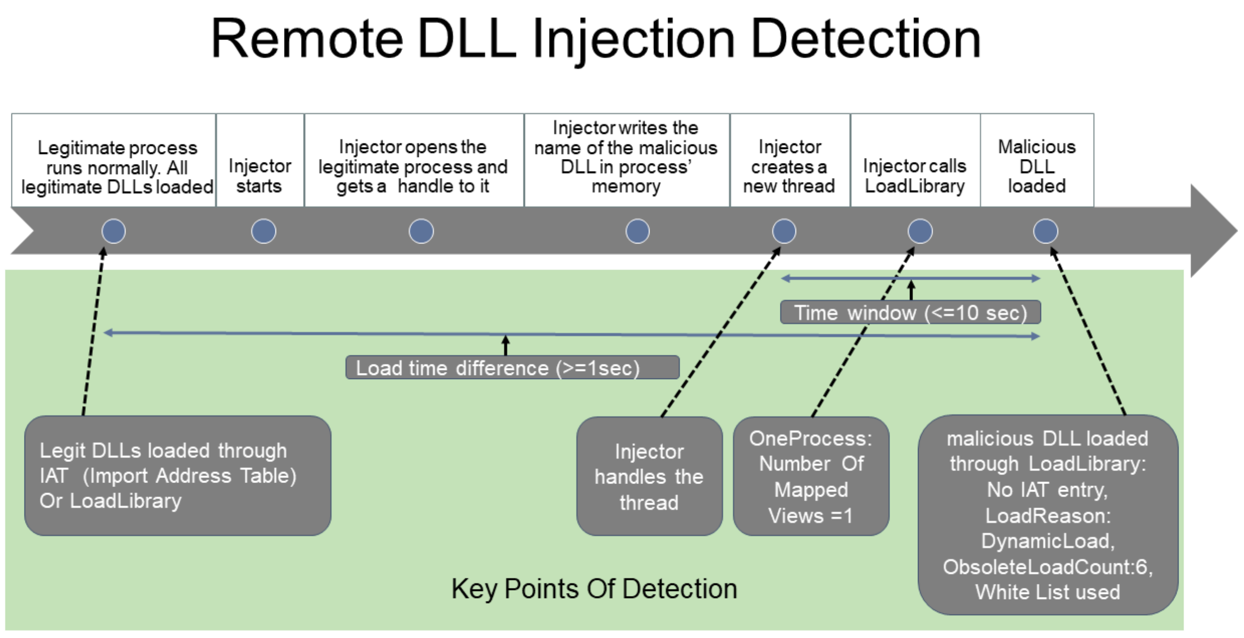

Remote DLL Injection Detection
Legitimate process runs normally. All legitimate DLLs loaded
Injector starts
Injector opens the legitimate process and gets a handle to it
Injector writes the name of the malicious DLL in process' memory
Injector creates a new thread
Injector calls LoadLibrary
Malicious DLL loaded
Time window (<=10 sec)
Load time difference (>=1sec)
Legit DLLs loaded through IAT (Import Address Table) Or LoadLibrary
Injector handles the thread
OneProcess: Number Of Mapped Views =1
malicious DLL loaded through LoadLibrary: No IAT entry, LoadReason: DynamicLoad, ObsoleteLoadCount:6, White List used
Key Points Of Detection

Figure 19 – Key points of DLL Injection Detection

The **key points** in which the idea of detection is based are presented below together with comments that help to understand the produced code:

1. Since the injected DLL is not commonly used, the process' corresponding **VAD** (Virtual Address Descriptor) node is marked with the value 1 in its **NumberOfMappedViews** and NumberOfUserReferences fields. This applies to the first time the malware is executed. In case the same DLL is injected in different processes, this does not apply. This is the reason why in the final

script, it is chosen a parameter (OneProcess) to be used that determines whether the above condition should be checked or not.

2. The injected DLL is explicitly loaded at run-time **using LoadLibrary** (or LoadLibraryEx) API function [50], meaning that:
   - there isn't any corresponding entry in the processes' IAT (Import Address Table) and
   - DLLs' LoadReason attribute in corresponding windows data structure is LoadReasonDynamicLoad or its ObsoleteLoadCount is 6.
     For further details, see sections Determine whether a DLL file is loaded via LoadLibrary using its attributes and Scan processes' memory for IAT entries.

3. **The injected DLL has a load time difference** in relation with the last DLL that the remote process loads for legitimate reasons. This is due to the delay of the malware execution in relation to the legitimate process's creation time. Because this time difference cannot be estimated, inside the context of this work, it is considered that the injected DLL may be loaded just one second after the last legitimate one.

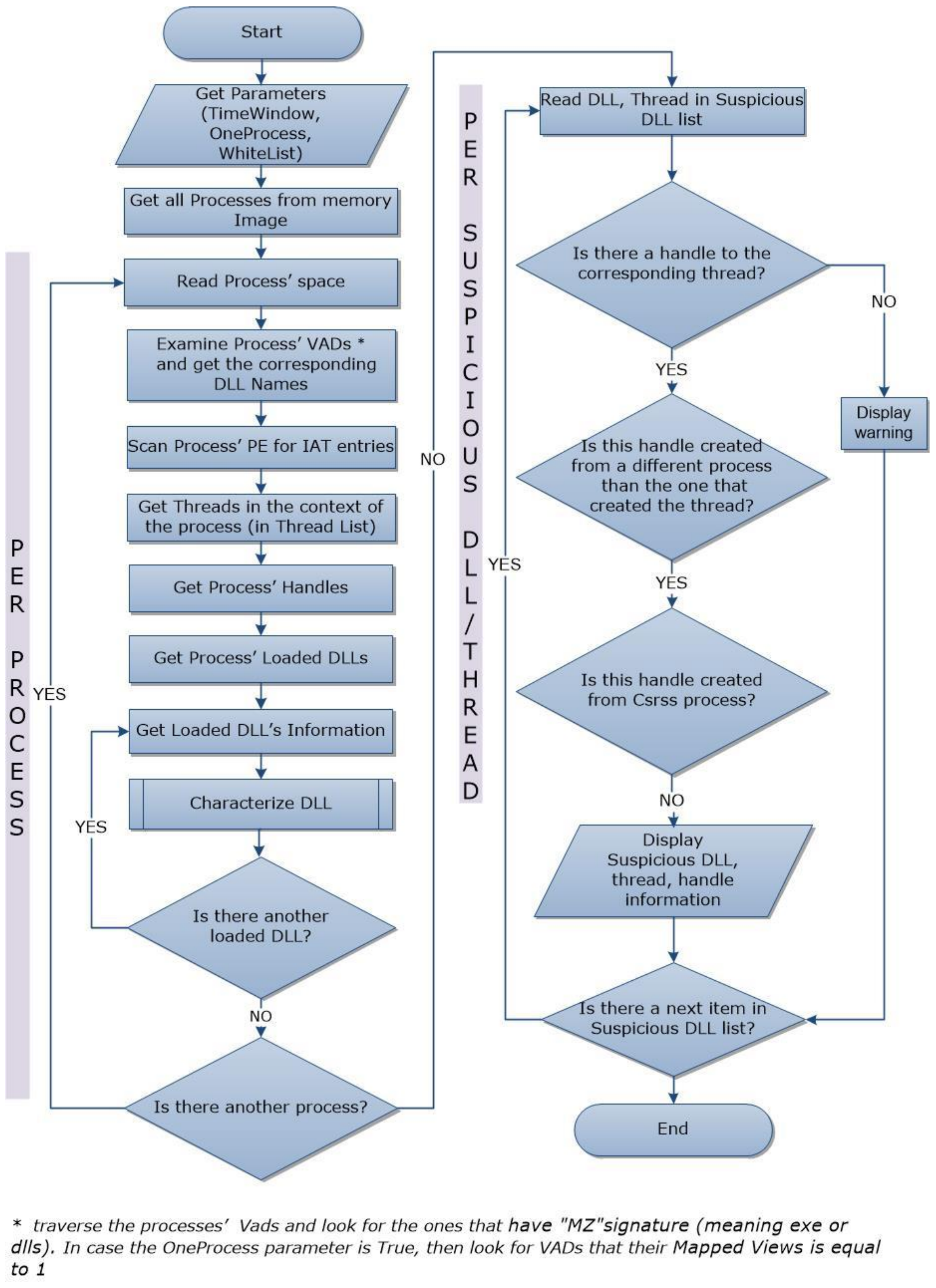
4. The thread that executes the LoadLibrary function is created by the malware just before the malicious DLL loading. The TimeWindow parameter is used to express the **time interval between malicious thread creation and DLL loading**. Usually this is done in the same second, but the default value for TimeWindow is 10 sec, so that any possibly suspicious thread is not excluded. From all the legitimate's process threads, the interesting ones are those that are not terminated and are created between 10 seconds before the DLL loading until the DLL loading. This means that one suspicious DLL may be correlated to more than one possibly responsible threads that caused its loading.

5. This thread is created by the malware using the handle to the legitimate process. So, from all processes' handles of type THREAD, we are interested in the ones that refer to the specific couple of process-thread id and are **handled from the legitimate process**. Handles created by csrss.exe, which is involved in the creation of every process and thread [6] are excluded.

6. To reduce the false positive results, there is the option to store in a list (WhiteList) the full path of DLLs that are considered harmless. If a DLL from the **White List** is characterized by the code as suspicious, a corresponding warning message is displayed.

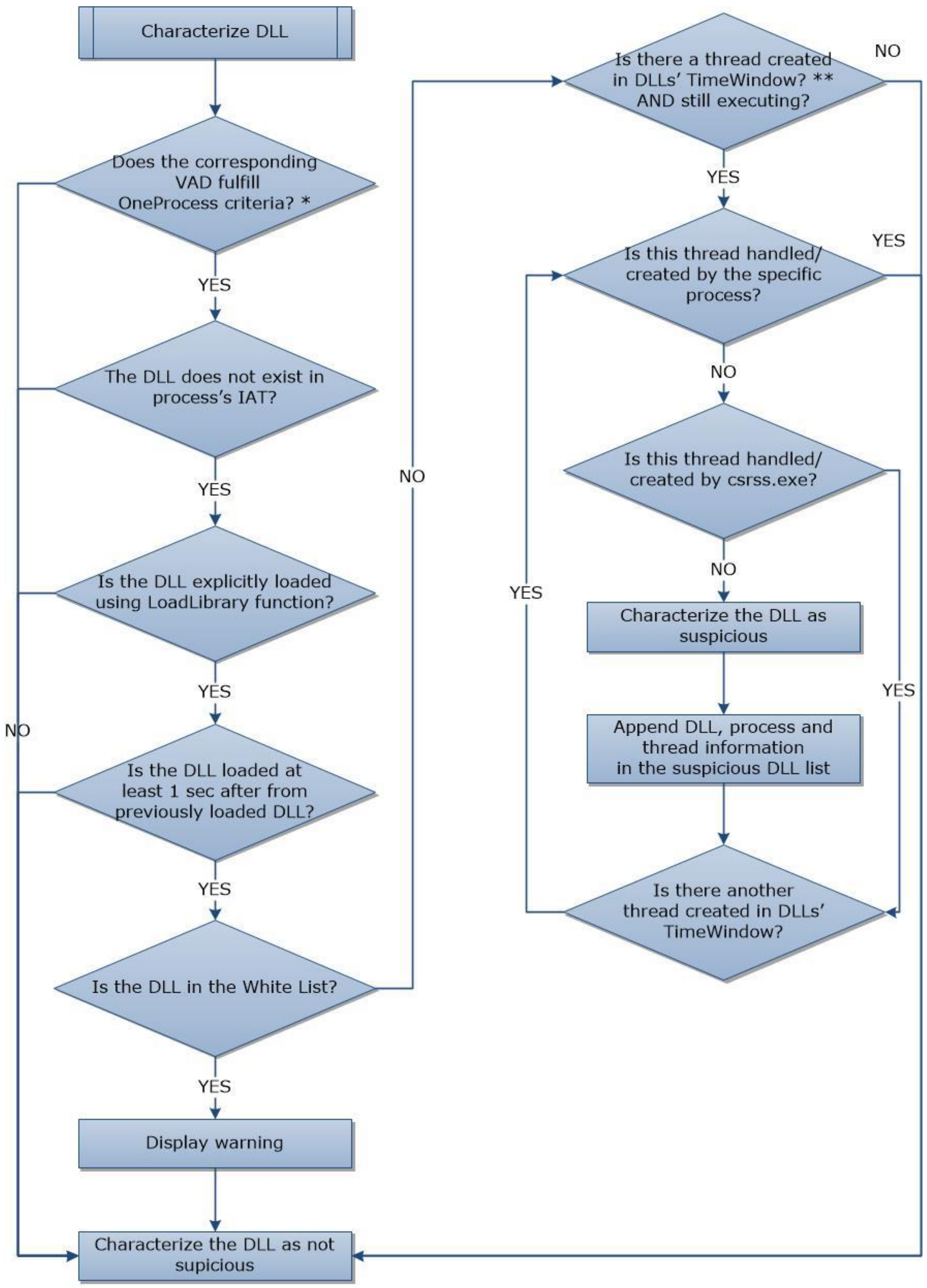
3.2.1 Detection Flowchart

The idea of the detection method, at a high level (without much detail) is presented in the following flowchart. The key points mentioned above are included in Characterize DLL subprocess.



Start
Get Parameters (TimeWindow, OneProcess, WhiteList)
Get all Processes from memory Image
Read Process' space
Examine Process' VADs * and get the corresponding DLL Names
Scan Process' PE for IAT entries
Get Threads in the context of the process (in Thread List)
Get Process' Handles
Get Process' Loaded DLLs
Get Loaded DLL's Information
Characterize DLL
Is there another loaded DLL?
YES
NO
Is there another process?
YES
NO
PER PROCESS
PER SUSPICIOUS DLL/THREAD
Read DLL, Thread in Suspicious DLL list
Is there a handle to the corresponding thread?
NO
Display warning
YES
Is this handle created from a different process than the one that created the thread?
YES
Is this handle created from Csrss process?
NO
Display Suspicious DLL, thread, handle information
Is there a next item in Suspicious DLL list?
YES
End
* traverse the processes' Vads and look for the ones that have "MZ"signature (meaning exe or dlls). In case the OneProcess parameter is True, then look for VADs that their Mapped Views is equal to 1

Figure 20 - DLL Injection Detection Flowchart


Characterize DLL
Does the corresponding VAD fulfill OneProcess criteria? *
YES
The DLL does not exist in process's IAT?
YES
Is the DLL explicitly loaded using LoadLibrary function?
YES
Is the DLL loaded at least 1 sec after from previously loaded DLL?
YES
Is the DLL in the White List?
YES
Display warning
Characterize the DLL as not supicious
NO
NO
Is there a thread created in DLLs' TimeWindow? ** AND still executing?
NO
YES
Is this thread handled/ created by the specific process?
YES
NO
Is this thread handled/ created by csrss.exe?
NO
Characterize the DLL as suspicious
Append DLL, process and thread information in the suspicious DLL list
Is there another thread created in DLLs' TimeWindow?
YES
YES

*** refers to thread that are created TimeWindow (default 10) seconds before and up to the DLL loading time*

Figure 21 - Characterize DLL subprocess flowchart

3.2.2 Detection Idea Verification

Initially, the concept described above was confirmed by calling a script inside volshell Volatilitys' plugin environment that enumerates the VADs with the specific characteristics and manually combining its results with the following Volatility's plugins: a) timeliner which creates a timeline from various artifacts in memory and b) handles which displays the open handles in a process [18]. These plugins are time consuming and produce more information than the needed.

The IAT entries were also manually found, creating the processes' exe with procdump plugin and using readpe command. Example of this procedure is presented in a following section.

After confirmation, **all the steps of the detection flowchart were combined in a single python script**, called FindDLLInj.py, which is called from Volatility's volshell plugin.

3.3 Detection Code - FindDLLInj.py

The parameters that the script uses are:

OneProcess

Boolean that determines the characteristics of VADs selected while traversing processes' memory looking for DLL names. Consequently, it determines the characteristics of corresponding DLL. It is used in Characterize DLL step.
Possible values:
True : we are suspicious only of DLLs that their NumberOfMappedViews and NumberOfUserReferences are equal to 1.
False : we are suspicious of any DLL, regardless their NumberOfMappedViews and NumberOfUserReferences values.

TimeWindow

Integer that represents the time interval (in sec, before DLL load time) during which it is being searched for starting threads. This is used to correlate the DLLs with threads that possibly loaded them. The default value is 10 seconds.

WhiteList

A list of DLLs' full names that are considered as not suspicious.

The output file is a Tab delimited text file named SuspectedDlls.txt which includes:

- an introduction line that contains information about the memory image and parameters used, e.g. Time of execution, memory image on which the script is executing, TimeWindow and OneProcess parameters values,
- a columns header line and
- output lines which consist of the following fields:

Field Name	Field Explanation
Pid	Process ID
Description	In case of DLL injection detection: “Suspicious process” Otherwise, warnings such as: “No Imported Dll found”, “No Loaded Dlls found”, “No handle found”, “Dll in WhiteList”
ImageFileName	Process Name
Dll	DLL full name
DllLoadTime	Date and Time that the DLL was loaded (UTC+0000)
TimeDifference from previously loaded Dll in sec	Time Difference from the previously loaded DLL inside the context of the specific process (in sec)
ThreadPid	ID of the process under which the injection thread is created (injected process, same as Pid)
ThreadTid	ID of the injection thread created by the malicious process, inside the context of the Pid
ThreadLoadTime	Date and Time that the Thread begun executing (UTC+0000)
ThreadExitTime	Thread Exit Date and Time or ‘ ‘
ThreadHandlePid	ID of the process that handles the injection Thread (injector process ID)

The complete script that implements the steps described above is cited in FindDLLInj Script section and alternatively can also be found in the repository: https://github.com/Soterball/DLLInjectionDetection, together with some produced output files.

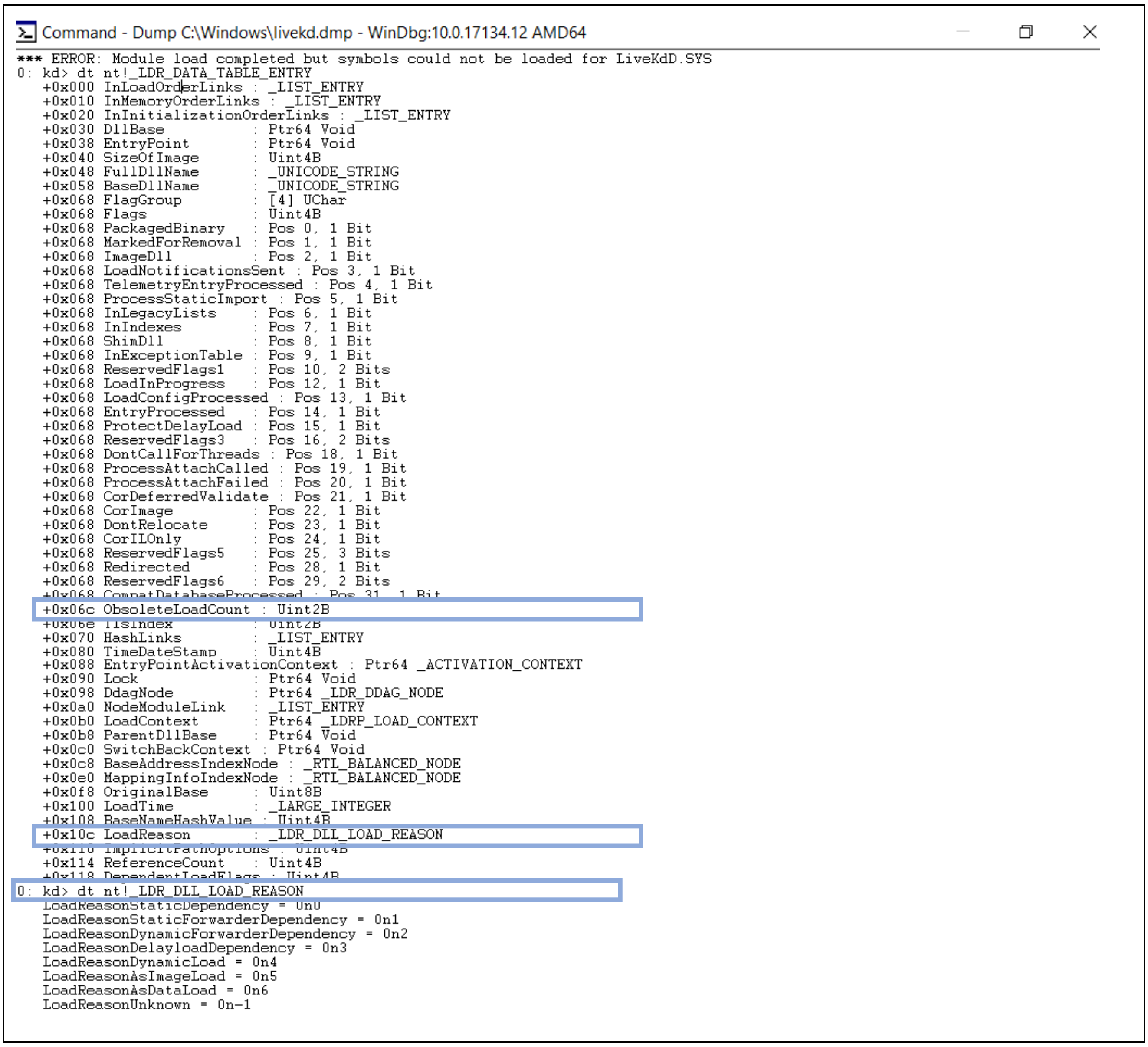
Note: Warning messages are displayed when no loaded DLLs are found and in case the IAT table cannot be reconstructed due to the high likelihood that one or more pages in the PE header or IAT are not memory resident (paged). Also, a warning is displayed when no handle that correspond to the suspicious DLL/thread pair is found.

3.3.1 Explanations relating the script

Before looking at the entire script, it is necessary to give some explanations, to make clear how the above-mentioned key points of detection are implemented through the code.

Determining whether a DLL file is loaded via LoadLibrary using its attributes

In windows operating system, the DLLs are represented though _LDR_DATA_TABLE_ENTRY data structure which is analyzed below, using Sysinternals Livekd on the testing environment (Win10 Build 14393) to debug the Windows kernel [59]. The _LDR_LOAD_REASON structure, representing the LoadReason described above is also shown. The command dt("_LDR_DATA_TABLE_ENTRY") inside Volatility's volshell environment gives the same output.


Command - Dump C:\Windows\livekd.dmp - WinDbg:10.0.17134.12 AMD64

```
*** ERROR: Module load completed but symbols could not be loaded for LiveKdD.SYS
0: kd> dt nt!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x010 InMemoryOrderLinks : _LIST_ENTRY
   +0x020 InInitializationOrderLinks : _LIST_ENTRY
   +0x030 DllBase          : Ptr64 Void
   +0x038 EntryPoint       : Ptr64 Void
   +0x040 SizeOfImage      : Uint4B
   +0x048 FullDllName      : _UNICODE_STRING
   +0x058 BaseDllName      : _UNICODE_STRING
   +0x068 FlagGroup        : [4] UChar
   +0x068 Flags            : Uint4B
   +0x068 PackagedBinary   : Pos 0, 1 Bit
   +0x068 MarkedForRemoval : Pos 1, 1 Bit
   +0x068 ImageDll         : Pos 2, 1 Bit
   +0x068 LoadNotificationsSent : Pos 3, 1 Bit
   +0x068 TelemetryEntryProcessed : Pos 4, 1 Bit
   +0x068 ProcessStaticImport : Pos 5, 1 Bit
   +0x068 InLegacyLists    : Pos 6, 1 Bit
   +0x068 InIndexes        : Pos 7, 1 Bit
   +0x068 ShimDll          : Pos 8, 1 Bit
   +0x068 InExceptionTable : Pos 9, 1 Bit
   +0x068 ReservedFlags1   : Pos 10, 2 Bits
   +0x068 LoadInProgress   : Pos 12, 1 Bit
   +0x068 LoadConfigProcessed : Pos 13, 1 Bit
   +0x068 EntryProcessed   : Pos 14, 1 Bit
   +0x068 ProtectDelayLoad : Pos 15, 1 Bit
   +0x068 ReservedFlags3   : Pos 16, 2 Bits
   +0x068 DontCallForThreads : Pos 18, 1 Bit
   +0x068 ProcessAttachCalled : Pos 19, 1 Bit
   +0x068 ProcessAttachFailed : Pos 20, 1 Bit
   +0x068 CorDeferredValidate : Pos 21, 1 Bit
   +0x068 CorImage         : Pos 22, 1 Bit
   +0x068 DontRelocate     : Pos 23, 1 Bit
   +0x068 CorILOnly        : Pos 24, 1 Bit
   +0x068 ReservedFlags5   : Pos 25, 3 Bits
   +0x068 Redirected       : Pos 28, 1 Bit
   +0x068 ReservedFlags6   : Pos 29, 2 Bits
   +0x068 CompatDatabaseProcessed : Pos 31, 1 Bit
   +0x06c ObsoleteLoadCount : Uint2B
   +0x06e TlsIndex         : Uint2B
   +0x070 HashLinks        : _LIST_ENTRY
   +0x080 TimeDateStamp    : Uint4B
   +0x088 EntryPointActivationContext : Ptr64 _ACTIVATION_CONTEXT
   +0x090 Lock             : Ptr64 Void
   +0x098 DdagNode         : Ptr64 _LDR_DDAG_NODE
   +0x0a0 NodeModuleLink   : _LIST_ENTRY
   +0x0b0 LoadContext      : Ptr64 _LDRP_LOAD_CONTEXT
   +0x0b8 ParentDllBase    : Ptr64 Void
   +0x0c0 SwitchBackContext : Ptr64 Void
   +0x0c8 BaseAddressIndexNode : _RTL_BALANCED_NODE
   +0x0e0 MappingInfoIndexNode : _RTL_BALANCED_NODE
   +0x0f8 OriginalBase     : Uint8B
   +0x100 LoadTime         : _LARGE_INTEGER
   +0x108 BaseNameHashValue : Uint4B
   +0x10c LoadReason       : _LDR_DLL_LOAD_REASON
   +0x110 ImplicitPathOptions : Uint4B
   +0x114 ReferenceCount   : Uint4B
   +0x118 DependentLoadFlags : Uint4B
0: kd> dt nt!_LDR_DLL_LOAD_REASON
   LoadReasonStaticDependency = 0n0
   LoadReasonStaticForwarderDependency = 0n1
   LoadReasonDynamicForwarderDependency = 0n2
   LoadReasonDelayloadDependency = 0n3
   LoadReasonDynamicLoad = 0n4
   LoadReasonAsImageLoad = 0n5
   LoadReasonAsDataLoad = 0n6
   LoadReasonUnknown = 0n-1
```

Figure 22 - _LDR_DATA_TABLE_ENTRY and _LDR_LOAD_REASON sturctures

In [6] is mentioned that when the **LoadCount** field of LDR_DATA_TABLE_ENTRY is 0xffff (or -1 because it is a short integer), it means that the DLL is loaded because this was specified in the IAT. This statement is tested and confirmed in a memory image of older windows version called Stuxnet.vmem, downloaded from [21] (In Blog Archive, 2016, August, post: Automating Detection of Known Malware through Memory Forensics). In windows 8 and later, the LoadCount is replaced with **ObsoleteLoadCount** [60].

