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

Contents.................................................................................................................................................. i
List of Figures and Tables........................................................................................................................ iii
List of Abbreviations................................................................................................................................ iv
Abstract....................................................................................................................................................... v
1 Introduction................................................................................................................................................. 1
  1.1 Malware Analysis ............................................................................................................................... 1
  1.2 The Volatility Framework ................................................................................................................. 2
  1.3 Virtualization...................................................................................................................................... 2
  1.4 What is process injection ................................................................................................................... 3
  1.5 Prerequisites ..................................................................................................................................... 3
  1.6 Problem definition and thesis’ scope ............................................................................................... 4
  1.7 Organization of this document ........................................................................................................ 4
2 Hollow process or process replacement ............................................................................................... 5
  2.1 Introduction and hollow process description ................................................................................... 5
  2.2 Steps to hollow a process ................................................................................................................ 5
  2.3 Process Hollowing Variations .......................................................................................................... 9
    Address of the hollow and address of the allocated memory ......................................................... 9
    Memory protection for the region of pages at memory allocation step ....................................... 9
    No memory umapping .................................................................................................................... 10
    Modifying the legitimate code ......................................................................................................... 10
  2.4 Testing Environment ...................................................................................................................... 10
  2.5 Methodology of Detection ............................................................................................................. 12
    2.5.1 Process Listing ........................................................................................................................ 12
    2.5.2 Comparing kernel and process memory structures ............................................................... 14
    2.5.3 Checking memory permissions .............................................................................................. 18
    2.5.4 Vadinfo – Looking Deeper ..................................................................................................... 19
    2.5.5 Hollowfind plugin .................................................................................................................. 21
    2.5.6 Extracting executables ............................................................................................................ 23
    2.5.7 Comparing executables sizes ................................................................................................. 23
    2.5.8 Determining the owner of the hollowed process .................................................................. 24
    2.5.9 Priority level .......................................................................................................................... 25
    2.5.10 Malfofind plugin ................................................................................................................... 25
List of Figures and Tables

Figure 1 - Process Memory Space during hollowing .......................................................... 8
Figure 2 - Process Listing in test image ............................................................................. 13
Figure 3 – DLL listing in test image .................................................................................. 15
Figure 4 – VAD information in test image .......................................................................... 16
Figure 5 – Comparing module linked list and VAD information in Stuxnet. vmem .......... 17
Figure 6 – Comparing module linked list and VAD information in test images ............. 17
Figure 7 – Checking memory permissions in test image .................................................. 19
Figure 8 – Comparing VAD of legitimate and suspicious process ................................... 20
Figure 9 – Hollowfind plugin in test image ........................................................................ 21
Figure 10 – Hollowfind plugin false positives .................................................................... 22
Figure 11 – Extracting malicious executable in test image ............................................... 23
Figure 12 – Extracting malicious executable in test image ............................................... 23
Figure 13 – Comparing malicious and legit executable in test image ............................... 24
Figure 14 – Comparing legitimates’ and injected process’ priority .................................. 25
Figure 15 – Malfofind output for test image ...................................................................... 26
Figure 16 – ThreadMap output for Stuxnet memory image .............................................. 28
Figure 17 – ThreadMap output for test image .................................................................... 28
Figure 18 – Remote DLL injection steps ............................................................................ 31
Figure 19 – Key points of DLL Injection Detection ......................................................... 32
Figure 20 - DLL Injection Detection Flowchart ............................................................... 34
Figure 21 - Characterize DLL subprocess flowchart ....................................................... 35
Figure 22 - _LDR_DATA_TABLE_ENTRY and _LDR_LOAD_REASON structures .......... 38
Figure 23 - Diagram showing PE File structure and IAT .................................................. 40
Figure 24 – Diagram showing Processes’ and Threads’ data structures ......................... 44
Figure 25 - Processes’ and Threads’ data structures ......................................................... 46