To find out how ObsoleteLoadCount is manipulated by Windows, since it is not sufficiently documented by Microsoft, a list of all loaded modules in the testing memory images with their corresponding attributes is created. This list is compared with the IAT reproduction output and some paradoxes were found: a) the district values for ObsoleteLoadCount are 0xffff (or 65535 or -1) and 6, b) there are DLLs that have corresponding entries in the IAT and their ObsoleteLoadCount is 6 (and not 0xffff). The situation is a bit confusing, but it becomes clearer when ObsoleteLoadCount is associated with LoadReason field of _LDR_DATA_TABLE_ENTRY structure. It is safe to conclude that the **DLLs that are loaded using LoadLibrary have either ObsoleteLoadCount equal to 6 OR LoadReason equal to LoadReasonDynamicLoad**. The OR logical operand is used in a pessimistic way so that more DLLs are going to be checked. LoadReasonDynamicLoad corresponds to the value of 4, as shown in the above figure, which is confirmed in file \volatility\plugins\overlays\windows\ win10_x64_vtypes.py and also stated in [60].

Note: The dlllist Volatility plugin still shows the LoadCount (with the default value of 0) and not the contents of ObsoleteLoadCount, even when the profile is determined as "Win10x64_14393".

The code used for this test is (inside Volatility's volshell plugin) is:

```
for proc_id in getprocs():
        p_id= proc_id.UniqueProcessId
        cc(pid=p_id)
        process=proc()

        mods = process.get_load_modules()

        for mod in mods:
                print  p_id, mod.FullDLLName, mod.BaseDLLName, mod.LoadCount,
                        mod.ReferenceCount, mod.ParentDLLBase, mod.ObsoleteLoadCount,
                        mod.LoadReason ,mod.ParentDLLBase, mod.DLLBase
```

Scan processes' memory for IAT entries

When an executable is first loaded in memory, the Windows loader, amongst others, is responsible for loading the needed DLLs. This information is stored in the IAT (Import Address Table), which is part of the PE structure. On the other hand, the DLLs loaded using LoadLibrary function are explicitly loaded at run-time [61] and there isn't any corresponding entry in the IAT. So, a useful step to the detection process is the **reproduction of the IAT**. Loaded DLLs stored in it are not considered suspicious, in contrast with the other loaded DLLs.

Although the PE (Portable Executable) file format is outside the scope of this thesis, it follows a visual representation of PE file structure at an abstract level and only to assist the understanding the code used for retrieving the names of the imported DLLs. The cells with black background refer to the structure names and the gray cells to their fields. The blue dashed lines show how the respective fields are analyzed and the blue arrows represent pointers. All these data structures are defined in WINNT.H [51].

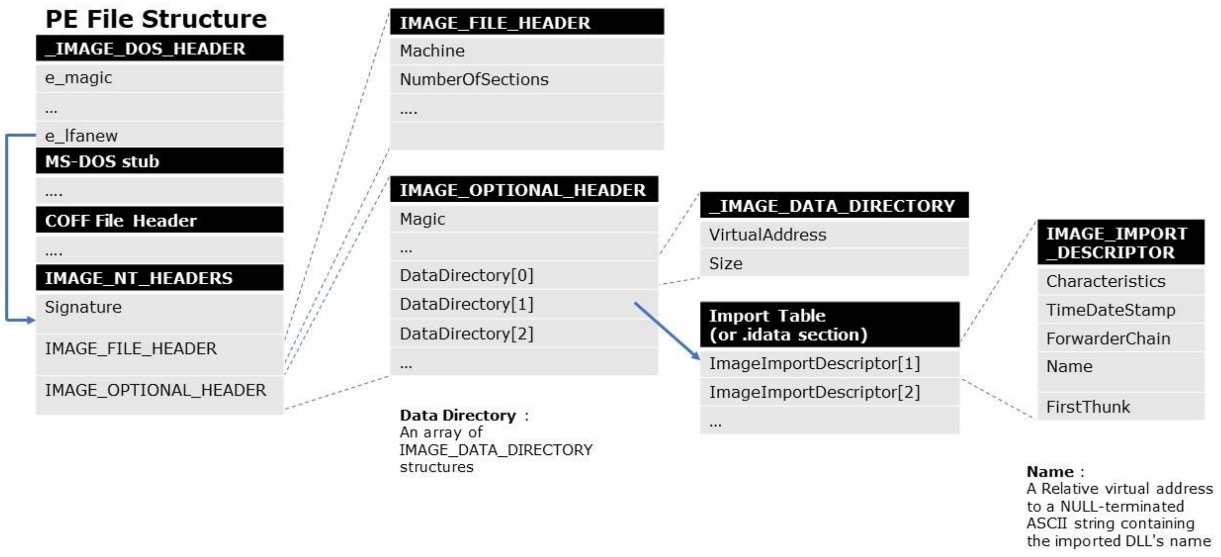


Figure 23 - Diagram showing PE File structure and IAT

As shown in the above diagram, the IAT (Import Address Table) is represented as an array of IMAGE_IMPORT_DESCRIPTORs. The last element of the array is indicated by an IMAGE_IMPORT_DESCRIPTOR that has fields filled with NULLs. The import table (or .idata section) of the PE begins with this array. The import table is the second entry of the DataDirectory table which is an array of IMAGE_DATA_DIRECTORY structures that resides inside IMAGE_OPTIONAL_HEADER, field of IMAGE_NT_HEADERS Header [27][28][51].

Going in the opposite direction to that described in the previous paragraph, **the procedure of scanning each process's memory for IAT entries** using the Volatility's volshell plugin, has the following steps:

- Create a _IMAGE_DOS_HEADER object using as offset the process' base address
- Get the NT header, using the get_nt_header function (which is the same as following the e_lfanew member to it)
- Find DataDirectory[1] inside OptionalHeader
- Scan the IAT and create _IMAGE_IMPORT_DESCRIPTOR objects until an empty one is found. The name field is the address of an ASCII string that contains the name of the DLL and is relative to the image base. Read the DLL Name and append it in the list of Imported DLLs [6][52]

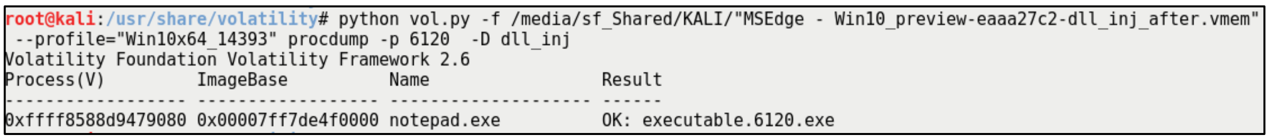
The code that reproduces the IAT table follows. Part of it is taken from file plugins/overlays/windows/pe_vtypes.py, imports() function of class _LDR_DATA_TABLE_ENTRY.

```
# "Scan" PE to reconstruct the IAT table - 1st method
        DLLsFromImport=[]
        dos_header = obj.Object("_IMAGE_DOS_HEADER",offset =               \
                     process.Peb.ImageBaseAddress,vm = process_space)
        nt_header = dos_header.get_nt_header()
        data_dir = nt_header.OptionalHeader.DataDirectory[1]

        i = 0

        #desc_size = self.obj_vm.profile.get_obj_size('_IMAGE_IMPORT_DESCRIPTOR')
        desc_size=20

        while 1:
                desc = obj.Object('_IMAGE_IMPORT_DESCRIPTOR',
                         vm = process_space,
                         offset = process.Peb.ImageBaseAddress +
                                data_dir.VirtualAddress
                                + (i * desc_size), parent = self)

                # Stop if the IID is paged or all zeros
                if desc == None or desc.is_list_end():
                   break

                # Stop if the IID contains invalid fields
                if not desc.valid(nt_header):
                   break

                DLLName=obj.Object("String", offset =
                                desc.Name+process.Peb.ImageBaseAddress,
                                vm = process_space, length = 128)
                DLLsFromImport.append(str(DLLName).lower())

                i += 1
```

Confirmation of the IAT reproduction

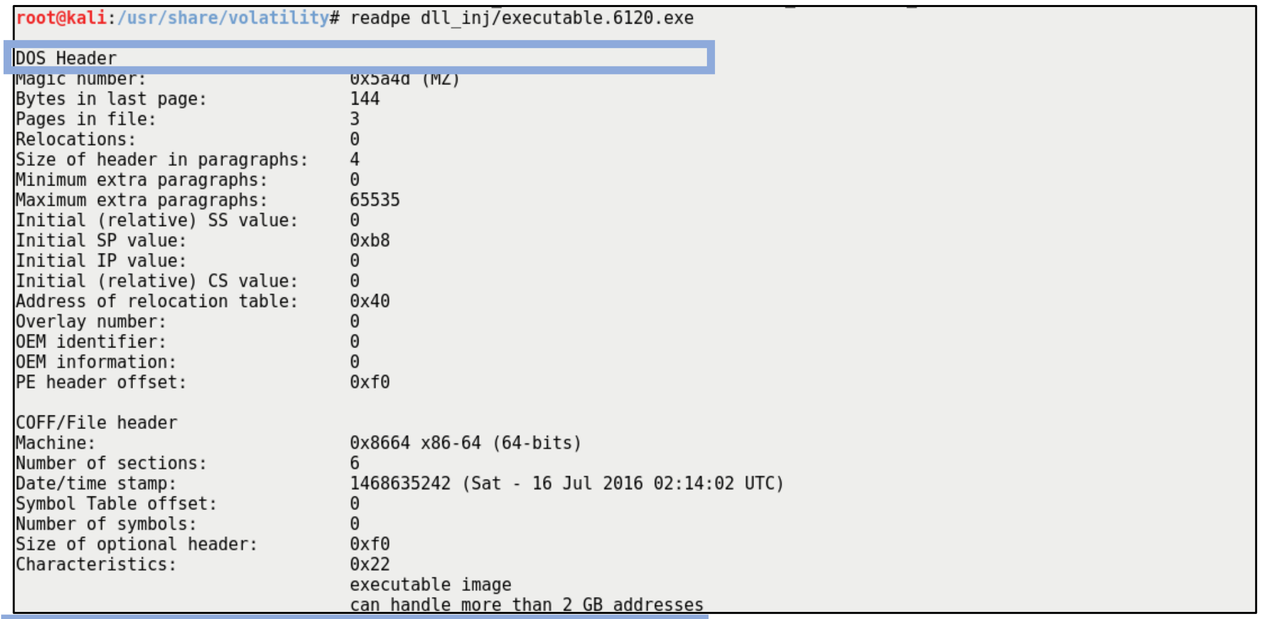
The above code has been tested in various processes of various images. The results are identical to those of using objdump and readpe commands to the corresponding .exe files, created by procdump volatility. The following screenshots show the results for process 6120 (notepad) on one of the testing memory images.

At first, the process 6120 is dumped and the corresponding .exe is created using procdump plugin.

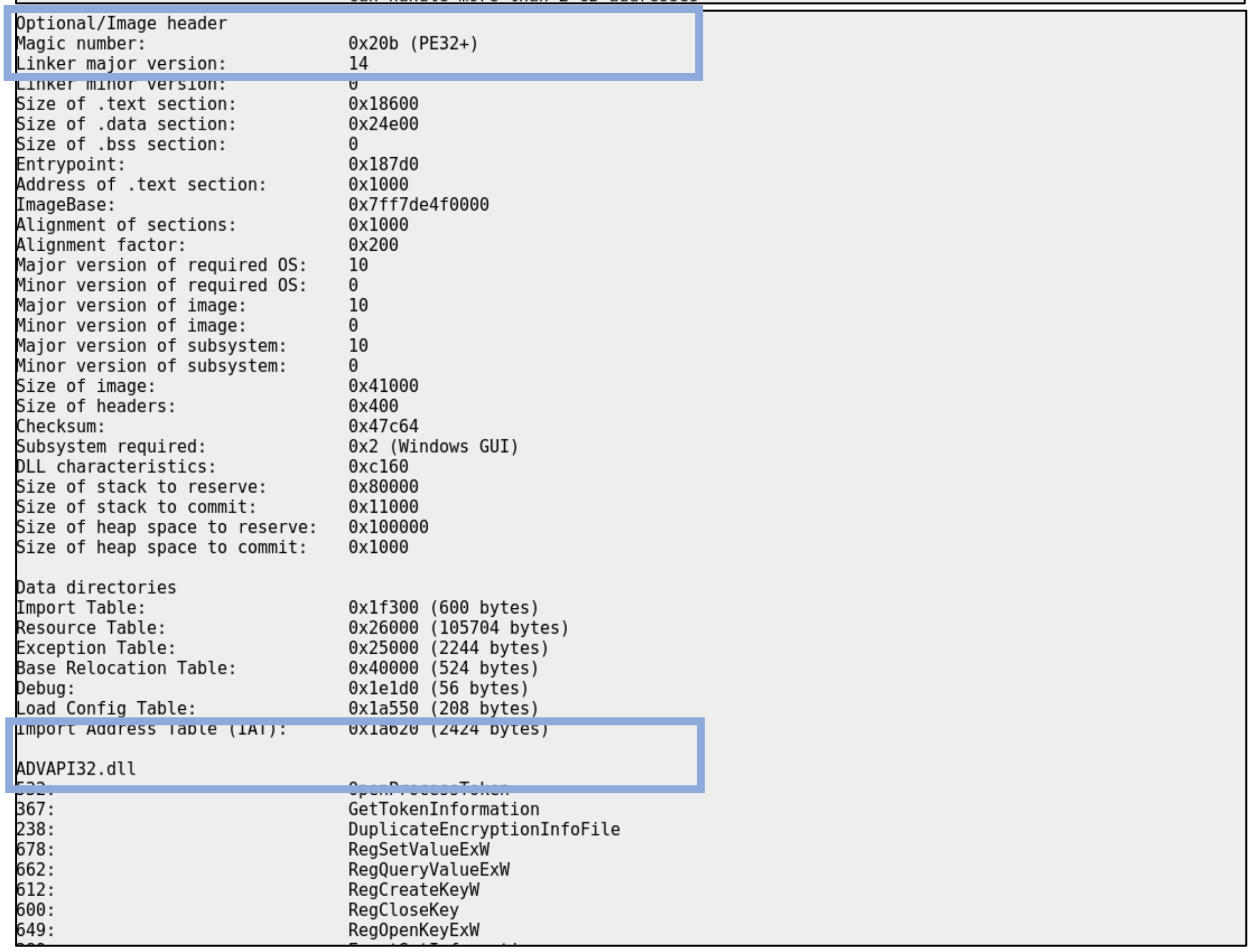


```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-dll_inj_after.vmem"
 --profile="Win10x64_14393" procdump -p 6120  -D dll_inj
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffff8588d9479080 0x00007ff7de4f0000 notepad.exe          OK: executable.6120.exe
```

Using the readpe command, the PE structure as described above, is presented.

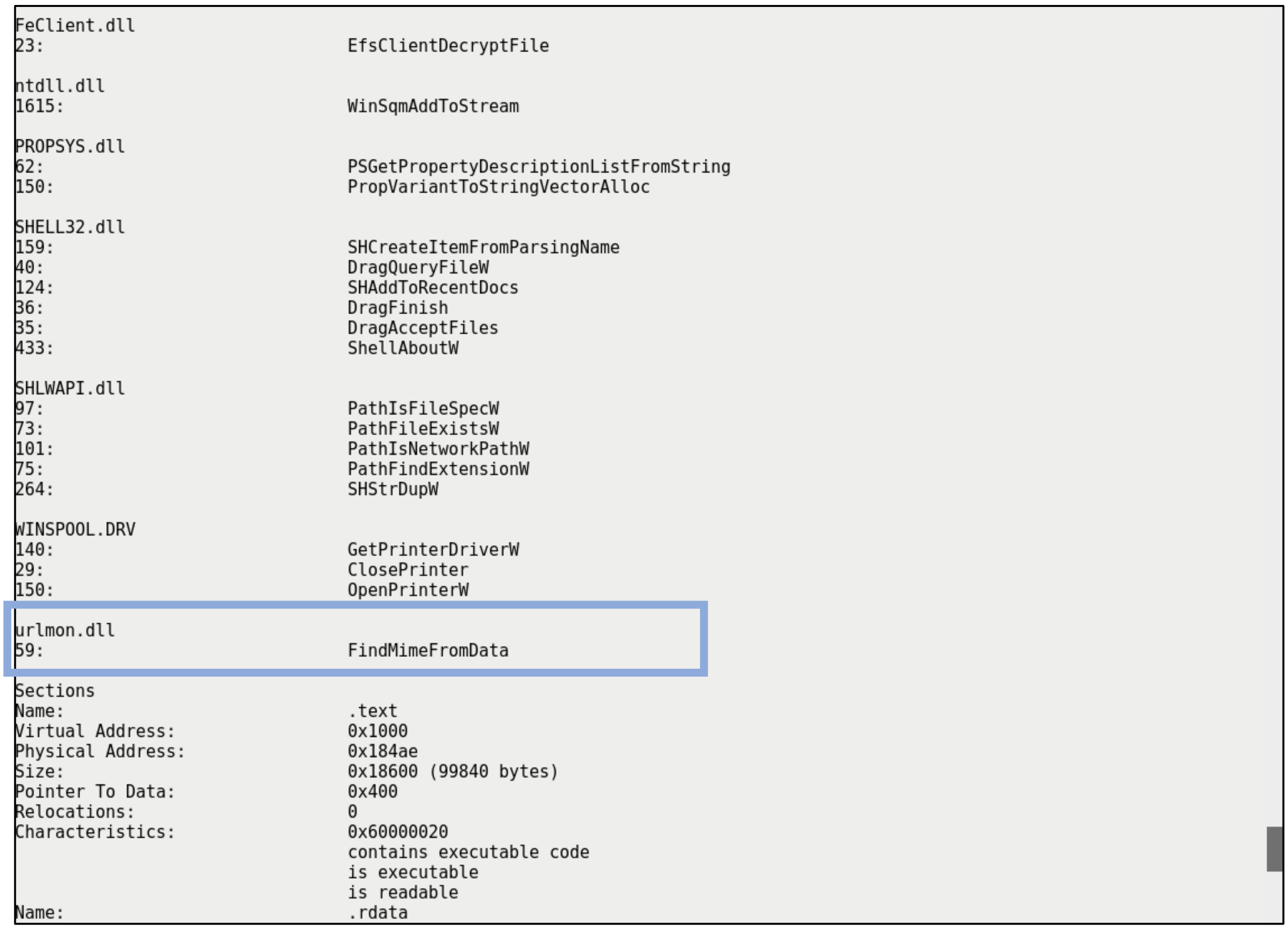


```
root@kali:/usr/share/volatility# readpe dll_inj/executable.6120.exe

DOS Header
Magic number:                    0x5a4d (MZ)
Bytes in last page:              144
Pages in file:                   3
Relocations:                     0
Size of header in paragraphs:    4
Minimum extra paragraphs:        0
Maximum extra paragraphs:        65535
Initial (relative) SS value:     0
Initial SP value:                0xb8
Initial IP value:                0
Initial (relative) CS value:     0
Address of relocation table:     0x40
Overlay number:                  0
OEM identifier:                  0
OEM information:                 0
PE header offset:                0xf0

COFF/File header
Machine:                         0x8664 x86-64 (64-bits)
Number of sections:              6
Date/time stamp:                 1468635242 (Sat - 16 Jul 2016 02:14:02 UTC)
Symbol Table offset:             0
Number of symbols:               0
Size of optional header:         0xf0
Characteristics:                 0x22
                                 executable image
                                 can handle more than 2 GB addresses
```



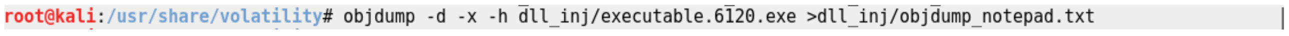
```
Optional/Image header
Magic number:                    0x20b (PE32+)
Linker major version:            14
Linker minor version:            0
Size of .text section:           0x18600
Size of .data section:           0x24e00
Size of .bss section:            0
Entrypoint:                      0x187d0
Address of .text section:        0x1000
ImageBase:                       0x7ff7de4f0000
Alignment of sections:           0x1000
Alignment factor:                0x200
Major version of required OS:    10
Minor version of required OS:    0
Major version of image:          10
Minor version of image:          0
Major version of subsystem:      10
Minor version of subsystem:      0
Size of image:                   0x41000
Size of headers:                 0x400
Checksum:                        0x47c64
Subsystem required:              0x2 (Windows GUI)
DLL characteristics:             0xc160
Size of stack to reserve:        0x80000
Size of stack to commit:         0x11000
Size of heap space to reserve:   0x100000
Size of heap space to commit:    0x1000

Data directories
Import Table:                    0x1f300 (600 bytes)
Resource Table:                  0x26000 (105704 bytes)
Exception Table:                 0x25000 (2244 bytes)
Base Relocation Table:           0x40000 (524 bytes)
Debug:                           0x1e1d0 (56 bytes)
Load Config Table:               0x1a550 (208 bytes)
Import Address Table (IAT):      0x1a620 (2424 bytes)

ADVAPI32.dll
533:                             OpenProcessToken
367:                             GetTokenInformation
238:                             DuplicateEncryptionInfoFile
678:                             RegSetValueExW
662:                             RegQueryValueExW
612:                             RegCreateKeyW
600:                             RegCloseKey
649:                             RegOpenKeyExW
```

[snip]



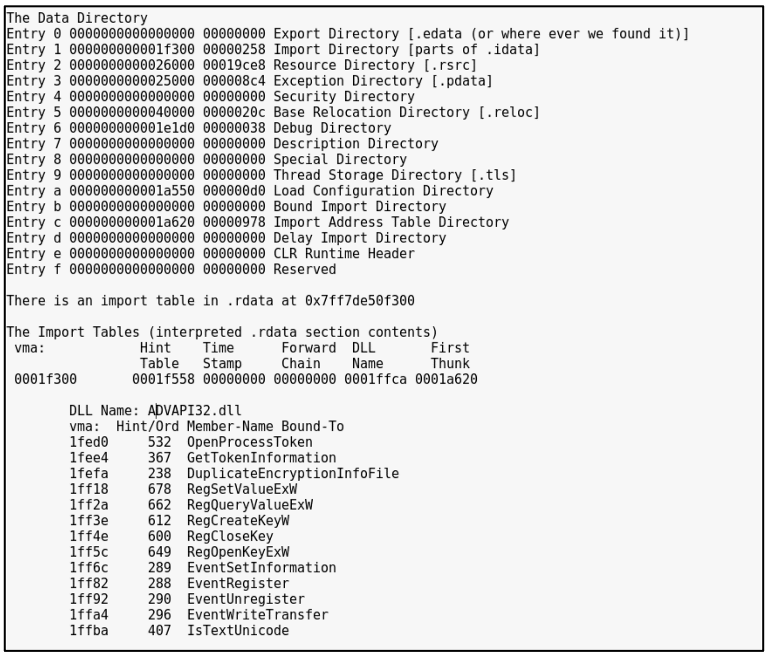
```
FeClient.dll
23:                             EfsClientDecryptFile

ntdll.dll
1615:                           WinSqmAddToStream

PROPSYS.dll
62:                             PSGetPropertyDescriptionListFromString
150:                            PropVariantToStringVectorAlloc

SHELL32.dll
159:                            SHCreateItemFromParsingName
40:                             DragQueryFileW
124:                            SHAddToRecentDocs
36:                             DragFinish
35:                             DragAcceptFiles
433:                            ShellAboutW

SHLWAPI.dll
97:                             PathIsFileSpecW
73:                             PathFileExistsW
101:                            PathIsNetworkPathW
75:                             PathFindExtensionW
264:                            SHStrDupW

WINSPOOL.DRV
140:                            GetPrinterDriverW
29:                             ClosePrinter
150:                            OpenPrinterW

urlmon.dll
59:                             FindMimeFromData

Sections
Name:                           .text
Virtual Address:                0x1000
Physical Address:               0x184ae
Size:                           0x18600 (99840 bytes)
Pointer To Data:                0x400
Relocations:                    0
Characteristics:                0x60000020
                                contains executable code
                                is executable
                                is readable
Name:                           .rdata
```

As one can see, the IAT is displayed as well as the imported DLL's names. Then the information of the .text is displayed. The following objdump command is used to create a text file with the disassembly information.



```
root@kali:/usr/share/volatility# objdump -d -x -h dll_inj/executable.6120.exe >dll_inj/objdump_notepad.txt
```

which confirms the readpe output. Part of this text file is shown below.



```
The Data Directory
Entry 0 0000000000000000 00000000 Export Directory [.edata (or where ever we found it)]
Entry 1 000000000001f300 00000258 Import Directory [parts of .idata]
Entry 2 0000000000026000 00019ce8 Resource Directory [.rsrc]
Entry 3 0000000000025000 000008c4 Exception Directory [.pdata]
Entry 4 0000000000000000 00000000 Security Directory
Entry 5 0000000000040000 0000020c Base Relocation Directory [.reloc]
Entry 6 000000000001e1d0 00000038 Debug Directory
Entry 7 0000000000000000 00000000 Description Directory
Entry 8 0000000000000000 00000000 Special Directory
Entry 9 0000000000000000 00000000 Thread Storage Directory [.tls]
Entry a 000000000001a550 000000d0 Load Configuration Directory
Entry b 0000000000000000 00000000 Bound Import Directory
Entry c 000000000001a620 00000978 Import Address Table Directory
Entry d 0000000000000000 00000000 Delay Import Directory
Entry e 0000000000000000 00000000 CLR Runtime Header
Entry f 0000000000000000 00000000 Reserved

There is an import table in .rdata at 0x7ff7de50f300

The Import Tables (interpreted .rdata section contents)
 vma:            Hint    Time      Forward  DLL       First
                 Table   Stamp     Chain    Name      Thunk
 0001f300       0001f558 00000000 00000000 0001ffca 0001a620

        DLL Name: ADVAPI32.dll
        vma:  Hint/Ord Member-Name Bound-To
        1fed0     532  OpenProcessToken
        1fee4     367  GetTokenInformation
        1fefa     238  DuplicateEncryptionInfoFile
        1ff18     678  RegSetValueExW
        1ff2a     662  RegQueryValueExW
        1ff3e     612  RegCreateKeyW
        1ff4e     600  RegCloseKey
        1ff5c     649  RegOpenKeyExW
        1ff6c     289  EventSetInformation
        1ff82     288  EventRegister
        1ff92     290  EventUnregister
        1ffa4     296  EventWriteTransfer
        1ffba     407  IsTextUnicode
```

On the other hand, **the list of imported DLLs resulting from the code, has the same entries**:

```
['advapi32.dll',
 'kernel32.dll',
 'gdi32.dll',
 'user32.dll',
 'msvcrt.dll',
 'api-ms-win-core-com-l1-1-1.dll',
 'oleaut32.dll',
 'api-ms-win-core-synch-l1-2-0.dll',
 'api-ms-win-core-rtlsupport-l1-2-0.dll',
 'api-ms-win-core-errorhandling-l1-1-1.dll',
 'api-ms-win-core-processthreads-l1-1-2.dll',
 'api-ms-win-core-libraryloader-l1-2-0.dll',
 'api-ms-win-core-profile-l1-1-0.dll',
 'api-ms-win-core-sysinfo-l1-2-1.dll',
 'api-ms-win-core-heap-l1-2-0.dll',
 'api-ms-win-core-winrt-string-l1-1-0.dll',
 'api-ms-win-core-winrt-error-l1-1-1.dll',
 'api-ms-win-core-string-l1-1-0.dll',
 'api-ms-win-core-winrt-l1-1-0.dll',
 'api-ms-win-core-debug-l1-1-1.dll',
 'comctl32.dll',
 'comdlg32.dll',
 'feclient.dll',
 'ntdll.dll',
 'propsys.dll',
 'shell32.dll',
 'shlwapi.dll',
 'winspool.drv',
 'urlmon.dll']
```

So, this part of the code is sufficiently tested. The only problem seems to be what will happen in case one or more pages in the PE header or IAT are not memory resident. For now, in case the IAT cannot be reproduced, a warning is displayed.

Process's Threads

The following figure is created to help understanding the code for getting the process's threads with just one look. Once again, the blue dashed lines show how the respective fields are analyzed and the blue arrows represent pointers.

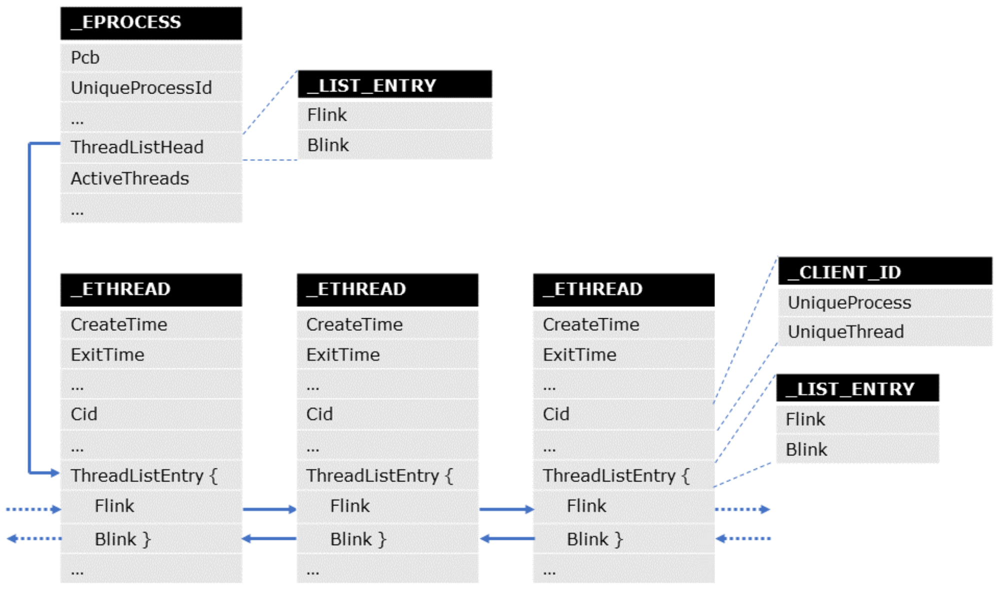


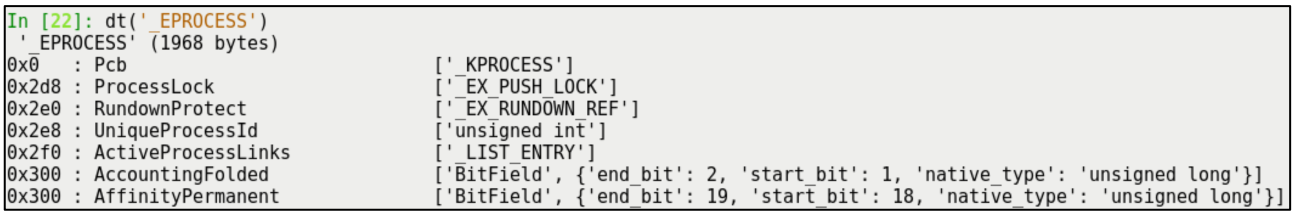
Figure 24 – Diagram showing Processes' and Threads' data structures

All that must be done is following the processes' ThreadListHead which is a doubly linked list that chains together all the process' threads (each list element is an _ETHREAD) [6]. The ThreadListEntry of each _ETHREAD points to the next _ETHREAD [26][53][54]. This is implemented by a for loop (taken from \volatility\plugins\malware\threads.py):

As seen above, the items of interest regarding the threads are creation and exit time as well as process and thread id. The last are stored in Cid structure. This information is needed so that one thread can be correlated to a loaded DLL using TimeWindow parameter, as explained before.

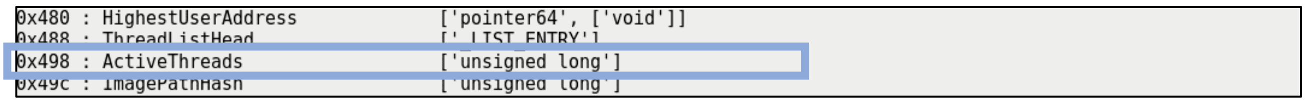
```
#---- Get Threads in the context of the process (in Thread List) ----
Thread_List=[]
for thread in process.ThreadListHead.list_of_type("_ETHREAD", "ThreadListEntry"):
        timestamp_utc = calendar.timegm(time.strptime(str(thread.CreateTime),
                            "%Y-%m-%d %H:%M:%S UTC+0000"))
        Thread_List.append([int(thread.Cid.UniqueProcess),int(thread.Cid.UniqueThread,
                            str(thread.CreateTime), timestamp_utc,
                            str(thread.ExitTime or ' '), hex(thread.StartAddress)])
```

All the involved Windows data structures are presented below, inside Volatility's volshell plugin environment (alternatively to the Windows kernel debugger used before).



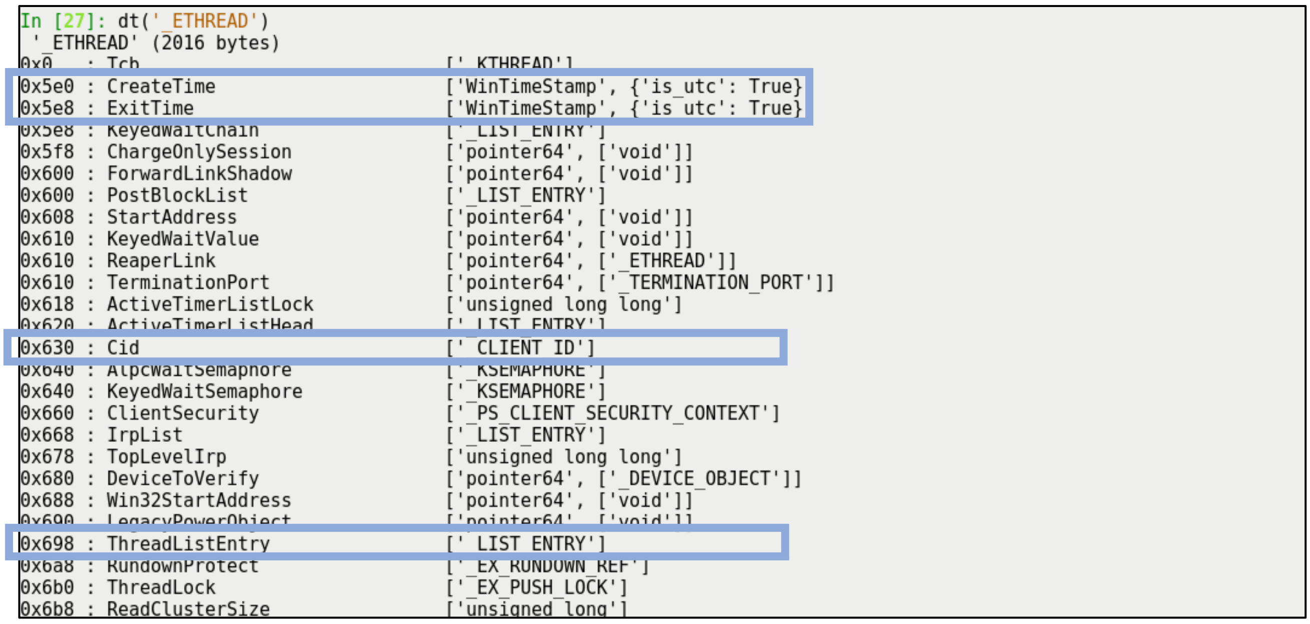
```
In [22]: dt('_EPROCESS')
 '_EPROCESS' (1968 bytes)
0x0   : Pcb                          ['_KPROCESS']
0x2d8 : ProcessLock                  ['_EX_PUSH_LOCK']
0x2e0 : RundownProtect               ['_EX_RUNDOWN_REF']
0x2e8 : UniqueProcessId              ['unsigned int']
0x2f0 : ActiveProcessLinks           ['_LIST_ENTRY']
0x300 : AccountingFolded             ['BitField', {'end_bit': 2, 'start_bit': 1, 'native_type': 'unsigned long'}]
0x300 : AffinityPermanent            ['BitField', {'end_bit': 19, 'start_bit': 18, 'native_type': 'unsigned long'}]
```

[snip]



```
0x480 : HighestUserAddress           ['pointer64', ['void']]
0x488 : ThreadListHead               ['_LIST_ENTRY']
0x498 : ActiveThreads                ['unsigned long']
0x49c : ImagePathHash                ['unsigned long']
```

[snip]


```
In [27]: dt('_ETHREAD')
 '_ETHREAD' (2016 bytes)
0x0    : Tcb                          ['_KTHREAD']
0x5e0 : CreateTime                    ['WinTimeStamp', {'is_utc': True}
0x5e8 : ExitTime                      ['WinTimeStamp', {'is_utc': True}
0x5e8 : KeyedWaitChain                ['_LIST_ENTRY']
0x5f8 : ChargeOnlySession             ['pointer64', ['void']]
0x600 : ForwardLinkShadow             ['pointer64', ['void']]
0x600 : PostBlockList                 ['_LIST_ENTRY']
0x608 : StartAddress                  ['pointer64', ['void']]
0x610 : KeyedWaitValue                ['pointer64', ['void']]
0x610 : ReaperLink                    ['pointer64', ['_ETHREAD']]
0x610 : TerminationPort               ['pointer64', ['_TERMINATION_PORT']]
0x618 : ActiveTimerListLock           ['unsigned long long']
0x620 : ActiveTimerListHead           ['_LIST_ENTRY']
0x630 : Cid                           ['_CLIENT_ID']
0x640 : AlpcWaitSemaphore             ['_KSEMAPHORE']
0x640 : KeyedWaitSemaphore            ['_KSEMAPHORE']
0x660 : ClientSecurity                ['_PS_CLIENT_SECURITY_CONTEXT']
0x668 : IrpList                       ['_LIST_ENTRY']
0x678 : TopLevelIrp                   ['unsigned long long']
0x680 : DeviceToVerify                ['pointer64', ['_DEVICE_OBJECT']]
0x688 : Win32StartAddress             ['pointer64', ['void']]
0x690 : LegacyPowerObject             ['pointer64', ['void']]
0x698 : ThreadListEntry               ['_LIST_ENTRY']
0x6a8 : RundownProtect                ['_EX_RUNDOWN_REF']
0x6b0 : ThreadLock                    ['_EX_PUSH_LOCK']
0x6b8 : ReadClusterSize               ['unsigned long']
```

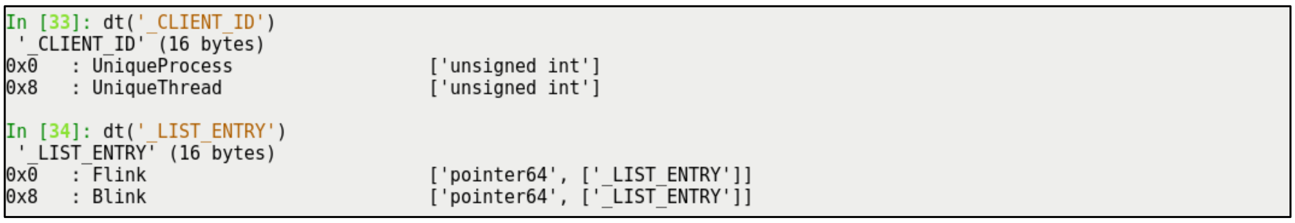

```
In [33]: dt('_CLIENT_ID')
 '_CLIENT_ID' (16 bytes)
0x0   : UniqueProcess                 ['unsigned int']
0x8   : UniqueThread                  ['unsigned int']

In [34]: dt('_LIST_ENTRY')
 '_LIST_ENTRY' (16 bytes)
0x0   : Flink                         ['pointer64', ['_LIST_ENTRY']]
0x8   : Blink                         ['pointer64', ['_LIST_ENTRY']]
```

Figure 25 - Processes' and Threads' data structures

Handles on the Threads

As already mentioned, one key point for DLL injection detection is the fact that the thread that executes the LoadLibrary function inside the context of the legitimate process is created by the malicious process. **Consequently, in case a thread is handled by the same process under which it is executed, the thread is not suspicious**. To derive this conclusion a list of the process handles (Handle_List) is created (code taken from \volatility\plugins\handles.py):

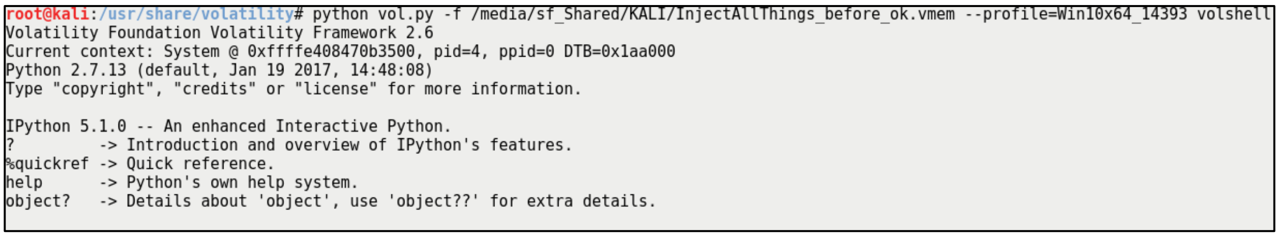
```
#---- Get Process' Handles (in Handle_List) and update ALL Processes' handles
#(AllProcessHandle_List) ----
Handle_List=[]
process.ObjectTable.HandleTableList
pid=int(p_id)
for handle in process.ObjectTable.handles():
        if not handle.is_valid():
                continue
        object_type = handle.get_object_type()
        if object_type == "Thread":
                thrd_obj = handle.dereference_as("_ETHREAD")

                # Details handle PID, TID, process id that owns the handle
                Handle_List.append([int(thrd_obj.Cid.UniqueProcess),
                                        int(thrd_obj.Cid.UniqueThread),p_id])

                AllProcessHandle_List([int(thrd_obj.Cid.UniqueProcess),
                                           int(thrd_obj.Cid.UniqueThread),p_id])
```

For each possibly suspicious thread and DLL combination (per process), the processes’ handle list is checked and in case a corresponding entry is found, the thread is considered as not suspicious. AllProcessHandle_List is also updated so at the end it contains information about all handles from the memory image.

As presented in the detection flowchart, **after “scanning” all processes, each pair of suspicious DLL/thread is checked to ascertain whether there is a handle on the thread**. If the handle exists, the process that created this handle is the one that also created the thread and consequently is the malicious process. The malicious’ process id is also updated in the suspicious DLL/thread list. On the other hand, if no handle is found, a warning is displayed because although this is an anomaly, it is not indicating a DLL injection with certainty. Maybe the region of memory that holds the handles is mapped.

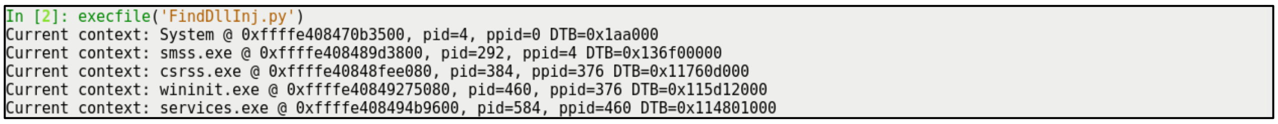
3.4 Calling FindDllInj.py

At first, FindDllInj.py must be copied in Volatility’s directory, e.g. /usr/share/volatility/ and then the script is called inside Volatility’s volshell plugin, for example:



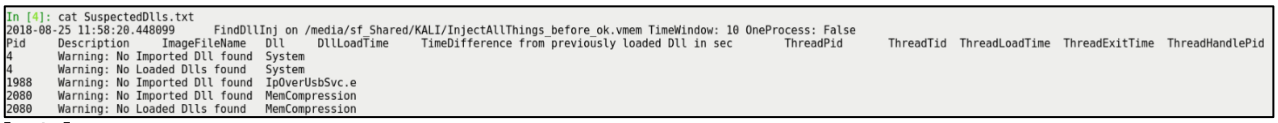
```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/InjectAllThings_before_ok.vmem --profile=Win10x64_14393 volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0xffffe408470b3500, pid=4, ppid=0 DTB=0x1aa000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

while the script is executing, it displays the various processes’ context



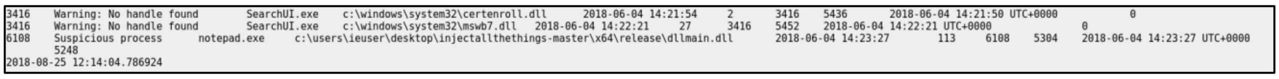
```
In [2]: execfile('FindDllInj.py')
Current context: System @ 0xffffe408470b3500, pid=4, ppid=0 DTB=0x1aa000
Current context: smss.exe @ 0xffffe408489d3800, pid=292, ppid=4 DTB=0x136f00000
Current context: csrss.exe @ 0xffffe40848fee080, pid=384, ppid=376 DTB=0x11760d000
Current context: wininit.exe @ 0xffffe40849275080, pid=460, ppid=376 DTB=0x115d12000
Current context: services.exe @ 0xffffe408494b9600, pid=584, ppid=460 DTB=0x114801000
```

and the output file can be manipulated as a text file or imported in spreadsheet



```
In [4]: cat SuspectedDlls.txt
2018-08-25 11:58:20.448099      FindDllInj on /media/sf_Shared/KALI/InjectAllThings_before_ok.vmem TimeWindow: 10 OneProcess: False
Pid     Description     ImageFileName   Dll     DllLoadTime     TimeDifference from previously loaded Dll in sec        ThreadPid       ThreadTid  ThreadLoadTime  ThreadExitTime  ThreadHandlePid
4       Warning: No Imported Dll found  System
4       Warning: No Loaded Dlls found   System
1988    Warning: No Imported Dll found  IpOverUsbSvc.e
2080    Warning: No Imported Dll found  MemCompression
2080    Warning: No Loaded Dlls found   MemCompression
```

[snip]



```
3416    Warning: No handle found        SearchUI.exe    c:\windows\system32\certenroll.dll      2018-06-04 14:21:54     2       3416    5436    2018-06-04 14:21:50 UTC+0000            0
3416    Warning: No handle found        SearchUI.exe    c:\windows\system32\mswb7.dll   2018-06-04 14:22:21     27      3416    5452    2018-06-04 14:22:21 UTC+0000            0
6108    Suspicious process      notepad.exe     c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll       2018-06-04 14:23:27     113     6108    5304    2018-06-04 14:23:27 UTC+0000
        5248
2018-08-25 12:14:04.786924
```

3.5 FindDLLInj Testing

3.5.1 Testing Environment

In websites https://gist.github.com/zmwangx/e728c56f428bc703c6f6 and https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/ there are many windows virtual machines available for and for testing was downloaded Windows 10 Enterprise Evaluation, Build 14393.rs1_release.180209-1727 (https://az792536.vo.msecnd.net/vms/VMBuild_20160802/VMWare/MSEdge/MSEdge.Win10_RS1.VMWare.zip) which was processed with VMware® Workstation 12 Pro software, version 12.5.2 build-4638234 [9].

To perform the DLL injection, the source code of two different malware Projects was used: injectAllTheThings [55] and InjectProc [41]. The source code was compiled using Microsoft Visual Studio Community 2017, version 15.5.6. In both cases, the "malicious" DLL (dllmain.dll and mbox.dll respectively) just pops up a window displaying "Process attach" or "Injected".

On the investigating hand, KALI Linux 2017.1 was used (downloaded file name kali-linux-2017.1-amd64.iso from https://www.kali.org/downloads/) which includes Volatility's Foundation Volatility Framework 2.6. KALI and was opened with Oracle VirtualBox version 5.2.12 r122591 (Qt5.6.2) [10].

Memory images are taken at various moments, as described in the following table: Before Injection, on DLL execution, after terminating the DLL, after terminating the injected process, after terminating the injection process, on DLL injection in more than one process, e.t.c.

In each case, just before the memory image was created, the running Windows Virtual Machine environment was checked to validate that the DLL injection was achieved. For this purpose, ProcessHacker software was used (downloaded from *https://processhacker.sourceforge.io/downloads.php*). To be more precise, it was checked that the "malicious" DLL was loaded within the specific process address space and was recorded any information needed to verify that FindDllInj.py works correctly: the malicious full path and name, the injected process ID, the thread ID that loaded the DLL and the injection process ID. In appendix, section A DLL injection testing example, is shown exactly how this information was retrieved, using image InjectAllThings_before_ok.vmem as an example. It is also shown how the script was tested and its output confirmed.

3.5.2 Testing results

The following table presents the results of the created script execution (FindDLLInj.py) using several memory images files. The second column contains details about the system when the memory image is taken, the corresponding image file name and the output file name created by the script. The third column contains one or more lines of the output file regarding the suspicious process found. In case there is not any suspicious process, no output line is included. Not all the warnings are presented, only a few as an example. The output line of the first test is explained in detail and analyzed according to the output file layout. The output lines of the remaining tests have the same structure. All the results were confirmed as described in A DLL injection testing example.

The testing results on the memory images injected using the code presented in **injectAllTheThings** project [55] are:

Test No.	Test Memory Image Characteristics	Output line of interest
1	**DLL injection in process notepad.exe (pid=6108)** During DLL execution (respective window popped up) Injecting process still active (injectAllTheThings.exe , pid 5248) Image File name: InjectAllThings_before_ok.vmem Output file name: SuspectedDlls - InjectAllThings_before_ok.txt	6108 Suspicious process notepad.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 14:23:27 113 6108 5304 2018-06-04 14:23:27 UTC+0000 5248 This output line is analyzed as:

Pid	6108
Description	Suspicious process
ImageFileName	notepad.exe
Dll	c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll
DllLoadTime	2018-06-04 14:23:27
TimeDifference	113
ThreadPid	6108
ThreadTid	5304
ThreadLoadTime	2018-06-04 14:23:27
ThreadExitTime	
ThreadHandlePid	5248

2	Injection in process notepad.exe (pid 6108) achieved During DLL execution (respective window popped up) **Injecting Process terminated** Image File name: InjectAllThings_closeExe.vmem Output file name: SuspectedDlls - InjectAllThings_closeExe.txt Note: process 1140 is ProcessHacker	6108 Suspicious process notepad.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 14:23:27 113 6108 5304 2018-06-04 14:23:27 UTC+0000 1140
3	Injection in process notepad.exe (pid 6108) achieved **DLL terminated** (respective window closed) Injecting Process terminated Image File name: InjectAllThings_after_ok.vmem Output file name: SuspectedDlls - dll_inj_after_ok.txt	No suspicious processes, just warnings. Since the corresponding thread is terminated, the script gives no output.
4	**Virtual machine restarted** **Injection in two different processes explorer.exe (pid 5000) and notepad.exe (pid 2560)** During DLL execution (respective window popped up) Injecting Processes still active Image File name: InjectAllThings_2proc.vmem Output file name: SuspectedDlls - InjectAllThings_2proc.txt Note: Process 6060 is ProcessHacker, the OneProcess parameter is set to False	5000 Suspicious process explorer.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 15:05:16 195 5000 68 2018-06-04 15:05:16 UTC+0000 244 2560 Suspicious process notepad.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 15:02:09 125 2560 2540 2018-06-04 15:02:08 UTC+0000 6060 2560 Suspicious process notepad.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 15:02:09 125 2560 2540 2018-06-04 15:02:08 UTC+0000 3492

5	Injection in two different Processes **DLL terminated** (respective window closed) Injecting Process active Image File name: InjectAllThings_2proc_afterOK.vmem Output file name: SuspectedDlls - InjectAllThings_2proc_afterOK.txt Note: The OneProcess parameter is set to False	No suspicious processes, Just warnings

The testing results on the memory images injected using the code technique presented in **InjectProc** project [41] are:

Note: Inject proc author in its website states that it is only tested on Windows 10 build 1703, 64bit.

Test No.	**Test Memory Image Characteristics**	**Output line of interest**
6	**Clean system**, notepad.exe which is going to be injected (pid 6120) is running but not injected. Image File name: MSEdge - Win10_preview-eaaa27c2-dll_inj_before.vmem Output file name: SuspectedDlls - MSEdge - Win10_preview-eaaa27c2-dll_inj_before.txt	No suspicious process found. Warnings such as: 4 Warning: No Imported Dll found System 4 Warning: No Loaded Dlls found System 664 Warning: No handle found svchost.exe c:\windows\system32\licensemanagersvc.dll 2018-03-31 10:38:03 5 664 912 2018-03-31 10:37:57 UTC+0000 0 1136 Warning: No handle found svchost.exe c:\windows\system32\cryptsvc.dll 2018-03-31 10:35:49 3 1136 1684 2018-03-31 10:35:46 UTC+0000 0

7	**DLL Injection in process Notepad.exe (pid 6120).** During DLL execution (respective window popped up). Injecting Process (pid 1080) still active. Image File name: MSEdge - Win10_preview-eaaa27c2-dll_inj_after.vmem Output file name: SuspectedDlls-MSEdge - Win10_preview-eaaa27c2-dll_inj_after.txt	Output (other than warnings) 6120 Suspicious process notepad.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-03-31 12:52:14 8026 6120 3140 2018-03-31 12:52:14 UTC+0000 1080
8	**Virtual Machine restarted** DLL Injection in process notepad.exe (pid 6756) achieved. **DLL terminated** (respective window closed) Injecting Process just terminated Image File name: MSEdge - Win10_preview-eaaa27c2-dll_inj_after_term-new.vmem Output file name: SuspectedDlls -MSEdge - Win10_preview-eaaa27c2-dll_inj_after_term-new (-oneproc=False).txt	No suspicious process found
9	**Virtual Machine restarted** **Injection in process explorer.exe (pid 2760)** During DLL execution (respective window popped up) injecting Process still active (pid 5336) Image File name: MSEdge - Win10_dll_inj_before_ok.vmem Output file name: SuspectedDlls-MSEdge - Win10_dll_inj_before_ok.txt	2760 Suspicious process explorer.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-05-29 14:50:54 61 2760 5620 2018-05-29 14:50:54 UTC+0000 5336

10	Virtual Machine not restarted Injection in process explorer.exe (pid 2760) completed **DLL terminated** (respective window closed) Injecting Process terminated Image File name: MSEdge - Win10_dll_inj_after_ok.vmem Output file name: SuspectedDlls - MSEdge - Win10_dll_inj_after_ok.txt	No suspicious process found, just warnings. Since the corresponding thread is terminated, the script gives no output.
11	Virtual Machine restarted **Injection in three different Processes** During DLL execution (respective windows popped up). Image File name: MSEdge - Win10_preview-eaaa27c2-3_DifferentProcs.vmem Output file name: SuspectedDllsWin10_preview-eaaa27c2-3_DifferentProcs.txt Note: The OneProcess parameter is set to False	408 Suspicious process explorer.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:14:42 26 408 3908 2018-08-31 14:14:42 UTC+0000 5372 6052 Suspicious process notepad.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:12:21 43 6052 3632 2018-08-31 14:12:20 UTC+0000 5436 4328 Suspicious process mspaint.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:13:48 56 4328 2144 2018-08-31 14:13:48 UTC+0000 5056

12	**Injection in three different Processes One of them terminated (pid 6052)** injecting Processes still active Image File name: MSEdge - Win10_preview-eaaa27c2-3_DifferentProcs-1_Term.vmem Output file name: SuspectedDlls- Win10_preview-eaaa27c2-3_DifferentProcs-1_Term.txt Note: The OneProcess parameter is set to False	408 Suspicious process explorer.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:14:42 26 408 3908 2018-08-31 14:14:42 UTC+0000 5372 4328 Suspicious process mspaint.exe c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:13:48 56 4328 2144 2018-08-31 14:13:48 UTC+0000 5056

Table 1 – Remote DLL Injection Detection Testing Results

3.6 Testing conclusions

As shown in the results' table, the idea for DLL injection detection works satisfactory. In case there is an active DLL injection, the message is straightforward: "Suspicious process" and the DLL is tracked down, as well as the corresponding injected process, the thread that loaded the DLL and the injection process. This also applies when there are more than one injected processes, as long as OneProcess parameter is set to false.

The warnings like "No Imported Dll found" or "No Loaded Dlls found", together with the corresponding executable name, may give the analyst a hint worth investigating. The "No handle found" warning which also displays the full DLL name, may help the analyst to recognize a suspicious DLL from its name or path. In general, the warnings can lead to targeted search for information either from the memory image or from other sources, such as the disk.

The proposed solution works only when the injected DLL is currently executing, regardless whether the malware that performs the injection has been terminated. The reason for that when the DLL has finished executing, the corresponding thread is terminated. Information about the terminated thread cannot be retrieved by the script, although there is information available for some terminated threads, such as exit time. The same problem appears when the corresponding volatility plugins such as timeliner, threads and thrdscan have been used in the aforementioned memory images. This confirms that this is an issue that concerns the data available in the memory rather than the code used. Only currently active processes and threads are available in memory. In case of a DLL injection that happened in the past, an analyst must gather information from both the memory and the disk of detect it. In the context of this thesis, only memory is used, so this is justified problem.

3.7 Future Improvements

As already mentioned, FindDLLInj.py is executed inside Volatilitys' volshell plugin. The conversion of it into a new Volatility plugin is a good idea. Alongside, the warning messages displayed by volshell environment such as the following could be manipulated.