Table 1 – Remote DLL Injection Detection Testing Results ............................................. 54
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>IAT</td>
<td>Import Address Table</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PE</td>
<td>Portable Executable</td>
</tr>
<tr>
<td>PEB</td>
<td>Process Environment Block</td>
</tr>
<tr>
<td>RVA</td>
<td>Relative virtual Address</td>
</tr>
<tr>
<td>SID</td>
<td>Security Identifier</td>
</tr>
<tr>
<td>VAD</td>
<td>Virtual Address Descriptor</td>
</tr>
</tbody>
</table>
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 snapshot 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.
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 legitiimates’ 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 pslit plugin is used.

```
python vol.py - f mem.bin pslit
```

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

- **dlllist** (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>
```
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.

Figure 3 – DLL listing in test image

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].

![Image of VAD information in test image](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:
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):

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

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

17
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):

```python
python vol.py -f mem.bin/vad_mal2/vad_mal2 -v -s sus -m mem盛典.exe -p profile -m -i -t -x -b -o -r -p -q -n -l -x -t -x
```

![Examinining VAD](snip)

**Examinining VAD 0x7f6b1150000L**

Found signature with no ControlArea in VAD 0x7f6b1150000L PrivateMemory: 1, Protection: 6 Vads 1

![Examinining VAD](snip)

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

The code of VadEnumerate.py is:

```python
[snip]
```

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:

```bash
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):

```
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

**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.
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 (VadsS). 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’ 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-easa27c2-4mrepl.vmem --profile="Win10x64_14393"
hollowfind
```

![Hollowfind plugin in test image](image)

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:
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-eaa272c2-onprepl.vmem --profile=Win10x64_14393
VAD node @ 0xfffff80b701271300 Start 0xffffffffc0000000 End 0xfffffffff6a12400000 Flags: Protection: PAGE_EXECUTE_WRITECOPY
```

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.

```bash
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].

![Figure 11 – Extracting malicious executable in test image](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:

![Figure 12 – Extracting malicious executable in test image](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:

![Figure 13 – Comparing malicious and legit executable in test image](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
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-1-5-11 (Authenticated Users)
# This is a process started from Explorer by the user
eaelas.exe (1984): S-1-5-21-1614895754-436374069-839522115-513 (Domain Users)
eaelas.exe (1984): S-1-5-1-0 (Everyone)
eaelas.exe (1984): S-1-5-32-545 (Users)
eaelas.exe (1984): S-1-5-4 (Interactive)
eaelas.exe (1984): S-1-5-5-0-59917 (Logon Session)
eaelas.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 see 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:/user/share/volatility# python vol.py -f /media/sf_Shared/KALI/"stuxnet.vmem" volshell
Volatility Foundation Volatility Framework 2.5
Current context: System @ 0x823c8330, 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 -> IPython's own help system.
obj ect? -> 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=0xa9000a0
680 9
Current context: lsass.exe @ 0x81c698c8, pid=868, ppid=688 DTB=0xa90360
868 8
Current context: lsass.exe @ 0x81c47c00, pid=1928, ppid=668 DTB=0xa903c0
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 malfoind
```

Malfoind 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"
-p=\"\"malfoind\"\nVolatility Foundation Volatility Framework 2.6
Process: notepad.exe Pid: 3912 Ppid: 2604
Address: 0x7ff0b1150000 Protection: PAGE_EXECUTE_READWRITE
Initially mapped file object: C:\Windows\System32\notepad.exe
Currently mapped file object: None
0x7ff0b1150000 4d 5a 90 60 03 00 00 00 04 00 00 00 ff 00 00 MZ..............
0x7ff0b1150010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..........@
0x7ff0b1150020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..........@
0x7ff0b1150030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..........@
```

![Figure 15 – Malfoind output for test image](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 malfoind but was detected by hollowfind. The second variation was detected only by malfoind 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 KDB6 search...
Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with WinXPSP2x86)
AS Layer1 : IA32PageMemoryPage (Kernel AS)
AS Layer2 : FileAddressSpace (/media/sf_Shared/KALI/stuxnet.vmem)
PAE type : PAE
DBT : 0x3190000
KDBG : 0x80545ae00
Number of Processors : 1
Image Type (Service Pack) : 3
KPCR for CPU 0 : 0xfdf6f0000
KUSER_SHARED DATA : 0xfdf6f0000
Image date and time : 2011-06-03 04:31:36 UTC+0000
Image local date and time : 2011-06-03 09:31:36 -0400
```