```
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B951D18,
instantiating String

WARNING : volatility.debug    : NoneObject as string: Invalid offset 18446717726403732832
for dereferencing Buffer as String

WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF7C0C92A6E,
instantiating String

WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x00000000,
instantiating _FILE_OBJECT
```

The size of the _IMAGE_IMPORT_DESCRIPTOR, described in section Scan processes' memory for IAT entries, is not defined by getting the object size from the windows profile, but it is hardcoded. This means that the code may not work correctly in another windows version with an altered object size. This problem could also be solved by creating the plugin.

The script is tested only on Windows 10 system. It could be tested and in other windows versions to see its correspondence. For Windows 8.1 the code should be working exactly as it is, but for older versions a change should be made: LoadCount DLL attribute should be used instead of ObsoleteLoadCount.

For now, a DLL is correlated with one or more threads associating its load time with the thread creation time using a time window. This seems to work all right, but maybe there is a more precise way to do the correlation that did not revealed within this work. This involves greater deepen on the Windows objects and data structures manipulating the threads.

As already referred, there is the case that the script is not able to reconstruct the IAT (Import Address Table). This does not cause any false results because of the other safeguards used and the worst scenario is the production of a false positive result. The Volatility's plugin impscan [18] creators claim that "the IAT may not properly be reconstructed due to the high likelihood that one or more pages in the PE header or IAT are not memory resident (paged). Thus, we created impscan. Impscan identifies calls to APIs without parsing a PE's IAT." It seems that this plugin could be a good inspiration for creating a code that recreates IAT with certainty. In this case, the comparison of the IAT list to the loaded DLL, could give definite results.

During the tests, it is observed that the script finds out less distinct handles than the volatility command python vol.py -f <image file name> --profile="Win10x64_14393" handles -t THREAD. Although this is not causing any false positives or any overlooking DLL detection, it could lead to more warnings of type "No handle found".

4 Conclusion

This thesis focuses on the detection of two different process injection techniques: Process Hollowing and Remote DLL Injection. For this purpose, these techniques have been studied (meaning their concept and the source code of two different injectors for each technique) and presented in detail. Injections were performed in Windows 10 Virtual Machines, using different injector malware and totally 15 memory images are acquired. Dynamic memory analysis was applied on the memory images using Volatility's Core plugins, downloaded plugins and custom-made scripts executed inside the environment of volshell plugin. Either way, this document could be used as an introduction to the Volatility Framework.

Process Hollowing is commonly used, so it has caused the interest of the malware analyst. There is a lot of relevant literature, articles and posts. In the later years, there have been created Volatility plugins dedicated for Process Hollowing detection, so the creation of another plugin would not contribute to the analyst community. In this thesis, the concept of Hollow Process Injection is described, as well as the basic variations of it. The anomalies it causes in the memory and the possible giveaways are analyzed through the proposed methodology of detection. This methodology is performed using various test images and its results are confirmed. After thorough research, it is believed by the author of this document, that this methodology has incorporated and organized in distinct steps most of the current literature, relevant articles on the web and research on the subject.

Regarding the Classic (or remote) DLL injection, a completely new methodology of detection is proposed, verified, implemented and tested. This methodology does not rely on standard DLL injection detection techniques, but on the fact that LoadLibrary and CreateRemoteThread functions are used as well as on time sequence of events. The key points of this detection approach are analyzed and the relative Windows Data structures are visualized and explained. This alone could help those who wish to deepen on these structures. The whole idea is implemented in a python script of approximately 200 lines of code that can be executed inside Volatility's volshell plugin environment. The results of the script executed on 12 distinct memory images, presented in the relative table, prove that the script works satisfactory, as reasoned in Testing conclusions paragraph. A main improvement for the script could be its conversion to a distinct Volatility plugin that can be independently called, not through volshell. This issue is analyzed in Future Improvements.

5 References

[1] ENISA, Threat Landscape Report 2017, ENISA, published 15 January 2018, ISBN 978-92-9204-250-9, ISSN 2363-3050, DOI 10.2824/967192, https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2017, last accessed on 11 September 2018

[2] ENISA, ETL (ENISA Thread Landscape) Web based tool, ENISA, https://etl.enisa.europa.eu/#/, last accessed on 11 September 2018

[3] Michael Sikorski and Andrew Honig, PRACTICAL MALWARE ANALYSIS, no starch press, 2012, ISBN-10: 1-59327-290-1, ISBN-13: 978-1-59327-290-6

[4] Nwokedi Idika and Aditya P. Mathur, A Survey of Malware Detection Techniques, February 2, 2007, downloaded from https://www.researchgate.net/publication/229008321_A_survey_of_malware_detection_techniques, last accessed on 11 September 2018

[5] Esan P. Pancha, Extraction of Persistence and Volatile Forensics Evidences from Computer System, International Journal of Computer Trends and Technology (IJCTT) - volume4 Issue 5, May 2013, downloaded from http://ijcttjournal.org/Volume4/issue-5/IJCTT-V4I5P1.pdf, last accessed on 11 September 2018

[6] Michael Hale Ligh, Andrew Case, Jamie Levy, AAron Walters, The Art of Memory Forensics, Wiley, 2014, ISBN: 978-1-118-82499-3

[7] Hal Pomeranz, Detecting Malware with Memory Forensics, SANS Webcast, Oct 2012, http://www.deer-run.com/~hal/, last accessed on 11 September 2018

[8] Technopedia.com, Virtualization, https://www.techopedia.com/definition/719/virtualization, last accessed on 11 September 2018

[9] vmwrare®, https://www.vmware.com/, last accessed on 11 September 2018

[10] Oracle, VirtualBox, https://www.virtualbox.org/, last accessed on 11 September 2018

[11] Microsoft, Hyper-V Technology Overview, https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview, last accessed on 11 September 2018

[12] Parallels, https://www.parallels.com/, last accessed on 11 September 2018

[13] WindowsSCOPE, Cyber Forensics 3.2, http://www.windowsscope.com/windowsscope-cyber-forensics/, last accessed on 11 September 2018

[14] F-Response, https://www.f-response.com/ , last accessed on 11 September 2018

[15] GitHub, Awesome Incident Response, https://github.com/meirwah/awesome-incident-response, last committed on 3 October 2018, last accessed on 10 October 2018

[16] Kali Tools, Volatility Package Description, https://tools.kali.org/forensics/volatility, last accessed on 11 September 2018

[17] VOLATILITY FOUNDATION, Home Page, https://www.volatilityfoundation.org/, last accessed on 11 September 2018

[18] GitHub, volatilityfoundation/volatility - Command Reference, https://github.com/volatilityfoundation/volatility/wiki/Command-Reference, last edited on 22 April 2017, last accessed on 1 October 2018

[19] GitHub, volatilityfoundation/volatility**,** https://github.com/volatilityfoundation/volatility, last accessed on 1 October 2018

[20] GitHub, volatilityfoundation/volatility - Volatility Documentation Project, https://github.com/volatilityfoundation/volatility/wiki/Volatility-Documentation-Project, last edited on 8 September 2015, last accessed on 1 October 2018

[21] Volatility Labs, https://volatility-labs.blogspot.com/, last accessed on 1 October 2018

[22] Ashkan Hosseini, , Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques, ENDGAME, https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process, last accessed on 1 October 2018

[23] Microsoft, Memory Management, https://docs.microsoft.com/en-us/windows/desktop/Memory/memory-management, 31 May 2018, last accessed on 1 October 2018

[24] Tim Sneath, PDC10: Mysteries of Windows Memory Management Revealed: Part One, Microsoft, uploaded on 28 October 2010, https://blogs.msdn.microsoft.com/tims/2010/10/28/pdc10-mysteries-of-windows-memory-management-revealed-part-one, last accessed on 1 October 2018

[25] Mike Czumak, Windows Exploit Development – Part 1: The Basics, Security Sift, written on December 6, 2013 , https://www.securitysift.com/windows-exploit-development-part-1-basics/, last accessed on 1 October 2018

[26] Mark Russinovich - David A. Solomon - Alex Ionescu, Windows® Internals Part 1 6th edition, Microsoft Press, 2012, ISBN: 978-0-7356-4873-9

[27] Matt Pietrek, Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, written on March 1994, https://msdn.microsoft.com/en-us/library/ms809762.aspx, last accessed on 1 October 2018

[28] Microsoft, PE Format, https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format, uploaded on 31 May 2018, last accessed on 1 October 2018

[29] INFOSEC Institute, The Import Directory: Part 1, https://resources.infosecinstitute.com/the-import-directory-part-1/#gref, uploaded on 24 April 2013, , last accessed on 1 October 2018

[30] Matt Pietrek, , An In-Depth Look into the Win32 Portable Executable File Format, msdn magazine, February 2002 issue, https://msdn.microsoft.com/en-us/magazine/bb985992.aspx, , last accessed on 1 October 2018

[31] Wikipedia, Portable Executable, https://en.wikipedia.org/wiki/Portable_Executable, last accessed on 1 October 2018

[32] Microsoft, Thread Handles and Identifiers, https://docs.microsoft.com/en-us/windows/desktop/procthread/thread-handles-and-identifiers, uploaded on 31 May 2018, last accessed on 1 October 2018

[33] Michael Hale Ligh - Steven Adair - Blake Hartstein - Matthew Richard, Malware Analyst's Cookbook and DVD, Wiley Publishing, Inc., 2011, ISBN: 978-0-470-61303-0

[34] GitHub, Process Hollowing, https://github.com/m0n0ph1/Process-Hollowing, last accessed on 1 October 2018

[35] Monmappa K.A., DETECTING DECEPTIVE PROCESS HOLLOWING TECHNIQUES USING HOLLOWFIND VOLATILITY PLUGIN, 2016, https://cysinfo.com/detecting-deceptive-hollowing-techniques/, last accessed on 1 October 2018

[36] Monnappa K.A., Understanding Evasive Hollow Process Injection techniques , CYSINFO 11th Meetup, https://cysinfo.com/11th-meetup-understanding-evasive-hollow-process-injection-techniques/, last accessed on 1 October 2018

[37] 00xsec, Userland API Monitoring and Code Injection Detection, https://0x00sec.org/t/userland-api-monitoring-and-code-injection-detection/5565, uploaded 21 February 2018, last accessed on 1 October 2018

[38] Jared Atkinson and Joe Desimone , Taking Hunting to the Next Level - Hunting in Memory presentation, Endgame, SANS Institute Threat Hunting and IR Summit (April 2017), https://www.sans.org/summit-archives/file/summit-archive-1492714038.pdf, last accessed on 1 October 2018

[39] Luis Rocha, Malware Analysis – Dridex & Process Hollowing, https://countuponsecurity.com/2015/12/07/malware-analysis-dridex-process-hollowing/, uploaded 07 December 2015, last accessed on 1 October 2018

[40] MITRE's ATT&CK, Process Hollowing, https://attack.mitre.org/wiki/Technique/T1093, last accessed on 1 October 2018

[41] GitHub, secrary/InjectProc, InjectProc - Process Injection Techniques, https://github.com/secrary/InjectProc, latest commit on 24 March 2018, last accessed on 1 October 2018

[42] John Leitch, Process Hollowing, http://www.autosectools.com/process-hollowing.pdf, last accessed on 1 October 2018

[43] Eric Monti, Analyzing Malware Hollow Processes, Trustwave SpiderLabs® Blog, https://www.trustwave.com/Resources/SpiderLabs-Blog/Analyzing-Malware-Hollow-Processes/, last accessed on 1 October 2018

[44] Microsoft, Security identifiers, https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/security-identifiers, uploaded on 19 April 2018, last accessed on 1 October 2018

[45] Michael Hale Ligh, Stuxnet's Footprint in Memory with Volatility 2.0, MNIN Security Blog, http://mnin.blogspot.com/2011/06/examining-stuxnets-footprint-in-memory.html, uploaded on 3 June 2011, last accessed on 1 October 2018

[46] GitHub, community/DimaPshoul/, https://github.com/volatilityfoundation/community/tree/master/DimaPshoul, committed on 6 February 2017, , last accessed on 1 October 2018

[47] GitHub, threadmap plugin for Volatility Foundation, https://github.com/kslgroup/threadmap, committed on 21 September 2017, last accessed on 1 October 2018

[48], WIKIPEDIA, DLL injection, https://en.wikipedia.org/wiki/DLL_injection, edited on 18 September 2018, last accessed on 1 October 2018

[49] Dejan Lukan, Using CreateRemoteThread for DLL Injection on Windows, INFOSEC INSTITUTE,https://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows/, posted on 30 May 2013, last accessed on 1 October 2018

[50] WIKIPEDIA, Dynamic-link library, https://en.wikipedia.org/wiki/Dynamic-link_library, last edited on 28 September 2018, last accessed on 1 October 2018

[51] Microsoft, winnt.h header, https://docs.microsoft.com/en-us/windows/desktop/api/winnt/, uploaded on 10 October 2018, last accessed on 10 October 2018

[52] Joachim Bauch, Loading a DLL from memory, https://www.joachim-bauch.de/tutorials/loading-a-dll-from-memory/ , posted on 7 April 2010, last accessed on 10 October 2018

[53] Mark E. Russinovich and David A. Solomon, Processes, Threads, and Jobs in the Windows Operating System, The Microsoft Press Store by Pearson, https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=4, uploaded on 17 June 2009, last accessed on 10 October 2018

[54] CodeMachine, Catalog of key Windows kernel data structures, https://www.codemachine.com/article_kernelstruct.html, last accessed on 10 October 2018

[55] GitHub, fdiskyou/injectAllTheThings, https://github.com/fdiskyou/injectAllTheThings, latest commit on 21 Jul 2017, last accessed on 10 October 2018

[56] GitHub, monnappa22/HollowFind, https://github.com/monnappa22/HollowFind, Latest commit on 24 September 2016, last accessed on 10 October 2018

[57] Each Problem has An End, Difference between memdump, procexedump and procmemdump command in volatality, http://akovid.blogspot.com/2014/02/volatality-procexedump-and-memdump.html, Posted on 17 February 2014, last accessed on 1 October 2018

[58] Microsoft, https://docs.microsoft.com/en-us/windows/desktop/api/, last accessed on 10 October 2018

[59] Mark Russinovich and Ken Johnson, LiveKd v5.62, Microsoft, https://docs.microsoft.com/en-us/sysinternals/downloads/livekd, published on 16 May 2017, last accessed on 1 September 2018

[60] Geoff Chappel, Win32 NTDLL structures LDR_DATA_TABLE_ENTRY, https://www.geoffchappell.com/studies/windows/win32/ntdll/structs/ldr_data_table_entry.htm, Last updated on 2 October 2018, last accessed on 10 October 2018

[61] Microsoft, LoadLibrary and AfxLoadLibrary, https://msdn.microsoft.com/en-us/library/zzk20sxw.aspx, last accessed on 10 October 2018

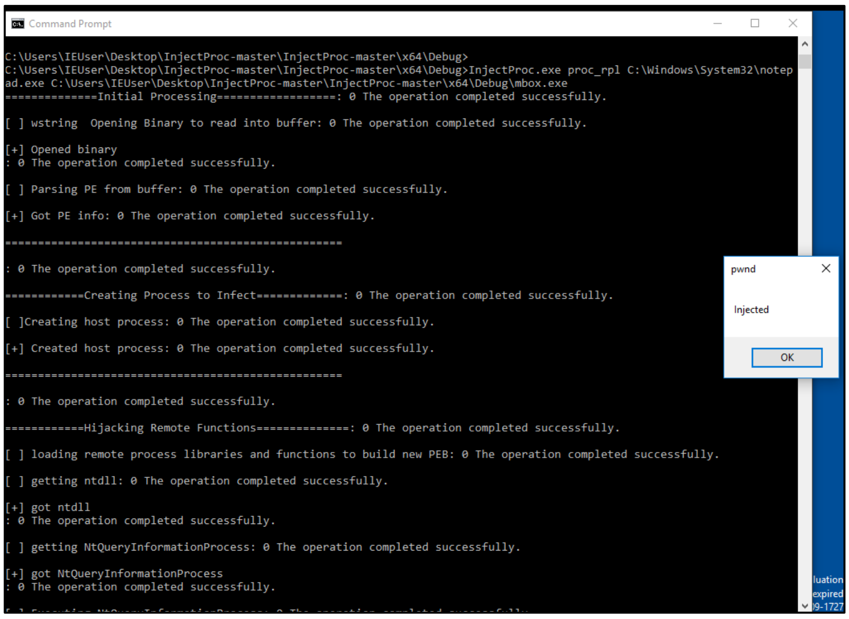
6 Appendix

6.1 Hollow process injection and detection examples

6.1.1 Injection Using InjectProc project

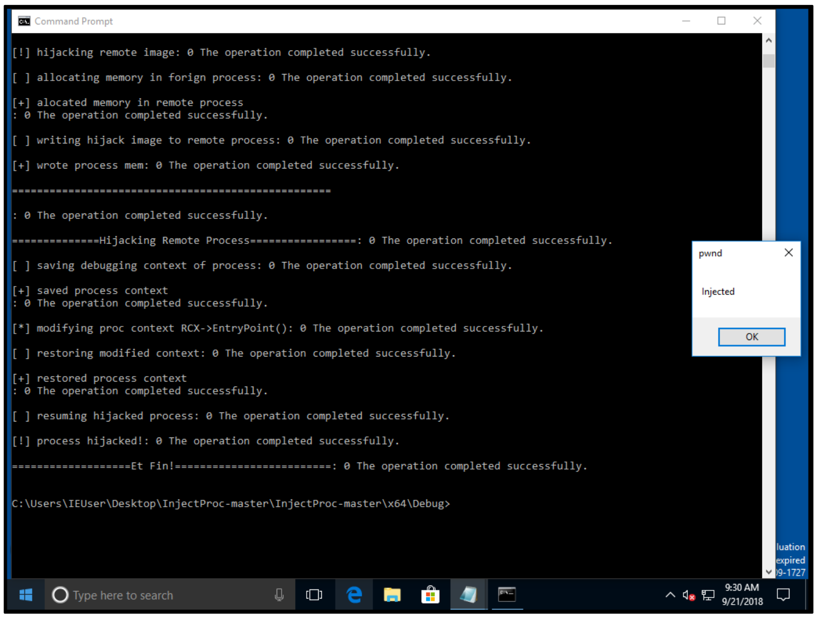
Here it is described how the injection is performed in the Virtual Machine using the source code found in InjectProc project [41]. It is also shown how the information needed for the verification of the methodology is gained. The acquired memory image is called MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem

At first, notepad.exe is opened, which is used as a reference for the injected instance. The command for the injection is shown below, as well the result of the injected mbox.exe (the pop up window).



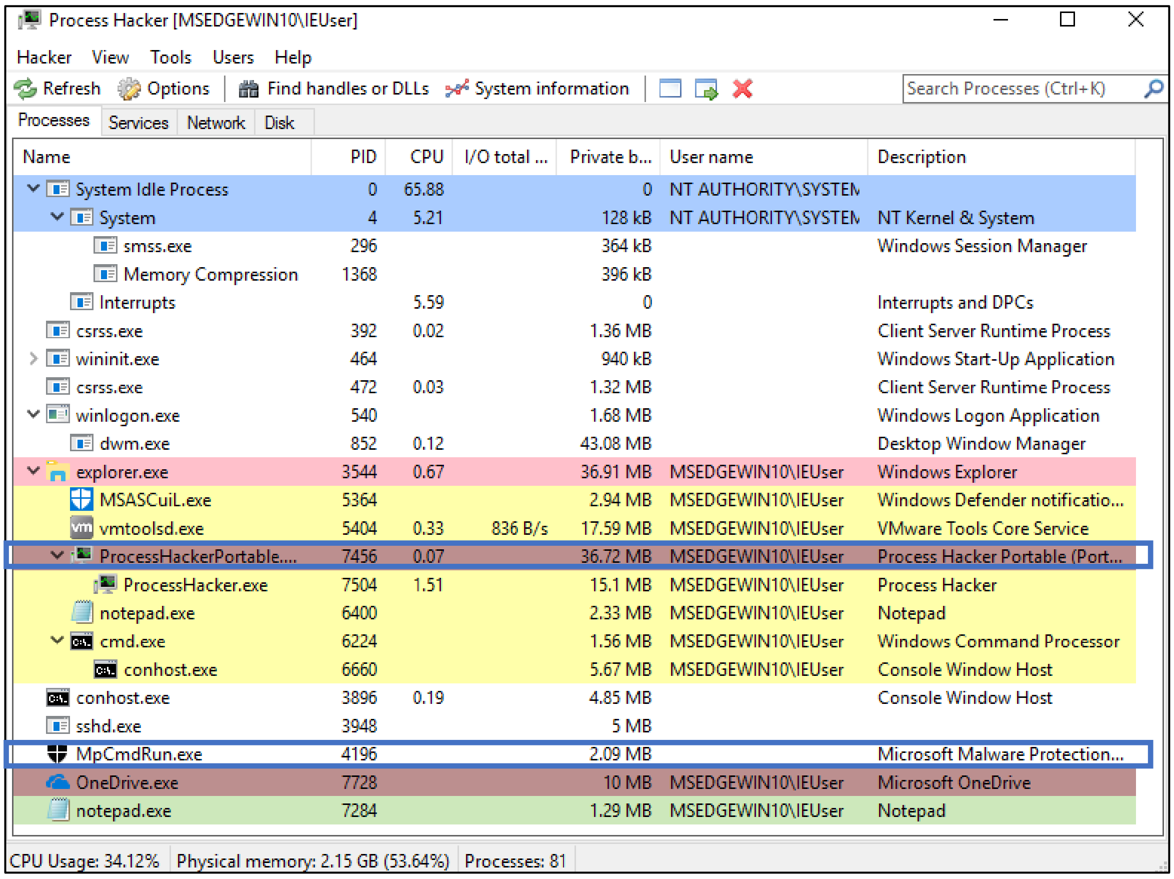
```
C:\Users\IEUser\Desktop\InjectProc-master\InjectProc-master\x64\Debug>
C:\Users\IEUser\Desktop\InjectProc-master\InjectProc-master\x64\Debug>InjectProc.exe proc_rpl C:\Windows\System32\notep
ad.exe C:\Users\IEUser\Desktop\InjectProc-master\InjectProc-master\x64\Debug\mbox.exe
==============Initial Processing==================: 0 The operation completed successfully.

[ ] wstring  Opening Binary to read into buffer: 0 The operation completed successfully.

[+] Opened binary
: 0 The operation completed successfully.

[ ] Parsing PE from buffer: 0 The operation completed successfully.

[+] Got PE info: 0 The operation completed successfully.

====================================================

: 0 The operation completed successfully.

============Creating Process to Infect=============: 0 The operation completed successfully.

[ ]Creating host process: 0 The operation completed successfully.

[+] Created host process: 0 The operation completed successfully.

====================================================

: 0 The operation completed successfully.

============Hijacking Remote Functions==============: 0 The operation completed successfully.

[ ] loading remote process libraries and functions to build new PEB: 0 The operation completed successfully.

[ ] getting ntdll: 0 The operation completed successfully.

[+] got ntdll
: 0 The operation completed successfully.

[ ] getting NtQueryInformationProcess: 0 The operation completed successfully.

[+] got NtQueryInformationProcess
: 0 The operation completed successfully.
```

[snip]



```
[!] hijacking remote image: 0 The operation completed successfully.

[ ] allocating memory in forign process: 0 The operation completed successfully.

[+] alocated memory in remote process
: 0 The operation completed successfully.

[ ] writing hijack image to remote process: 0 The operation completed successfully.

[+] wrote process mem: 0 The operation completed successfully.

====================================================

: 0 The operation completed successfully.

==============Hijacking Remote Process==================: 0 The operation completed successfully.

[ ] saving debugging context of process: 0 The operation completed successfully.

[+] saved process context
: 0 The operation completed successfully.

[*] modifying proc context RCX->EntryPoint(): 0 The operation completed successfully.

[ ] restoring modified context: 0 The operation completed successfully.

[+] restored process context
: 0 The operation completed successfully.

[ ] resuming hijacked process: 0 The operation completed successfully.

[!] process hijacked!: 0 The operation completed successfully.

==================Et Fin!=========================: 0 The operation completed successfully.


C:\Users\IEUser\Desktop\InjectProc-master\InjectProc-master\x64\Debug>
```

Using Process Hacker, the process id of the legitimate notepad.exe (pid 6400) and the injected one (pid 7284) was found out.

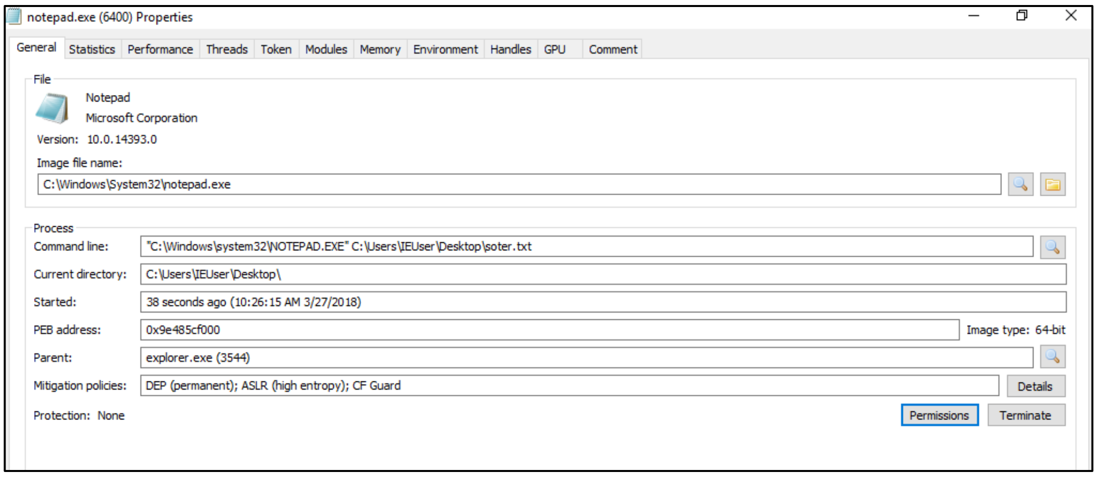

Process Hacker [MSEDGEWIN10\IEUser]

Hacker View Tools Users Help
Refresh Options Find handles or DLLs System information
Search Processes (Ctrl+K)
Processes Services Network Disk

Name	PID	CPU	I/O total ...	Private b...	User name	Description
System Idle Process	0	65.88		0	NT AUTHORITY\SYSTEM	
System	4	5.21		128 kB	NT AUTHORITY\SYSTEM	NT Kernel & System
smss.exe	296			364 kB		Windows Session Manager
Memory Compression	1368			396 kB		
Interrupts		5.59		0		Interrupts and DPCs
csrss.exe	392	0.02		1.36 MB		Client Server Runtime Process
wininit.exe	464			940 kB		Windows Start-Up Application
csrss.exe	472	0.03		1.32 MB		Client Server Runtime Process
winlogon.exe	540			1.68 MB		Windows Logon Application
dwm.exe	852	0.12		43.08 MB		Desktop Window Manager
explorer.exe	3544	0.67		36.91 MB	MSEDGEWIN10\IEUser	Windows Explorer
MSASCuiL.exe	5364			2.94 MB	MSEDGEWIN10\IEUser	Windows Defender notificatio...
vmtoolsd.exe	5404	0.33	836 B/s	17.59 MB	MSEDGEWIN10\IEUser	VMware Tools Core Service
ProcessHackerPortable....	7456	0.07		36.72 MB	MSEDGEWIN10\IEUser	Process Hacker Portable (Port...
ProcessHacker.exe	7504	1.51		15.1 MB	MSEDGEWIN10\IEUser	Process Hacker
notepad.exe	6400			2.33 MB	MSEDGEWIN10\IEUser	Notepad
cmd.exe	6224			1.56 MB	MSEDGEWIN10\IEUser	Windows Command Processor
conhost.exe	6660			5.67 MB	MSEDGEWIN10\IEUser	Console Window Host
conhost.exe	3896	0.19		4.85 MB		Console Window Host
sshd.exe	3948			5 MB		
MpCmdRun.exe	4196			2.09 MB		Microsoft Malware Protection...
OneDrive.exe	7728			10 MB	MSEDGEWIN10\IEUser	Microsoft OneDrive
notepad.exe	7284			1.29 MB	MSEDGEWIN10\IEUser	Notepad

CPU Usage: 34.12% Physical memory: 2.15 GB (53.64%) Processes: 81

The general properties for the legitimate process are:


notepad.exe (6400) Properties

General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment

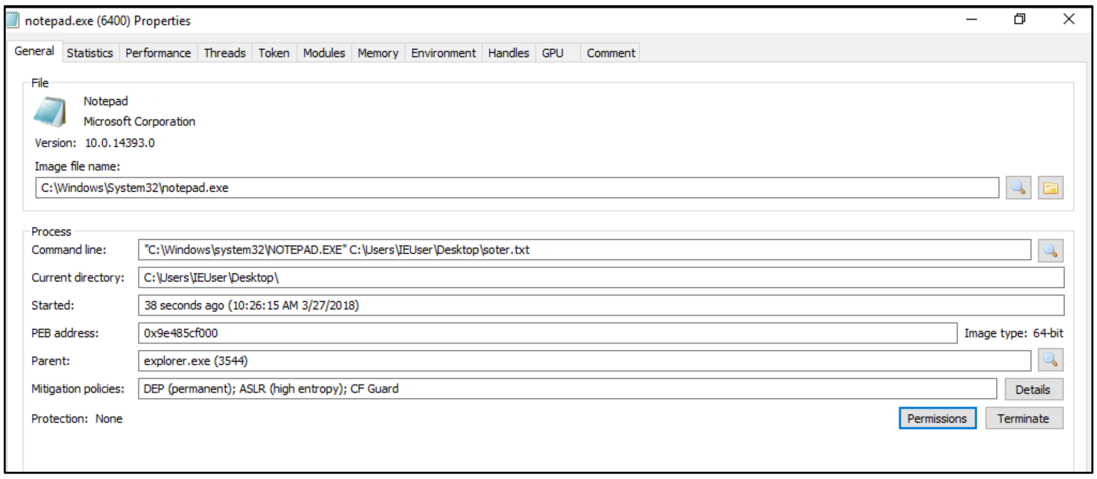
File

Notepad
Microsoft Corporation
Version: 10.0.14393.0
Image file name: C:\Windows\System32\notepad.exe

Process

Command line: "C:\Windows\system32\NOTEPAD.EXE" C:\Users\IEUser\Desktop\soter.txt
Current directory: C:\Users\IEUser\Desktop\
Started: 38 seconds ago (10:26:15 AM 3/27/2018)
PEB address: 0x9e485cf000 Image type: 64-bit
Parent: explorer.exe (3544)
Mitigation policies: DEP (permanent); ASLR (high entropy); CF Guard
Protection: None

and for the malicious one are:



notepad.exe (6400) Properties

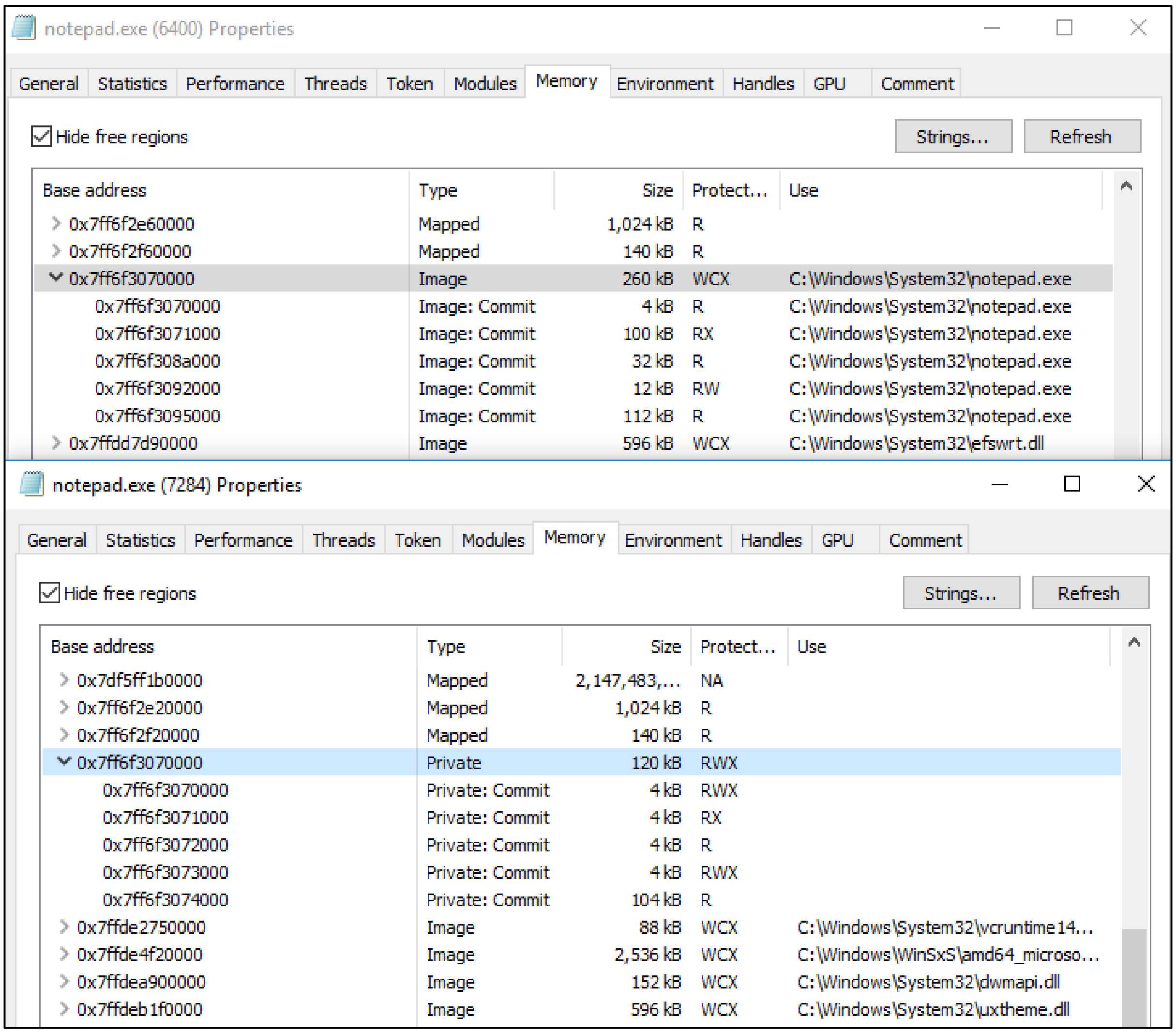
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

File
Notepad
Microsoft Corporation
Version: 10.0.14393.0
Image file name:
C:\Windows\System32\notepad.exe

Process

Command line:	"C:\Windows\system32\NOTEPAD.EXE" C:\Users\IEUser\Desktop\soter.txt
Current directory:	C:\Users\IEUser\Desktop\
Started:	38 seconds ago (10:26:15 AM 3/27/2018)
PEB address:	0x9e485cf000
Image type:	64-bit
Parent:	explorer.exe (3544)
Mitigation policies:	DEP (permanent); ASLR (high entropy); CF Guard
Protection:	None

Both processes use 0x7ff6f3070000 as base address. The different memory characteristics are demonstrated bellow and are consistent with the ones described in Methodology of Detection section.



notepad.exe (6400) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

Hide free regions

Base address	Type	Size	Protect...	Use
0x7ff6f2e60000	Mapped	1,024 kB	R	
0x7ff6f2f60000	Mapped	140 kB	R	
0x7ff6f3070000	Image	260 kB	WCX	C:\Windows\System32\notepad.exe
0x7ff6f3070000	Image: Commit	4 kB	R	C:\Windows\System32\notepad.exe
0x7ff6f3071000	Image: Commit	100 kB	RX	C:\Windows\System32\notepad.exe
0x7ff6f308a000	Image: Commit	32 kB	R	C:\Windows\System32\notepad.exe
0x7ff6f3092000	Image: Commit	12 kB	RW	C:\Windows\System32\notepad.exe
0x7ff6f3095000	Image: Commit	112 kB	R	C:\Windows\System32\notepad.exe
0x7ffdd7d90000	Image	596 kB	WCX	C:\Windows\System32\efswrt.dll

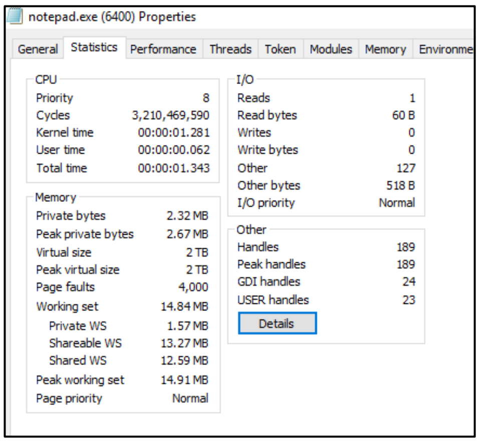
notepad.exe (7284) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

Hide free regions

Base address	Type	Size	Protect...	Use
0x7df5ff1b0000	Mapped	2,147,483,...	NA	
0x7ff6f2e20000	Mapped	1,024 kB	R	
0x7ff6f2f20000	Mapped	140 kB	R	
0x7ff6f3070000	Private	120 kB	RWX	
0x7ff6f3070000	Private: Commit	4 kB	RWX	
0x7ff6f3071000	Private: Commit	4 kB	RX	
0x7ff6f3072000	Private: Commit	4 kB	R	
0x7ff6f3073000	Private: Commit	4 kB	RWX	
0x7ff6f3074000	Private: Commit	104 kB	R	
0x7ffde2750000	Image	88 kB	WCX	C:\Windows\System32\vcruntime14...
0x7ffde4f20000	Image	2,536 kB	WCX	C:\Windows\WinSxS\amd64_microso...
0x7ffdea900000	Image	152 kB	WCX	C:\Windows\System32\dwmapi.dll
0x7ffdeb1f0000	Image	596 kB	WCX	C:\Windows\System32\uxtheme.dll

The other characteristics such as priority and environment variables, are similar:



notepad.exe (6400) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environme

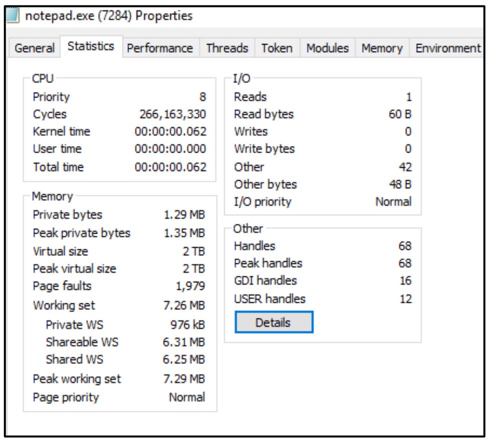
CPU	
Priority	8
Cycles	3,210,469,590
Kernel time	00:00:01.281
User time	00:00:00.062
Total time	00:00:01.343

Memory	
Private bytes	2.32 MB
Peak private bytes	2.67 MB
Virtual size	2 TB
Peak virtual size	2 TB
Page faults	4,000
Working set	14.84 MB
Private WS	1.57 MB
Shareable WS	13.27 MB
Shared WS	12.59 MB
Peak working set	14.91 MB
Page priority	Normal

I/O	
Reads	1
Read bytes	60 B
Writes	0
Write bytes	0
Other	127
Other bytes	518 B
I/O priority	Normal

Other	
Handles	189
Peak handles	189
GDI handles	24
USER handles	23

Details



notepad.exe (7284) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment

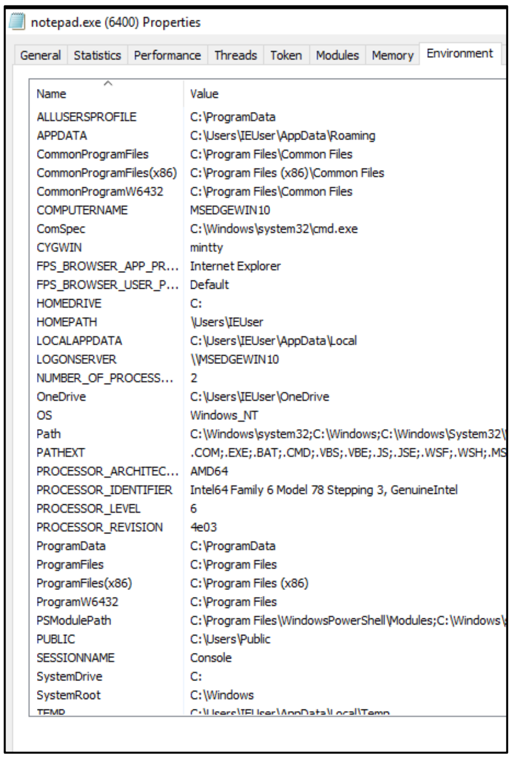
CPU	
Priority	8
Cycles	266,163,330
Kernel time	00:00:00.062
User time	00:00:00.000
Total time	00:00:00.062

Memory	
Private bytes	1.29 MB
Peak private bytes	1.35 MB
Virtual size	2 TB
Peak virtual size	2 TB
Page faults	1,979
Working set	7.26 MB
Private WS	976 kB
Shareable WS	6.31 MB
Shared WS	6.25 MB
Peak working set	7.29 MB
Page priority	Normal

I/O	
Reads	1
Read bytes	60 B
Writes	0
Write bytes	0
Other	42
Other bytes	48 B
I/O priority	Normal

Other	
Handles	68
Peak handles	68
GDI handles	16
USER handles	12

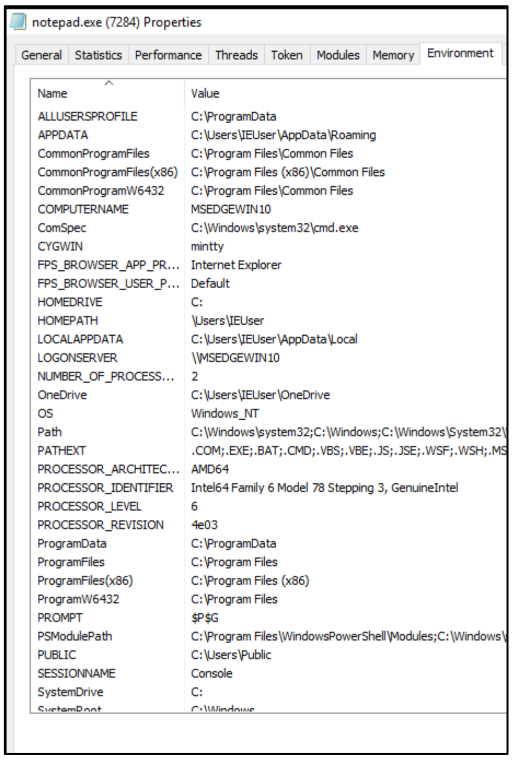
Details



notepad.exe (6400) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment

Name	Value
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\IEUser\AppData\Roaming
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	MSEDGEWIN10
ComSpec	C:\Windows\system32\cmd.exe
CYGWIN	mintty
FPS_BROWSER_APP_PR…	Internet Explorer
FPS_BROWSER_USER_P…	Default
HOMEDRIVE	C:
HOMEPATH	\Users\IEUser
LOCALAPPDATA	C:\Users\IEUser\AppData\Local
LOGONSERVER	\\MSEDGEWIN10
NUMBER_OF_PROCESS…	2
OneDrive	C:\Users\IEUser\OneDrive
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\Windows\System32\
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MS
PROCESSOR_ARCHITEC…	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 78 Stepping 3, GenuineIntel
PROCESSOR_LEVEL	6
PROCESSOR_REVISION	4e03
ProgramData	C:\ProgramData
ProgramFiles	C:\Program Files
ProgramFiles(x86)	C:\Program Files (x86)
ProgramW6432	C:\Program Files
PSModulePath	C:\Program Files\WindowsPowerShell\Modules;C:\Windows\
PUBLIC	C:\Users\Public
SESSIONNAME	Console
SystemDrive	C:
SystemRoot	C:\Windows



notepad.exe (7284) Properties

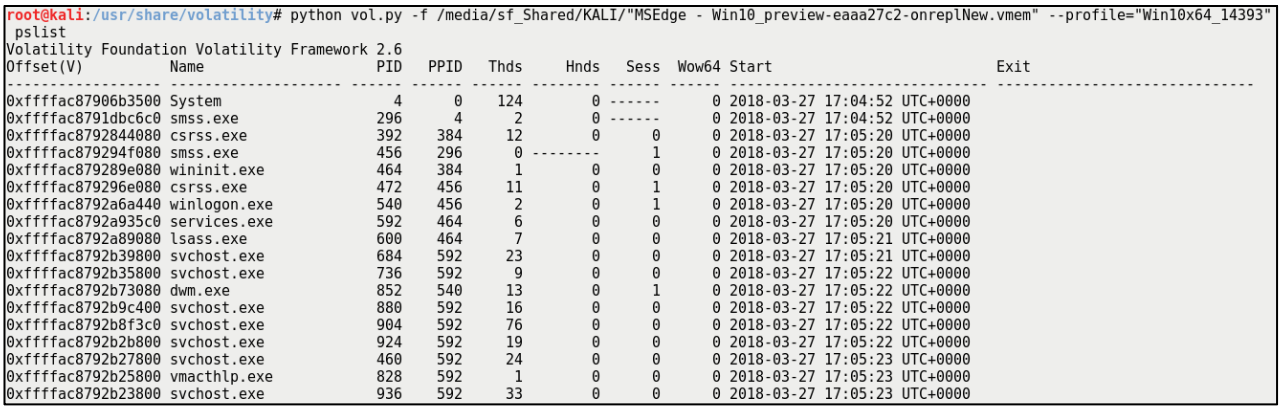
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment

Name	Value
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\IEUser\AppData\Roaming
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	MSEDGEWIN10
ComSpec	C:\Windows\system32\cmd.exe
CYGWIN	mintty
FPS_BROWSER_APP_PR…	Internet Explorer
FPS_BROWSER_USER_P…	Default
HOMEDRIVE	C:
HOMEPATH	\Users\IEUser
LOCALAPPDATA	C:\Users\IEUser\AppData\Local
LOGONSERVER	\\MSEDGEWIN10
NUMBER_OF_PROCESS…	2
OneDrive	C:\Users\IEUser\OneDrive
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\Windows\System32\
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MS
PROCESSOR_ARCHITEC…	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 78 Stepping 3, GenuineIntel
PROCESSOR_LEVEL	6
PROCESSOR_REVISION	4e03
ProgramData	C:\ProgramData
ProgramFiles	C:\Program Files
ProgramFiles(x86)	C:\Program Files (x86)
ProgramW6432	C:\Program Files
PROMPT	PG
PSModulePath	C:\Program Files\WindowsPowerShell\Modules;C:\Windows\
PUBLIC	C:\Users\Public
SESSIONNAME	Console
SystemDrive	C:

6.1.2 Injection Detection on InjectProc memory image

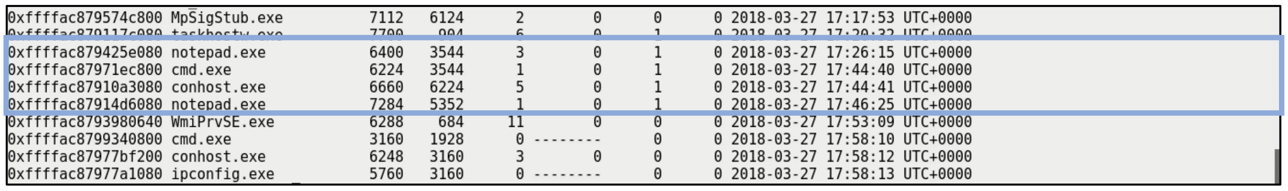
After acquisition of the memory image MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem, described in the previous paragraph, it follows the injection detection. Following the steps described in Methodology of Detection, the results are:

Process Listing



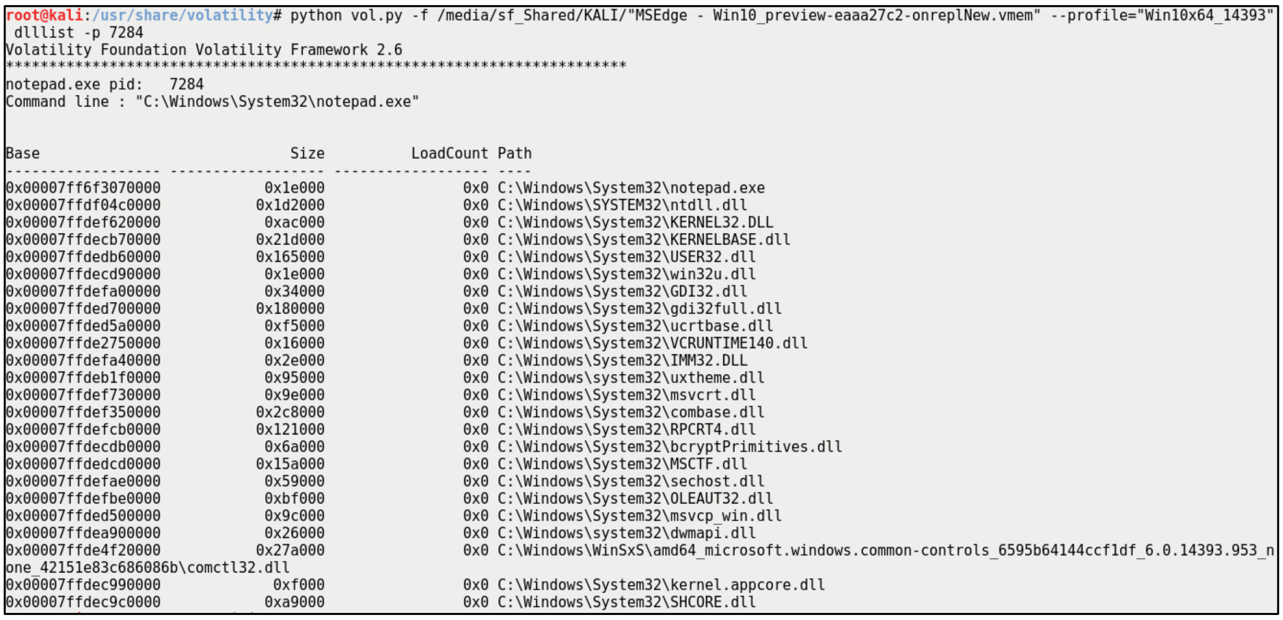
```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
------------------ -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ------------------------------
0xffffac87906b3500 System                    4      0    124        0 ------      0 2018-03-27 17:04:52 UTC+0000
0xffffac8791dbc6c0 smss.exe                296      4      2        0 ------      0 2018-03-27 17:04:52 UTC+0000
0xffffac8792844080 csrss.exe               392    384     12        0      0      0 2018-03-27 17:05:20 UTC+0000
0xffffac879294f080 smss.exe                456    296      0 --------      1      0 2018-03-27 17:05:20 UTC+0000
0xffffac879289e080 wininit.exe             464    384      1        0      0      0 2018-03-27 17:05:20 UTC+0000
0xffffac879296e080 csrss.exe               472    456     11        0      1      0 2018-03-27 17:05:20 UTC+0000
0xffffac8792a6a440 winlogon.exe            540    456      2        0      1      0 2018-03-27 17:05:20 UTC+0000
0xffffac8792a935c0 services.exe            592    464      6        0      0      0 2018-03-27 17:05:20 UTC+0000
0xffffac8792a89080 lsass.exe               600    464      7        0      0      0 2018-03-27 17:05:21 UTC+0000
0xffffac8792b39800 svchost.exe             684    592     23        0      0      0 2018-03-27 17:05:21 UTC+0000
0xffffac8792b35800 svchost.exe             736    592      9        0      0      0 2018-03-27 17:05:22 UTC+0000
0xffffac8792b73080 dwm.exe                 852    540     13        0      1      0 2018-03-27 17:05:22 UTC+0000
0xffffac8792b9c400 svchost.exe             880    592     16        0      0      0 2018-03-27 17:05:22 UTC+0000
0xffffac8792b8f3c0 svchost.exe             904    592     76        0      0      0 2018-03-27 17:05:22 UTC+0000
0xffffac8792b2b800 svchost.exe             924    592     19        0      0      0 2018-03-27 17:05:22 UTC+0000
0xffffac8792b27800 svchost.exe             460    592     24        0      0      0 2018-03-27 17:05:23 UTC+0000
0xffffac8792b25800 vmacthlp.exe            828    592      1        0      0      0 2018-03-27 17:05:23 UTC+0000
0xffffac8792b23800 svchost.exe             936    592     33        0      0      0 2018-03-27 17:05:23 UTC+0000
```

[snip]



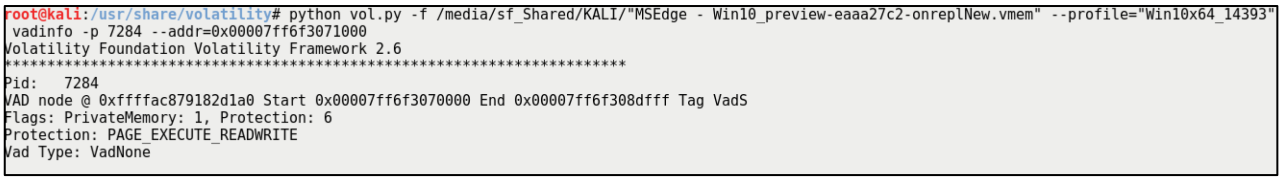
```
0xffffac879574c800 MpSigStub.exe          7112   6124      2        0      0      0 2018-03-27 17:17:53 UTC+0000
0xffffac879117c080 taskhostw.exe          7700    904      6        0      1      0 2018-03-27 17:20:32 UTC+0000
0xffffac879425e080 notepad.exe            6400   3544      3        0      1      0 2018-03-27 17:26:15 UTC+0000
0xffffac87971ec800 cmd.exe                6224   3544      1        0      1      0 2018-03-27 17:44:40 UTC+0000
0xffffac87910a3080 conhost.exe            6660   6224      5        0      1      0 2018-03-27 17:44:41 UTC+0000
0xffffac87914d6080 notepad.exe            7284   5352      1        0      1      0 2018-03-27 17:46:25 UTC+0000
0xffffac8793980640 WmiPrvSE.exe           6288    684     11        0      0      0 2018-03-27 17:53:09 UTC+0000
0xffffac8799340800 cmd.exe                3160   1928      0 --------      0      0 2018-03-27 17:58:10 UTC+0000
0xffffac87977bf200 conhost.exe            6248   3160      3        0      0      0 2018-03-27 17:58:12 UTC+0000
0xffffac87977a1080 ipconfig.exe           5760   3160      0 --------      0      0 2018-03-27 17:58:13 UTC+0000
```

Dynamic Link Library Listing



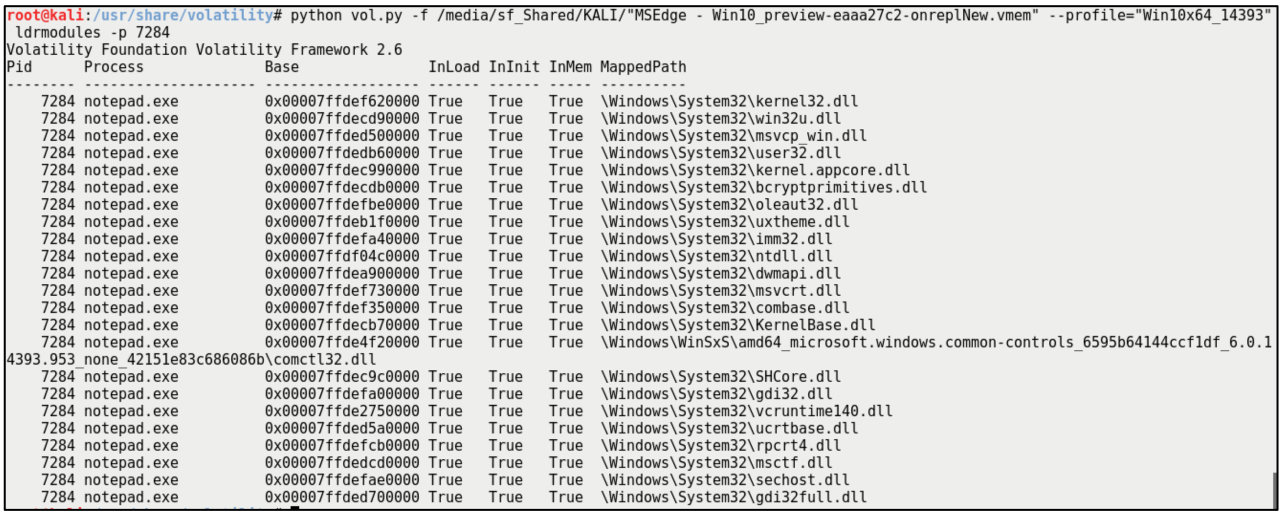
```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 dlllist -p 7284
Volatility Foundation Volatility Framework 2.6
************************************************************************
notepad.exe pid:   7284
Command line : "C:\Windows\System32\notepad.exe"


Base                             Size          LoadCount Path
------------------ ------------------ ------------------ ----
0x00007ff6f3070000            0x1e000                0x0 C:\Windows\System32\notepad.exe
0x00007ffdf04c0000           0x1d2000                0x0 C:\Windows\SYSTEM32\ntdll.dll
0x00007ffdef620000            0xac000                0x0 C:\Windows\System32\KERNEL32.DLL
0x00007ffdecb70000           0x21d000                0x0 C:\Windows\System32\KERNELBASE.dll
0x00007ffdedb60000           0x165000                0x0 C:\Windows\System32\USER32.dll
0x00007ffdecd90000            0x1e000                0x0 C:\Windows\System32\win32u.dll
0x00007ffdefa00000            0x34000                0x0 C:\Windows\System32\GDI32.dll
0x00007ffded700000           0x180000                0x0 C:\Windows\System32\gdi32full.dll
0x00007ffded5a0000            0xf5000                0x0 C:\Windows\System32\ucrtbase.dll
0x00007ffde2750000            0x16000                0x0 C:\Windows\System32\VCRUNTIME140.dll
0x00007ffdefa40000            0x2e000                0x0 C:\Windows\System32\IMM32.DLL
0x00007ffdeb1f0000            0x95000                0x0 C:\Windows\system32\uxtheme.dll
0x00007ffdef730000            0x9e000                0x0 C:\Windows\System32\msvcrt.dll
0x00007ffdef350000           0x2c8000                0x0 C:\Windows\System32\combase.dll
0x00007ffdefcb0000           0x121000                0x0 C:\Windows\System32\RPCRT4.dll
0x00007ffdecdb0000            0x6a000                0x0 C:\Windows\System32\bcryptPrimitives.dll
0x00007ffdedcd0000           0x15a000                0x0 C:\Windows\System32\MSCTF.dll
0x00007ffdefae0000            0x59000                0x0 C:\Windows\System32\sechost.dll
0x00007ffdefbe0000            0xbf000                0x0 C:\Windows\System32\OLEAUT32.dll
0x00007ffded500000            0x9c000                0x0 C:\Windows\System32\msvcp_win.dll
0x00007ffdea900000            0x26000                0x0 C:\Windows\system32\dwmapi.dll
0x00007ffde4f20000           0x27a000                0x0 C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.14393.953_n
one_42151e83c686086b\comctl32.dll
0x00007ffdec990000             0xf000                0x0 C:\Windows\System32\kernel.appcore.dll
0x00007ffdec9c0000             0xa9000                0x0 C:\Windows\System32\SHCORE.dll
```

Virtual Address Descriptor information



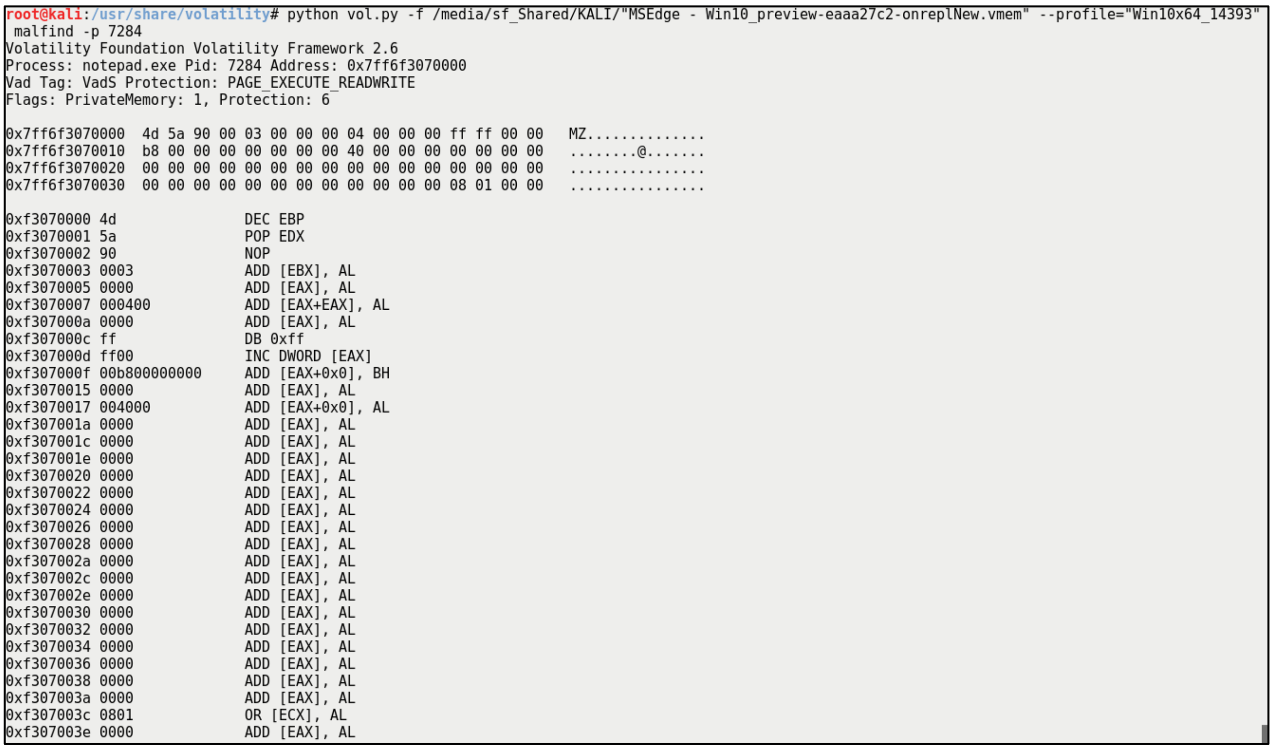
```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 vadinfo -p 7284 --addr=0x00007ff6f3071000
Volatility Foundation Volatility Framework 2.6
************************************************************************
Pid:   7284
VAD node @ 0xffffac879182d1a0 Start 0x00007ff6f3070000 End 0x00007ff6f308dfff Tag VadS
Flags: PrivateMemory: 1, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
Vad Type: VadNone
```

Module linked lists and VAD



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 ldrmodules -p 7284
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    7284 notepad.exe          0x00007ffdef620000 True   True   True  \Windows\System32\kernel32.dll
    7284 notepad.exe          0x00007ffdecd90000 True   True   True  \Windows\System32\win32u.dll
    7284 notepad.exe          0x00007ffded500000 True   True   True  \Windows\System32\msvcp_win.dll
    7284 notepad.exe          0x00007ffdedb60000 True   True   True  \Windows\System32\user32.dll
    7284 notepad.exe          0x00007ffdec990000 True   True   True  \Windows\System32\kernel.appcore.dll
    7284 notepad.exe          0x00007ffdecdb0000 True   True   True  \Windows\System32\bcryptprimitives.dll
    7284 notepad.exe          0x00007ffdefbe0000 True   True   True  \Windows\System32\oleaut32.dll
    7284 notepad.exe          0x00007ffdeb1f0000 True   True   True  \Windows\System32\uxtheme.dll
    7284 notepad.exe          0x00007ffdefa40000 True   True   True  \Windows\System32\imm32.dll
    7284 notepad.exe          0x00007ffdf04c0000 True   True   True  \Windows\System32\ntdll.dll
    7284 notepad.exe          0x00007ffdea900000 True   True   True  \Windows\System32\dwmapi.dll
    7284 notepad.exe          0x00007ffdef730000 True   True   True  \Windows\System32\msvcrt.dll
    7284 notepad.exe          0x00007ffdef350000 True   True   True  \Windows\System32\combase.dll
    7284 notepad.exe          0x00007ffdecb70000 True   True   True  \Windows\System32\KernelBase.dll
    7284 notepad.exe          0x00007ffde4f20000 True   True   True  \Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.1
4393.953_none_42151e83c686086b\comctl32.dll
    7284 notepad.exe          0x00007ffdec9c0000 True   True   True  \Windows\System32\SHCore.dll
    7284 notepad.exe          0x00007ffdefa00000 True   True   True  \Windows\System32\gdi32.dll
    7284 notepad.exe          0x00007ffde2750000 True   True   True  \Windows\System32\vcruntime140.dll
    7284 notepad.exe          0x00007ffded5a0000 True   True   True  \Windows\System32\ucrtbase.dll
    7284 notepad.exe          0x00007ffdefcb0000 True   True   True  \Windows\System32\rpcrt4.dll
    7284 notepad.exe          0x00007ffdedcd0000 True   True   True  \Windows\System32\msctf.dll
    7284 notepad.exe          0x00007ffdefae0000 True   True   True  \Windows\System32\sechost.dll
    7284 notepad.exe          0x00007ffded700000 True   True   True  \Windows\System32\gdi32full.dll
```

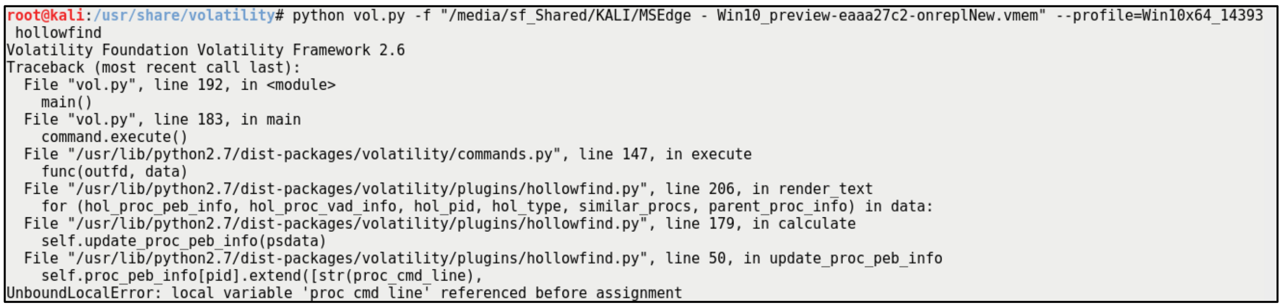
Checking memory permissions



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 malfind -p 7284
Volatility Foundation Volatility Framework 2.6
Process: notepad.exe Pid: 7284 Address: 0x7ff6f3070000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

0x7ff6f3070000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff6f3070010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff6f3070020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff6f3070030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

0xf3070000 4d               DEC EBP
0xf3070001 5a               POP EDX
0xf3070002 90               NOP
0xf3070003 0003             ADD [EBX], AL
0xf3070005 0000             ADD [EAX], AL
0xf3070007 000400           ADD [EAX+EAX], AL
0xf307000a 0000             ADD [EAX], AL
0xf307000c ff               DB 0xff
0xf307000d ff00             INC DWORD [EAX]
0xf307000f 00b800000000     ADD [EAX+0x0], BH
0xf3070015 0000             ADD [EAX], AL
0xf3070017 004000           ADD [EAX+0x0], AL
0xf307001a 0000             ADD [EAX], AL
0xf307001c 0000             ADD [EAX], AL
0xf307001e 0000             ADD [EAX], AL
0xf3070020 0000             ADD [EAX], AL
0xf3070022 0000             ADD [EAX], AL
0xf3070024 0000             ADD [EAX], AL
0xf3070026 0000             ADD [EAX], AL
0xf3070028 0000             ADD [EAX], AL
0xf307002a 0000             ADD [EAX], AL
0xf307002c 0000             ADD [EAX], AL
0xf307002e 0000             ADD [EAX], AL
0xf3070030 0000             ADD [EAX], AL
0xf3070032 0000             ADD [EAX], AL
0xf3070034 0000             ADD [EAX], AL
0xf3070036 0000             ADD [EAX], AL
0xf3070038 0000             ADD [EAX], AL
0xf307003a 0000             ADD [EAX], AL
0xf307003c 0801             OR [ECX], AL
0xf307003e 0000             ADD [EAX], AL
```

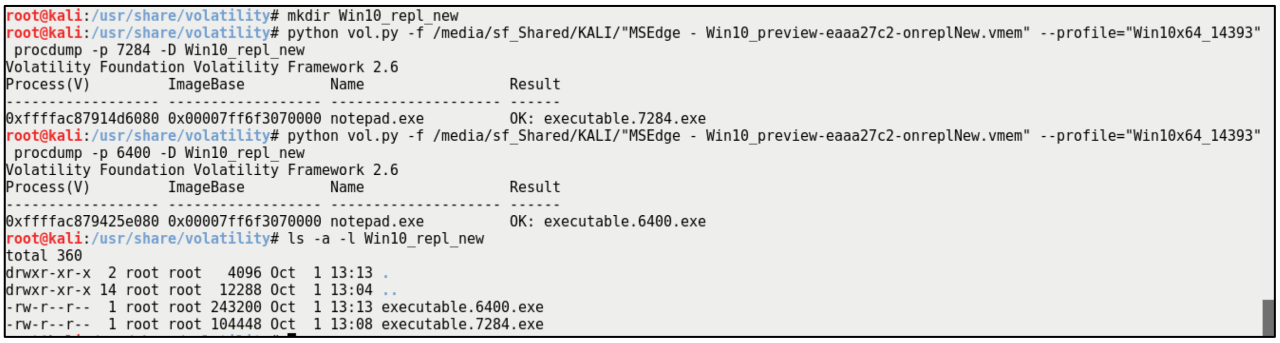
Hollowfind plugin

In this step, it seems to be a problem with hollowfind plugin.



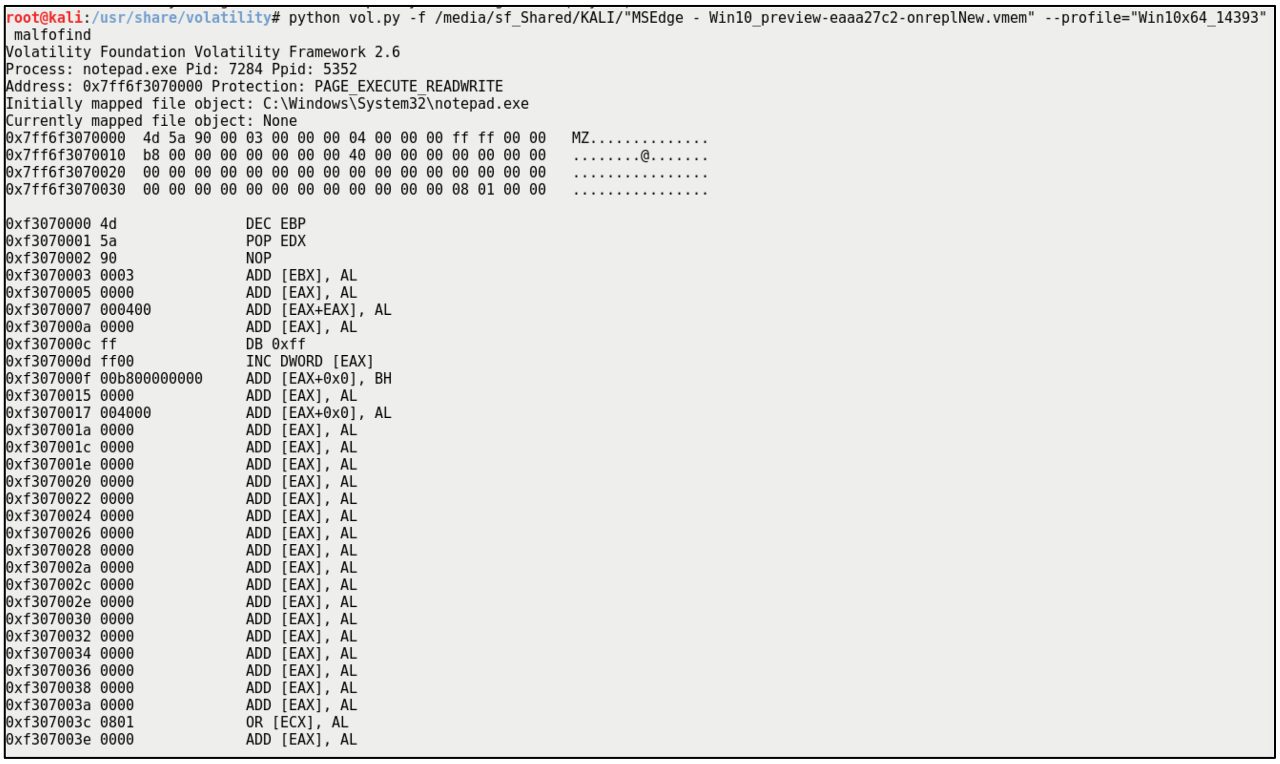
```
root@kali:/usr/share/volatility# python vol.py -f "/media/sf_Shared/KALI/MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile=Win10x64_14393
 hollowfind
Volatility Foundation Volatility Framework 2.6
Traceback (most recent call last):
  File "vol.py", line 192, in <module>
    main()
  File "vol.py", line 183, in main
    command.execute()
  File "/usr/lib/python2.7/dist-packages/volatility/commands.py", line 147, in execute
    func(outfd, data)
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/hollowfind.py", line 206, in render_text
    for (hol_proc_peb_info, hol_proc_vad_info, hol_pid, hol_type, similar_procs, parent_proc_info) in data:
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/hollowfind.py", line 179, in calculate
    self.update_proc_peb_info(psdata)
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/hollowfind.py", line 50, in update_proc_peb_info
    self.proc_peb_info[pid].extend([str(proc_cmd_line),
UnboundLocalError: local variable 'proc_cmd_line' referenced before assignment
```

Extracting executables, Comparing executables sizes



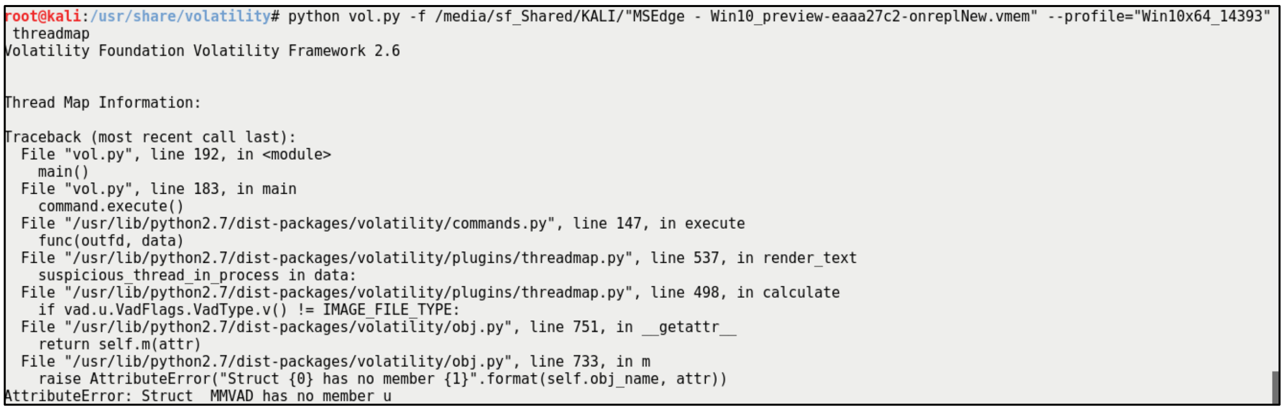
```
root@kali:/usr/share/volatility# mkdir Win10_repl_new
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 procdump -p 7284 -D Win10_repl_new
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffffac87914d6080 0x00007ff6f3070000 notepad.exe          OK: executable.7284.exe
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 procdump -p 6400 -D Win10_repl_new
Volatility Foundation Volatility Framework 2.6
Process(V)         ImageBase          Name                 Result
------------------ ------------------ -------------------- ------
0xffffac879425e080 0x00007ff6f3070000 notepad.exe          OK: executable.6400.exe
root@kali:/usr/share/volatility# ls -a -l Win10_repl_new
total 360
drwxr-xr-x  2 root root   4096 Oct  1 13:13 .
drwxr-xr-x 14 root root  12288 Oct  1 13:04 ..
-rw-r--r--  1 root root 243200 Oct  1 13:13 executable.6400.exe
-rw-r--r--  1 root root 104448 Oct  1 13:08 executable.7284.exe
```

Malfofind



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 malfofind
Volatility Foundation Volatility Framework 2.6
Process: notepad.exe Pid: 7284 Ppid: 5352
Address: 0x7ff6f3070000 Protection: PAGE_EXECUTE_READWRITE
Initially mapped file object: C:\Windows\System32\notepad.exe
Currently mapped file object: None
0x7ff6f3070000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x7ff6f3070010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x7ff6f3070020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7ff6f3070030  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00   ................

0xf3070000 4d               DEC EBP
0xf3070001 5a               POP EDX
0xf3070002 90               NOP
0xf3070003 0003             ADD [EBX], AL
0xf3070005 0000             ADD [EAX], AL
0xf3070007 000400           ADD [EAX+EAX], AL
0xf307000a 0000             ADD [EAX], AL
0xf307000c ff               DB 0xff
0xf307000d ff00             INC DWORD [EAX]
0xf307000f 00b800000000     ADD [EAX+0x0], BH
0xf3070015 0000             ADD [EAX], AL
0xf3070017 004000           ADD [EAX+0x0], AL
0xf307001a 0000             ADD [EAX], AL
0xf307001c 0000             ADD [EAX], AL
0xf307001e 0000             ADD [EAX], AL
0xf3070020 0000             ADD [EAX], AL
0xf3070022 0000             ADD [EAX], AL
0xf3070024 0000             ADD [EAX], AL
0xf3070026 0000             ADD [EAX], AL
0xf3070028 0000             ADD [EAX], AL
0xf307002a 0000             ADD [EAX], AL
0xf307002c 0000             ADD [EAX], AL
0xf307002e 0000             ADD [EAX], AL
0xf3070030 0000             ADD [EAX], AL
0xf3070032 0000             ADD [EAX], AL
0xf3070034 0000             ADD [EAX], AL
0xf3070036 0000             ADD [EAX], AL
0xf3070038 0000             ADD [EAX], AL
0xf307003a 0000             ADD [EAX], AL
0xf307003c 0801             OR [ECX], AL
0xf307003e 0000             ADD [EAX], AL
```

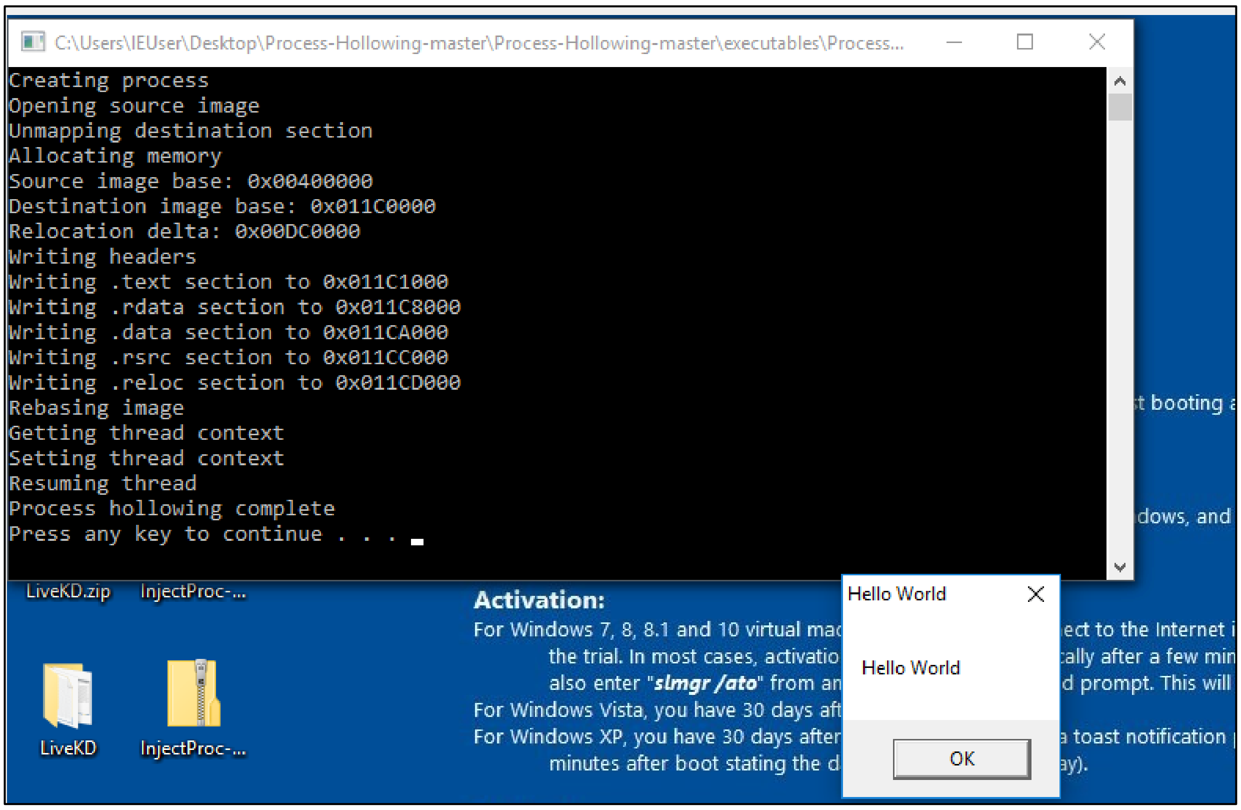
Threadmap plugin



```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem" --profile="Win10x64_14393"
 threadmap
Volatility Foundation Volatility Framework 2.6


Thread Map Information:

Traceback (most recent call last):
  File "vol.py", line 192, in <module>
    main()
  File "vol.py", line 183, in main
    command.execute()
  File "/usr/lib/python2.7/dist-packages/volatility/commands.py", line 147, in execute
    func(outfd, data)
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 537, in render_text
    suspicious_thread_in_process in data:
  File "/usr/lib/python2.7/dist-packages/volatility/plugins/threadmap.py", line 498, in calculate
    if vad.u.VadFlags.VadType.v() != IMAGE_FILE_TYPE:
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 751, in __getattr__
    return self.m(attr)
  File "/usr/lib/python2.7/dist-packages/volatility/obj.py", line 733, in m
    raise AttributeError("Struct {0} has no member {1}".format(self.obj_name, attr))
AttributeError: Struct _MMVAD has no member u
```

6.1.3 Injection Using Process Hollow project

The following screenshot shows how Process Hollow projects' executable is used and its effect.



```
C:\Users\IEUser\Desktop\Process-Hollowing-master\Process-Hollowing-master\executables\Process...
Creating process
Opening source image
Unmapping destination section
Allocating memory
Source image base: 0x00400000
Destination image base: 0x011C0000
Relocation delta: 0x00DC0000
Writing headers
Writing .text section to 0x011C1000
Writing .rdata section to 0x011C8000
Writing .data section to 0x011CA000
Writing .rsrc section to 0x011CC000
Writing .reloc section to 0x011CD000
Rebasing image
Getting thread context
Setting thread context
Resuming thread
Process hollowing complete
Press any key to continue . . .
```

Hello World

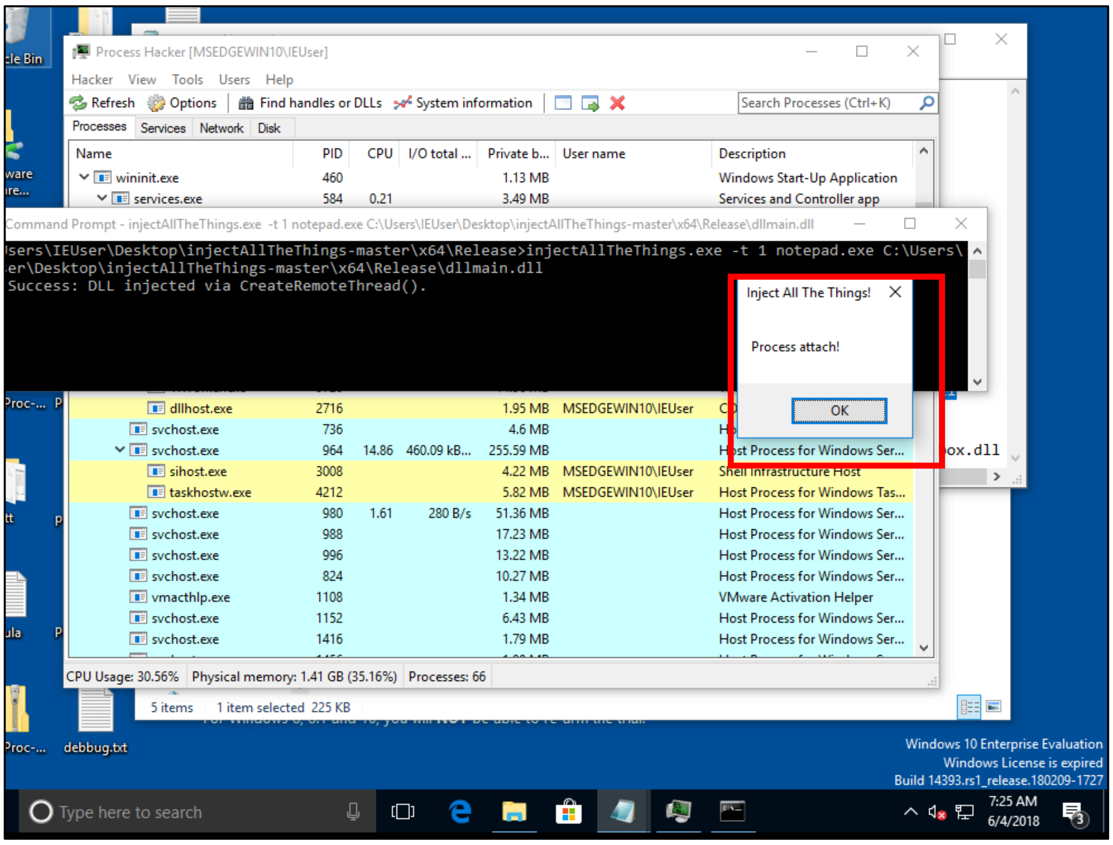
Hello World

6.2 A DLL injection testing example

6.2.1 Gathering useful information from Virtual Machines

This paragraph shows how useful information in a running virtual machine was retrieved to check that the code works correctly, as described in Testing Environment. The corresponding memory image taken is InjectAllThings_before_ok.vmem.

The following image shows how the malicious process is called (through command prompt window) and that the corresponding window (InjectAll The Things) is shown. The injected process is notepad.exe and the loaded DLL is dllmain.dll



Process Hacker [MSEDGEWIN10\IEUser]

Command Prompt - injectAllTheThings.exe -t 1 notepad.exe C:\Users\IEUser\Desktop\injectAllTheThings-master\x64\Release\dllmain.dll

```
Jsers\IEUser\Desktop\injectAllTheThings-master\x64\Release>injectAllTheThings.exe -t 1 notepad.exe C:\Users\
er\Desktop\injectAllTheThings-master\x64\Release\dllmain.dll
 Success: DLL injected via CreateRemoteThread().
```

Inject All The Things!

Process attach!

Name	PID	CPU	I/O total ...	Private b...	User name	Description
wininit.exe	460			1.13 MB		Windows Start-Up Application
services.exe	584	0.21		3.49 MB		Services and Controller app
dllhost.exe	2716			1.95 MB	MSEDGEWIN10\IEUser	C
svchost.exe	736			4.6 MB		H
svchost.exe	964	14.86	460.09 kB...	255.59 MB		Host Process for Windows Ser...
sihost.exe	3008			4.22 MB	MSEDGEWIN10\IEUser	Shell Infrastructure Host
taskhostw.exe	4212			5.82 MB	MSEDGEWIN10\IEUser	Host Process for Windows Tas...
svchost.exe	980	1.61	280 B/s	51.36 MB		Host Process for Windows Ser...
svchost.exe	988			17.23 MB		Host Process for Windows Ser...
svchost.exe	996			13.22 MB		Host Process for Windows Ser...
svchost.exe	824			10.27 MB		Host Process for Windows Ser...
vmacthlp.exe	1108			1.34 MB		VMware Activation Helper
svchost.exe	1152			6.43 MB		Host Process for Windows Ser...
svchost.exe	1416			1.79 MB		Host Process for Windows Ser...

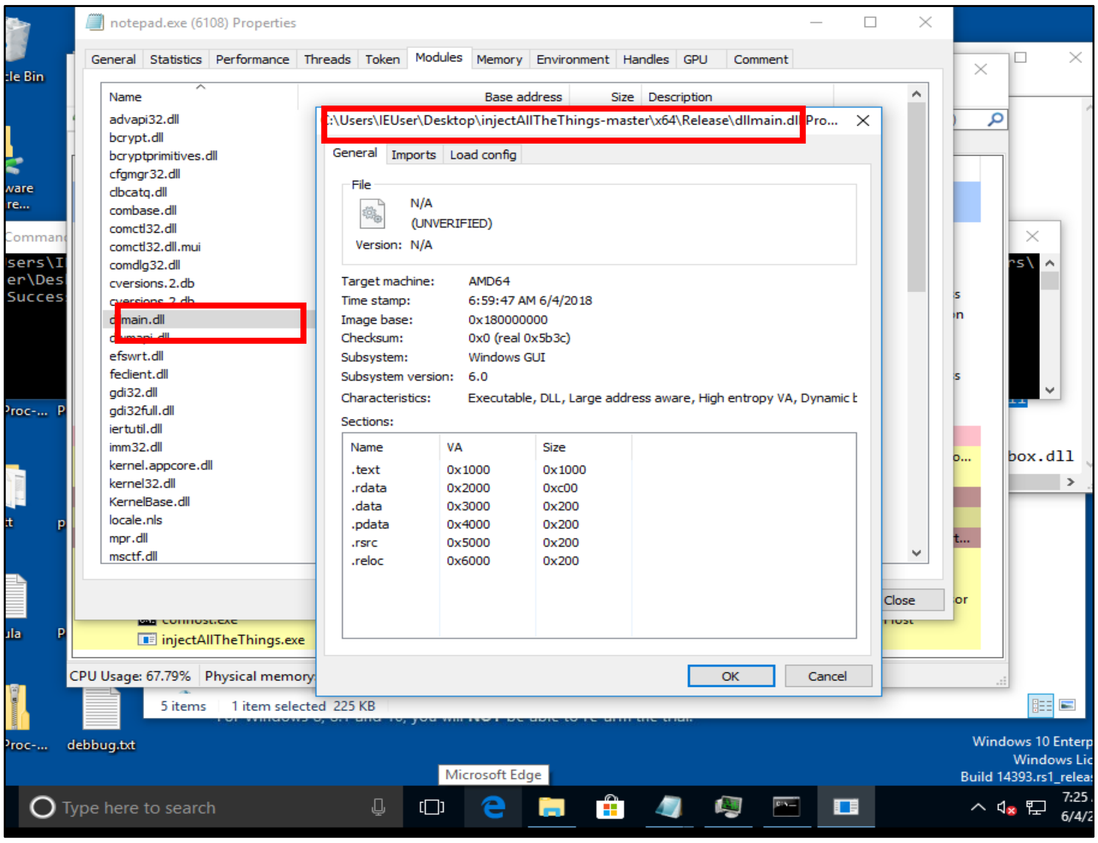
CPU Usage: 30.56% | Physical memory: 1.41 GB (35.16%) | Processes: 66

5 items 1 item selected 225 KB

Windows 10 Enterprise Evaluation
Windows License is expired
Build 14393.rs1_release.180209-1727

7:25 AM
6/4/2018

First, it was confirmed that the "malicious" DLL was loaded in process space (ProcessHacker software is used). The following picture shows main.dll properties, inside notepad process, process id 6108.



notepad.exe (6108) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

```
Name                    Base address   Size   Description
advapi32.dll
bcrypt.dll
bcryptprimitives.dll
cfgmgr32.dll
dbcatq.dll
combase.dll
comctl32.dll
comctl32.dll.mui
comdlg32.dll
cversions.2.db
cversions.2.db
dllmain.dll
dxmapi.dll
efswrt.dll
fcclient.dll
gdi32.dll
gdi32full.dll
iertutil.dll
imm32.dll
kernel.appcore.dll
kernel32.dll
KernelBase.dll
locale.nls
mpr.dll
msctf.dll
```

:\Users\IEUser\Desktop\injectAllTheThings-master\x64\Release\dllmain.dll Pro...

General | Imports | Load config

```
File
    N/A
    (UNVERIFIED)
Version:  N/A

Target machine:      AMD64
Time stamp:          6:59:47 AM 6/4/2018
Image base:          0x180000000
Checksum:            0x0 (real 0x5b3c)
Subsystem:           Windows GUI
Subsystem version:   6.0
Characteristics:     Executable, DLL, Large address aware, High entropy VA, Dynamic b

Sections:
Name      VA        Size
.text     0x1000    0x1000
.rdata    0x2000    0xc00
.data     0x3000    0x200
.pdata    0x4000    0x200
.rsrc     0x5000    0x200
.reloc    0x6000    0x200
```

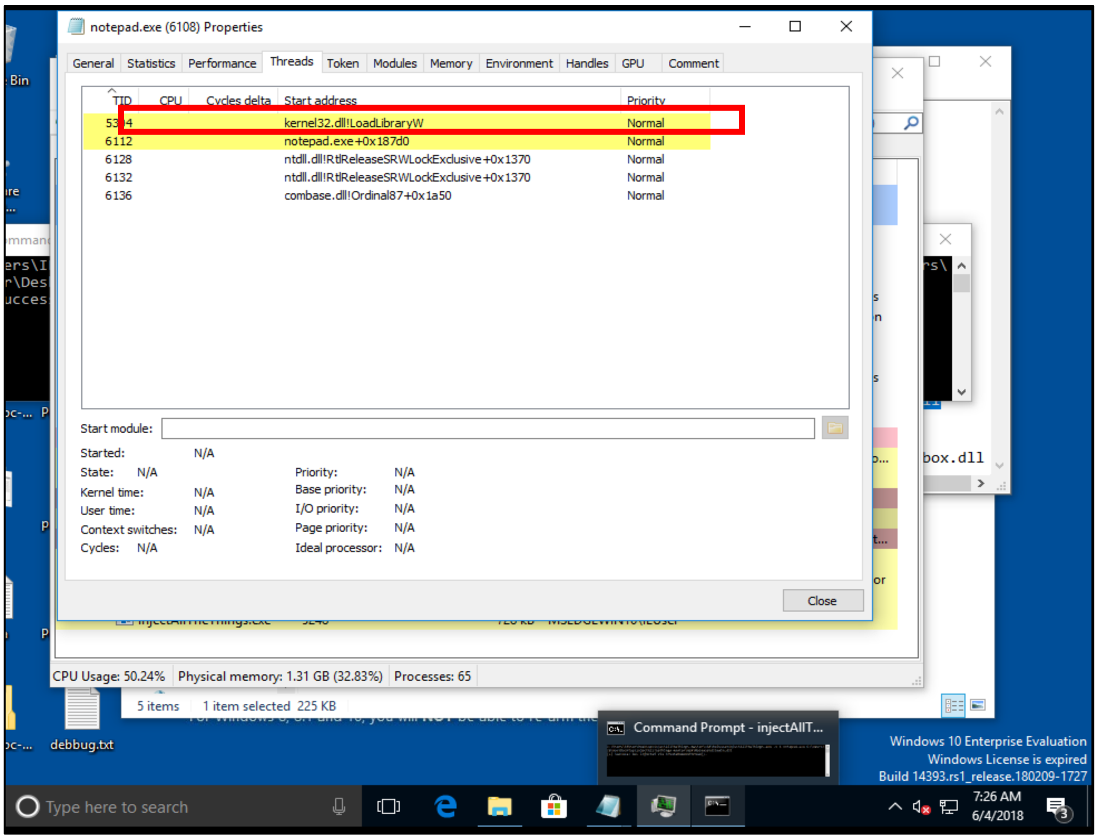
OK | Cancel

injectAllTheThings.exe

CPU Usage: 67.79% | Physical memory

5 items | 1 item selected 225 KB

Next, the corresponding thread that loaded the DLL was found out. It has Thread Id 5304, which is the one that executes LoadLibraryW API.



notepad.exe (6108) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

```
TID    CPU   Cycles delta   Start address                          Priority
5304                        kernel32.dll!LoadLibraryW              Normal
6112                        notepad.exe+0x187d0                    Normal
6128                        ntdll.dll!RtlReleaseSRWLockExclusive+0x1370   Normal
6132                        ntdll.dll!RtlReleaseSRWLockExclusive+0x1370   Normal
6136                        combase.dll!Ordinal87+0x1a50           Normal

Start module:

Started:          N/A
State:     N/A                      Priority:          N/A
Kernel time:      N/A               Base priority:     N/A
User time:        N/A               I/O priority:      N/A
Context switches: N/A               Page priority:     N/A
Cycles:    N/A                      Ideal processor:   N/A
```

Close

CPU Usage: 50.24% | Physical memory: 1.31 GB (32.83%) | Processes: 65

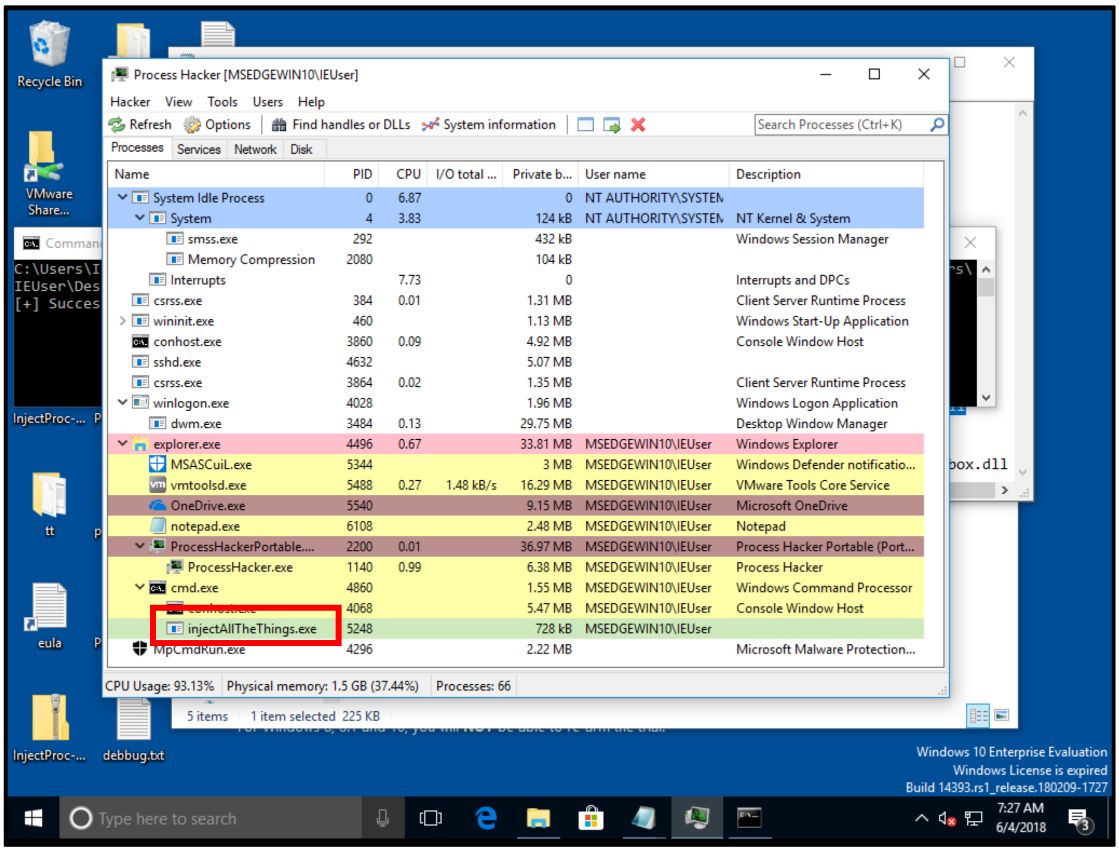
5 items | 1 item selected 225 KB

Command Prompt - injectAllT...

Windows 10 Enterprise Evaluation
Windows License is expired
Build 14393.rs1_release.180209-1727

7:26 AM
6/4/2018

The injection process is injectAllthethings.exe with process id 5248


Process Hacker [MSEDGEWIN10\IEUser]

Name	PID	CPU	I/O total ...	Private b...	User name	Description
System Idle Process	0	6.87		0	NT AUTHORITY\SYSTEM	
System	4	3.83		124 kB	NT AUTHORITY\SYSTEM	NT Kernel & System
smss.exe	292			432 kB		Windows Session Manager
Memory Compression	2080			104 kB		
Interrupts		7.73		0		Interrupts and DPCs
csrss.exe	384	0.01		1.31 MB		Client Server Runtime Process
wininit.exe	460			1.13 MB		Windows Start-Up Application
conhost.exe	3860	0.09		4.92 MB		Console Window Host
sshd.exe	4632			5.07 MB		
csrss.exe	3864	0.02		1.35 MB		Client Server Runtime Process
winlogon.exe	4028			1.96 MB		Windows Logon Application
dwm.exe	3484	0.13		29.75 MB		Desktop Window Manager
explorer.exe	4496	0.67		33.81 MB	MSEDGEWIN10\IEUser	Windows Explorer
MSASCuiL.exe	5344			3 MB	MSEDGEWIN10\IEUser	Windows Defender notificatio...
vmtoolsd.exe	5488	0.27	1.48 kB/s	16.29 MB	MSEDGEWIN10\IEUser	VMware Tools Core Service
OneDrive.exe	5540			9.15 MB	MSEDGEWIN10\IEUser	Microsoft OneDrive
notepad.exe	6108			2.48 MB	MSEDGEWIN10\IEUser	Notepad
ProcessHackerPortable....	2200	0.01		36.97 MB	MSEDGEWIN10\IEUser	Process Hacker Portable (Port...
ProcessHacker.exe	1140	0.99		6.38 MB	MSEDGEWIN10\IEUser	Process Hacker
cmd.exe	4860			1.55 MB	MSEDGEWIN10\IEUser	Windows Command Processor
conhost.exe	4068			5.47 MB	MSEDGEWIN10\IEUser	Console Window Host
injectAllTheThings.exe	5248			728 kB	MSEDGEWIN10\IEUser	
MpCmdRun.exe	4296			2.22 MB		Microsoft Malware Protection...

CPU Usage: 93.13% Physical memory: 1.5 GB (37.44%) Processes: 66

The virtual machine is suspended and the corresponding image is copied and renamed as InjectAllThings_before_ok.vmem.

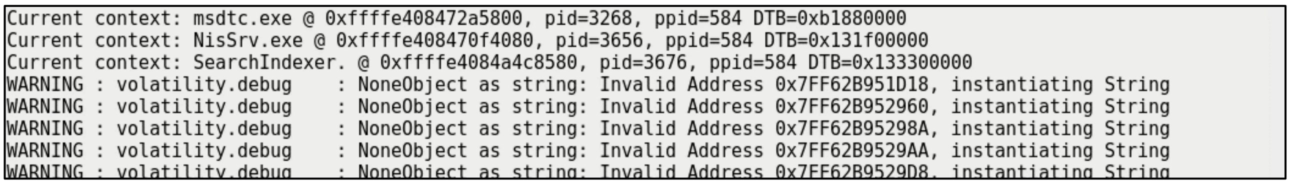
6.2.2 Executing the script

First of all, the volshell volatility is called on the memory image and then the script FindDLLInj.py is called

```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/"InjectAllThings_before_ok.vmem" --profile=
"Win10x64_14393" volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0xffffe408470b3500, pid=4, ppid=0 DTB=0x1aa000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: execfile('FindDllInj.py')
Current context: System @ 0xffffe408470b3500, pid=4, ppid=0 DTB=0x1aa000
Current context: smss.exe @ 0xffffe408489d3800, pid=292, ppid=4 DTB=0x136f00000
Current context: csrss.exe @ 0xffffe40848fee080, pid=384, ppid=376 DTB=0x11760d000
Current context: wininit.exe @ 0xffffe40849275080, pid=460, ppid=376 DTB=0x115d12000
Current context: services.exe @ 0xffffe408494b9600, pid=584, ppid=460 DTB=0x114801000
Current context: lsass.exe @ 0xffffe4084957a440, pid=592, ppid=460 DTB=0x114c20000
Current context: svchost.exe @ 0xffffe40849541800, pid=672, ppid=584 DTB=0x112fb5000
Current context: svchost.exe @ 0xffffe40849547800, pid=736, ppid=584 DTB=0x112421000
Current context: svchost.exe @ 0xffffe40849534800, pid=964, ppid=584 DTB=0x131ae0000
Current context: svchost.exe @ 0xffffe40849538800, pid=980, ppid=584 DTB=0x131ff4000
Current context: svchost.exe @ 0xffffe40849725780, pid=988, ppid=584 DTB=0x132291000
```

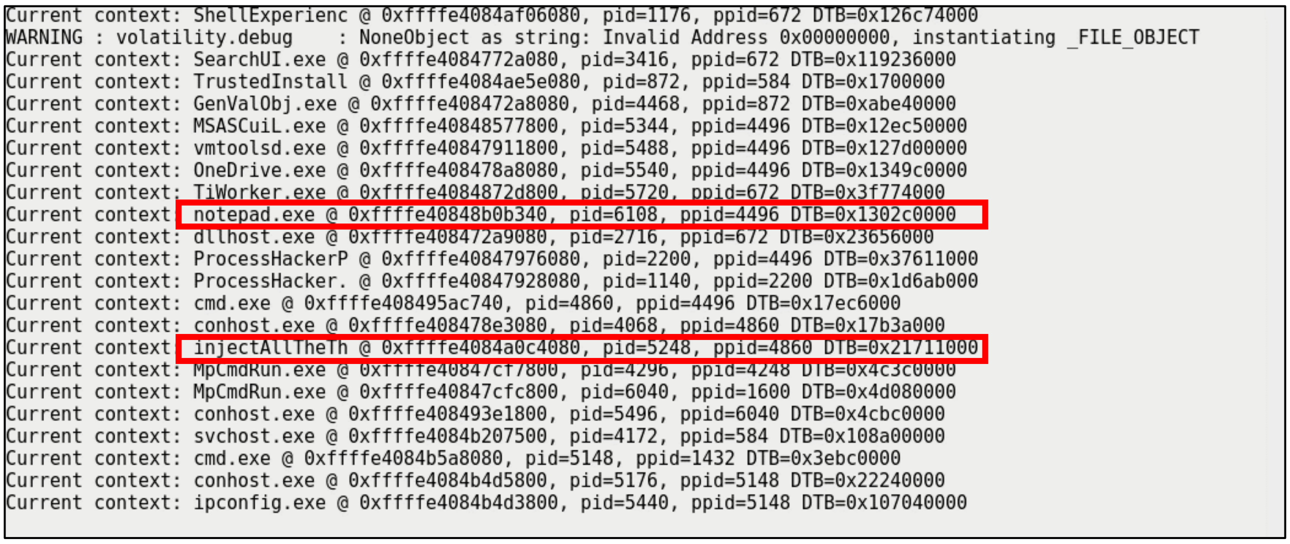
[snip]

as the script, navigates though the processes, the corresponding messages are shown, as well as some warnings



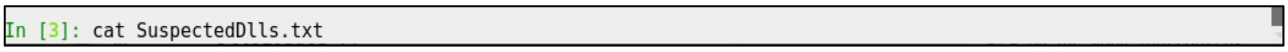
```
Current context: msdtc.exe @ 0xffffe408472a5800, pid=3268, ppid=584 DTB=0xb1880000
Current context: NisSrv.exe @ 0xffffe408470f4080, pid=3656, ppid=584 DTB=0x131f00000
Current context: SearchIndexer. @ 0xffffe4084a4c8580, pid=3676, ppid=584 DTB=0x133300000
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B951D18, instantiating String
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B952960, instantiating String
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B95298A, instantiating String
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B9529AA, instantiating String
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x7FF62B9529D8, instantiating String
```

[snip]



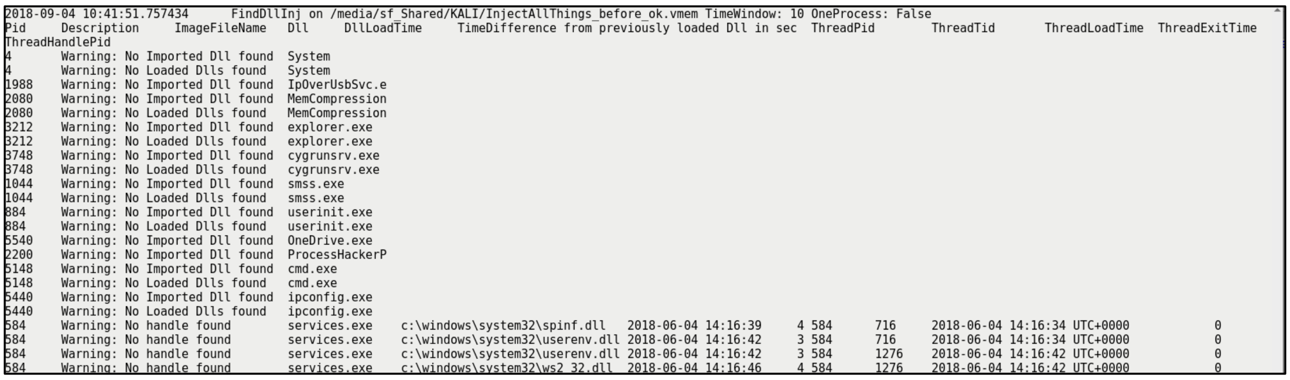
```
Current context: ShellExperienc @ 0xffffe4084af06080, pid=1176, ppid=672 DTB=0x126c74000
WARNING : volatility.debug    : NoneObject as string: Invalid Address 0x00000000, instantiating _FILE_OBJECT
Current context: SearchUI.exe @ 0xffffe4084772a080, pid=3416, ppid=672 DTB=0x119236000
Current context: TrustedInstall @ 0xffffe4084ae5e080, pid=872, ppid=584 DTB=0x1700000
Current context: GenValObj.exe @ 0xffffe408472a8080, pid=4468, ppid=872 DTB=0xabe40000
Current context: MSASCuiL.exe @ 0xffffe40848577800, pid=5344, ppid=4496 DTB=0x12ec50000
Current context: vmtoolsd.exe @ 0xffffe40847911800, pid=5488, ppid=4496 DTB=0x127d00000
Current context: OneDrive.exe @ 0xffffe408478a8080, pid=5540, ppid=4496 DTB=0x1349c0000
Current context: TiWorker.exe @ 0xffffe4084872d800, pid=5720, ppid=672 DTB=0x3f774000
Current context  notepad.exe @ 0xffffe40848b0b340, pid=6108, ppid=4496 DTB=0x1302c0000
Current context: dllhost.exe @ 0xffffe408472a9080, pid=2716, ppid=672 DTB=0x23656000
Current context: ProcessHackerP @ 0xffffe40847976080, pid=2200, ppid=4496 DTB=0x37611000
Current context: ProcessHacker. @ 0xffffe40847928080, pid=1140, ppid=2200 DTB=0x1d6ab000
Current context: cmd.exe @ 0xffffe408495ac740, pid=4860, ppid=4496 DTB=0x17ec6000
Current context: conhost.exe @ 0xffffe408478e3080, pid=4068, ppid=4860 DTB=0x17b3a000
Current context  injectAllTheTh @ 0xffffe4084a0c4080, pid=5248, ppid=4860 DTB=0x21711000
Current context: MpCmdRun.exe @ 0xffffe40847cf7800, pid=4296, ppid=4248 DTB=0x4c3c0000
Current context: MpCmdRun.exe @ 0xffffe40847cfc800, pid=6040, ppid=1600 DTB=0x4d080000
Current context: conhost.exe @ 0xffffe408493e1800, pid=5496, ppid=6040 DTB=0x4cbc0000
Current context: svchost.exe @ 0xffffe4084b207500, pid=4172, ppid=584 DTB=0x108a00000
Current context: cmd.exe @ 0xffffe4084b5a8080, pid=5148, ppid=1432 DTB=0x3ebc0000
Current context: conhost.exe @ 0xffffe4084b4d5800, pid=5176, ppid=5148 DTB=0x22240000
Current context: ipconfig.exe @ 0xffffe4084b4d3800, pid=5440, ppid=5148 DTB=0x107040000
```

When the script is done, the output is shown with the command



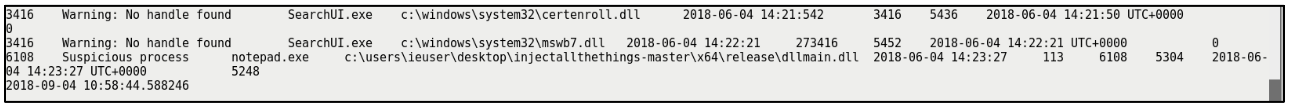
```
In [3]: cat SuspectedDlls.txt
```

these are the first lines of the output file:



```
2018-09-04 10:41:51.757434      FindDllInj on /media/sf_Shared/KALI/InjectAllThings_before_ok.vmem TimeWindow: 10 OneProcess: False
Pid     Description     ImageFileName   Dll     DllLoadTime     TimeDifference from previously loaded Dll in sec  ThreadPid        ThreadTid       ThreadLoadTime  ThreadExitTime
ThreadHandlePid
4       Warning: No Imported Dll found  System
4       Warning: No Loaded Dlls found   System
1988    Warning: No Imported Dll found  IpOverUsbSvc.e
2080    Warning: No Imported Dll found  MemCompression
2080    Warning: No Loaded Dlls found   MemCompression
3212    Warning: No Imported Dll found  explorer.exe
3212    Warning: No Loaded Dlls found   explorer.exe
3748    Warning: No Imported Dll found  cygrunsrv.exe
3748    Warning: No Loaded Dlls found   cygrunsrv.exe
1044    Warning: No Imported Dll found  smss.exe
1044    Warning: No Loaded Dlls found   smss.exe
884     Warning: No Imported Dll found  userinit.exe
884     Warning: No Loaded Dlls found   userinit.exe
5540    Warning: No Imported Dll found  OneDrive.exe
2200    Warning: No Imported Dll found  ProcessHackerP
5148    Warning: No Imported Dll found  cmd.exe
5148    Warning: No Loaded Dlls found   cmd.exe
5440    Warning: No Imported Dll found  ipconfig.exe
5440    Warning: No Loaded Dlls found   ipconfig.exe
584     Warning: No handle found        services.exe    c:\windows\system32\spinf.dll   2018-06-04 14:16:39     4 584       716     2018-06-04 14:16:34 UTC+0000            0
584     Warning: No handle found        services.exe    c:\windows\system32\userenv.dll 2018-06-04 14:16:42     3 584       716     2018-06-04 14:16:34 UTC+0000            0
584     Warning: No handle found        services.exe    c:\windows\system32\userenv.dll 2018-06-04 14:16:42     3 584       1276    2018-06-04 14:16:42 UTC+0000            0
584     Warning: No handle found        services.exe    c:\windows\system32\ws2_32.dll  2018-06-04 14:16:46     4 584       1276    2018-06-04 14:16:42 UTC+0000            0
```

and the last lines are:



```
3416    Warning: No handle found        SearchUI.exe    c:\windows\system32\certenroll.dll      2018-06-04 14:21:542       3416    5436    2018-06-04 14:21:50 UTC+0000
0
3416    Warning: No handle found        SearchUI.exe    c:\windows\system32\mswb7.dll   2018-06-04 14:22:21     273416      5452    2018-06-04 14:22:21 UTC+0000            0
6108    Suspicious process      notepad.exe     c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll  2018-06-04 14:23:27     113     6108    5304    2018-06-
04 14:23:27 UTC+0000            5248
2018-09-04 10:58:44.588246
```

All the above gathered information **is confirmed** as an entry in output file:

```
6108  Suspicious process notepad.exe c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll  2018-06-04 14:23:27     113    6108  5304  2018-06-04 14:23:27     5248
```

Meaning,

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

So, the result was confirmed. For a list of all the test results, see Testing results

6.3 FindDLLInj Script

```
# call from command line, for example:
# python vol.py -f /media/sf_Shared/KALI/"MSEdge - Win10_preview-eaaa27c2-dll_inj_after.vmem" --profile="Win10x64_14393" volshell
# inside volshell enviromnent -->  execfile('FindDllInj.py')

#---- Set Parameters (TimeWindow, OneProcess, WhiteList) ----
TimeWindow=10
OneProcess=False
WhiteList=[]
#TODO fill whitelist with dll names

# Create output file
Out_SuspectedDll = open('SuspectedDlls.txt','w')
import datetime
# introduction line
Out_SuspectedDll.write(str(datetime.datetime.now()) + '\t' + "FindDllInj on "+ sys.argv[2]+ " TimeWindow: " +str(TimeWindow ) + " OneProcess: " + str(OneProcess)+ '\n')
# writing headers
Out_SuspectedDll.write("Pid" + '\t' + "Description" + '\t' + "ImageFileName" + '\t' + "Dll" + '\t' +"DllLoadTime" + '\t' + "TimeDifference from previously loaded Dll in sec" + '\t' + "ThreadPid" + '\t' + "ThreadTid" + '\t' + "ThreadLoadTime" + '\t' + "ThreadExitTime" + '\t'+ "ThreadHandlePid" +'\n')

from datetime import datetime
import time
import calendar

# initialize lists
AllProcessHandle_List=[]
AllProcessSuspectedDlls=[]
CsrssPids=[]
```

```
#---- Get all Processes from memory Image ----
for proc_id in getprocs():
       #---- Read Processes space ----
       p_id= proc_id.UniqueProcessId
       cc(pid=p_id)
       process=proc()
       process_space = process.get_process_address_space()

       if str(process.ImageFileName).lower()=="csrss.exe":
              CsrssPids.append(p_id)

       # ---- Examine Processes VADs: find VADs with specific characteristics and get the corresponding DLL Names ----
       DLLsMappedOnce=[]
       for vad in process.VadRoot.traverse():
            data = process_space.read(vad.Start, 1024)
            if data:
               found = data.find("MZ")
               if found != -1:
                     if hasattr(vad,"ControlArea"):

                            if OneProcess == True:
                                   if int(vad.ControlArea.NumberOfMappedViews) == 1 and \
                                      int(vad.ControlArea.NumberOfUserReferences)==1 :
                                          DLLsMappedOnce.append(str(vad.FileObject.FileName))
                            else:
                                   DLLsMappedOnce.append(str(vad.FileObject.FileName))


       #---- Scan Process PE for IAT entries ----

       #"Scan" PE to reproduce the IAT table - 1st method
```

```
DLLsFromImport=[]
dos_header = obj.Object("_IMAGE_DOS_HEADER",offset = process.Peb.ImageBaseAddress,vm = process_space)
nt_header = dos_header.get_nt_header()
data_dir = nt_header.OptionalHeader.DataDirectory[1]

i = 0

# The following is taken from plugins/overlays/windows/pe_vtypes.py, imports() function of class _LDR_DATA_TABLE_ENTRY
# TODO --> desc_size = self.obj_vm.profile.get_obj_size('_IMAGE_IMPORT_DESCRIPTOR')
desc_size=20
while 1:
          desc = obj.Object('_IMAGE_IMPORT_DESCRIPTOR',
                  vm = process_space,
                  offset = process.Peb.ImageBaseAddress + data_dir.VirtualAddress + (i * desc_size),
                  parent = self)

          # Stop if the IID is paged or all zeros
          if desc == None or desc.is_list_end():
            break

          # Stop if the IID contains invalid fields
          if not desc.valid(nt_header):
            break

          DllName=obj.Object("String",offset = desc.Name+process.Peb.ImageBaseAddress,vm = process_space, length = 128)
          DLLsFromImport.append(str(DllName).lower())

          i += 1
```

```
        if len(DLLsFromImport) == 0:
                Out_SuspectedDll.write(str(p_id) + '\t' + "Warning: No Imported Dll found" + '\t' + process.ImageFileName +'\n')



        #---- Get Process Loaded Dlls ----
        DllsLoaded=[]
        mods = list(process.get_load_modules())
        if len(mods)>0:
                # mods[0] represents the exe module
                previous_load=mods[0].LoadTime
                #---- Get Loaded Dlls Information ----
                for mod in mods:
                        # DllsLoaded layout
                        # mod.LoadTime-previous_load : time difference between current and previous loaded module (in seconds)
                        DllsLoaded.append([str(mod.FullDllName).lower(),process.ImageFileName,          p_id,          mod.LoadCount,
mod.ObsoleteLoadCount,    mod.ReferenceCount,    hex(mod.DllBase),    hex(mod.ParentDllBase),    mod.ImageDll,       mod.LoadTime,
mod.LoadReason,mod.BaseDllName, mod.LoadTime-previous_load])
                        previous_load=mod.LoadTime


        if len(DllsLoaded) == 0:


                Out_SuspectedDll.write(str(p_id) + '\t' + "Warning: No Loaded Dlls found" + '\t' + process.ImageFileName +'\n')


        SuspectedDlls=[]


        #---- Characterize DLL : Find suspicious loaded DLLs in process ----
        for dll in DllsLoaded:


                #  DllName-->  dll name
                t1=dll[0].rfind(".dll")
                t2=dll[0].rfind("\\")
                DllName =dll[0][t2+1:t1+4]
                found1=False
                for item in DLLsFromImport:
                        if item.lower().find(DllName)!=-1:
                            found1=True
                            break
                found2=False
                for item in DLLsMappedOnce:
                        if item.lower().find(DllName)!=-1:
                            found2=True
                            break

                #---- The DLL does not exist in process IAT AND the corresponding VAD fulfill OneProcess criteria ----
                if found1==False and found2==True:
                        # ---- DLL loaded at least 1 sec after from previously loaded DLL AND ----
                        # ---- LoadReason==LoadReasonDynamicLoad OR ObsoleteLoadCount is 6 (DLL explicitly loaded using LoadLibrary
function) ----
                        if dll[12] > 1 and  (dll[10]==4 or dll[4]==6):
                            #---- Is the DLL in the White List? ----
                            if dll[0] not in WhiteList:
                                #  append  ImageFileName,  p_id,  mod.FullDllName,  mod.LoadTime,  mod.LoadTime-previous_load  ,
mod.DllBase, mod.LoadTime in UTC
                                SuspectedDlls.append([str(process.ImageFileName),int(dll[2]),dll[0],
int(dll[9]),int(dll[12]),dll[6], datetime.utcfromtimestamp(dll[9])])
                            else:
                                Out_SuspectedDll.write(str(p_id) + '\t' + "Warning: Dll in WhiteList" + '\t' + dll[0] +'\n')


        if len(SuspectedDlls) >0:

                #---- Get Threads in the context of the process (in Thread List) ----
                Thread_List=[]
                for thread in process.ThreadListHead.list_of_type("_ETHREAD", "ThreadListEntry"):
                        timestamp_utc = calendar.timegm(time.strptime(str(thread.CreateTime), "%Y-%m-%d %H:%M:%S UTC+0000"))
```

```
                    Thread_List.append([int(thread.Cid.UniqueProcess),    int(thread.Cid.UniqueThread),    str(thread.CreateTime),
timestamp_utc, str(thread.ExitTime or ' '), hex(thread.StartAddress)])

        #---- Get Process Handles (in Handle_List) and update ALL Processes' handles (AllProcessHandle_List) ----
        Handle_List=[]
        process.ObjectTable.HandleTableList

        pid=int(p_id)
        for handle in process.ObjectTable.handles():
                if not handle.is_valid():
                        continue

                object_type = handle.get_object_type()

                if object_type == "Thread":
                        thrd_obj = handle.dereference_as("_ETHREAD")
                        # Details handle PID, TID, process id that owns the handle
                        Handle_List.append([int(thrd_obj.Cid.UniqueProcess),int(thrd_obj.Cid.UniqueThread),p_id])
                        AllProcessHandle_List.append([int(thrd_obj.Cid.UniqueProcess),int(thrd_obj.Cid.UniqueThread),p_id])

        handle_pid=0 # to be filled later
        len_SuspectedDlls=len(SuspectedDlls)
        for i in range(0,len_SuspectedDlls):

                for thread_item in Thread_List:
                        #---- Is there a thread created in DLLs TimeWindow? AND still executing? ----
                        # ( if thread creation time between SupsectedDllLoadTime and SupsectedDllLoadTime + TimeWindow
                        #   and thread not terminated )
                        if (SuspectedDlls[i][3] >= thread_item[3] and SuspectedDlls[i][3] <= thread_item[3] + TimeWindow) \
                                and thread_item[4] ==" ":

                                FoundInHandleList=False
                        for hanle_item in Handle_List:
                              # TID of thread same as TID of specific process handle (meaning the thread is created by the
specific process)  --> not suspicious
                              if thread_item[1] == hanle_item[1]  :
                                    FoundInHandleList =True

                        #---- If this thread  is not handled/created by the specific process --> suspicious ----
                        if FoundInHandleList == False:
                              AllProcessSuspectedDlls.append([SuspectedDlls[i], thread_item, handle_pid])

# Minimize False Positives
len_AllProcessSuspectedDlls=len(AllProcessSuspectedDlls)
for i in range(0,len_AllProcessSuspectedDlls):

      FoundInHandleList=False
      FoundInCrss=False

      for handle_item in AllProcessHandle_List:
            # TID found in another processes' handle --> suscicious
            if AllProcessSuspectedDlls[i][1][1] == handle_item[1] :
                  FoundInHandleList=True

                  #---- The thread is not handled by csrss.exe --> suspicious
                  # Handle on the Thread Id not created by csrss.exe  --> suscicious
                  if handle_item[2] not in CsrssPids:
                        AllProcessSuspectedDlls[i][2]=int(handle_item[2]) #--> update with the malware Pid

                        Out_SuspectedDll.write(str(AllProcessSuspectedDlls[i][0][1]) + '\t' + "Suspicious process" + '\t' +
str(AllProcessSuspectedDlls[i][0][0]) + '\t' + str(AllProcessSuspectedDlls[i][0][2]) + '\t' +
str(AllProcessSuspectedDlls[i][0][6]) + '\t' + str(AllProcessSuspectedDlls[i][0][4]) + '\t' +
str(AllProcessSuspectedDlls[i][1][0]) + '\t' + str(AllProcessSuspectedDlls[i][1][1]) + '\t' +
str(AllProcessSuspectedDlls[i][1][2]) + '\t' + str(AllProcessSuspectedDlls[i][1][4]) + '\t'+ str(AllProcessSuspectedDlls[i][2])
+'\n')
                  else:
                        FoundInCrss=True

            if FoundInHandleList==False:
                Out_SuspectedDll.write(str(AllProcessSuspectedDlls[i][0][1])  +  '\t'  +  "Warning:  No  handle  found"  +  '\t'  +
str(AllProcessSuspectedDlls[i][0][0]) + '\t' + str(AllProcessSuspectedDlls[i][0][2]) +
'\t'    +    str(AllProcessSuspectedDlls[i][0][6])    +    '\t'    +    str(AllProcessSuspectedDlls[i][0][4])    +    '\t'    +
str(AllProcessSuspectedDlls[i][1][0])      +      '\t'      +      str(AllProcessSuspectedDlls[i][1][1])      +      '\t'      +
str(AllProcessSuspectedDlls[i][1][2]) + '\t' +  str(AllProcessSuspectedDlls[i][1][4]) + '\t' +str(AllProcessSuspectedDlls[i][2])
+'\n')

# footer line
import datetime
Out_SuspectedDll.write(str(datetime.datetime.now()) + '\n')
Out_SuspectedDll.close()
```