```
root@kali:/usr/share/volatility# python vol.py -f /media/sf_Shared/KALI/~stuxnet.vmem" --profile="WinXPSP2x86"
```

```Thread Map Information:
Process: ssms.exe PID: 376 PPID: 4
** No thread is pointing to process's image file
** Found suspicious threads in process
Thread ID: 388 (ACTIVE)
Reason:
Thread points to a vad without a file object
Vad Info:
  Thread Entry Point: 0x0
  Vad Base Address: 0x0
  Vad End Address: 0xffffffff
  Vad Size: 0xffffffff
  Vad Tag: VadS
  Vad Protection: PAGE READWRITE
  Vad Mapped File: '"
** Couldn't read memory
```

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

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

```
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](image)

**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

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.

* 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
             - Characterize the DLL as suspicious
           - NO
             - Append DLL, process and thread information in the suspicious DLL list
       - NO
         - Is the DLL loaded at least 1 sec after from previously loaded DLL?  
           - YES
             - Display warning
           - NO
             - Is the DLL in the White List?
               - YES
                 - Characterize the DLL as not suspicious
               - NO

- NO
  - Is there a thread created in DLLs’ TimeWindow? ** AND still executing?
    - YES
      - Is this thread handled/created by the specific process?
        - NO
          - Is this thread handled/created by csrss.exe?
            - NO
              - Characterize the DLL as suspicious
            - YES
              - Append DLL, process and thread information in the suspicious DLL list
        - YES
          - Is there another thread created in DLLs’ TimeWindow?

** 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 Volatility’s 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:
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pid</td>
<td>Process ID</td>
</tr>
<tr>
<td>Description</td>
<td>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”</td>
</tr>
<tr>
<td>ImageFileName</td>
<td>Process Name</td>
</tr>
<tr>
<td>Dll</td>
<td>DLL full name</td>
</tr>
<tr>
<td>DllLoadTime</td>
<td>Date and Time that the DLL was loaded (UTC+0000)</td>
</tr>
<tr>
<td>TimeDifference from previously loaded Dll in sec</td>
<td>Time Difference from the previously loaded DLL inside the context of the specific process (in sec)</td>
</tr>
<tr>
<td>ThreadPid</td>
<td>ID of the process under which the injection thread is created (injected process, same as Pid)</td>
</tr>
<tr>
<td>ThreadTid</td>
<td>ID of the injection thread created by the malicious process, inside the context of the Pid</td>
</tr>
<tr>
<td>ThreadLoadTime</td>
<td>Date and Time that the Thread begun executing (UTC+0000)</td>
</tr>
<tr>
<td>ThreadExitTime</td>
<td>Thread Exit Date and Time or ‘ ‘</td>
</tr>
<tr>
<td>ThreadHandlePid</td>
<td>ID of the process that handles the injection Thread (injector process ID)</td>
</tr>
</tbody>
</table>

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](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.

Figure 22 - _LDR_DATA_TABLE_ENTRY and _LDR_LOAD_REASON structures

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:

```python
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].
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()
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.

Using the readpe command, the PE structure as described above, is presented.
```plaintext
root@all:/usr/share/volatility# readpe dll_inj\executable.6120.exe

DOS Header
Magic number: 0x0040 (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: 0
Initial (relative) SS value: 0
Initial SP value: 0x08
Initial IP value: 0
Initial (relative) CS value: 0
Address of relocation table: 0x40
Overlay number: 0
DEM identifier: 0
DEM information: 0
PE header offset: 0xf0

COFF/File header
Machine: 0x0000 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: 0x70
Characteristics: 0x22
-executable image
-executable image
-can handle more than 2 GB addresses

Optional/Image header
Magic number: 0x020b (PE32+)
Linker major version: 14
linker minor version: 0
Size of .text section: 0x1b600
Size of .data section: 0x24e00
Size of .bss section: 0
Entry point: 0x1b7040
Address of .text section: 0x1000
Image Base: 0x7f7de4f00000
Alignemnt 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: 0x100
Size of stack to reserve: 0x80000
Size of stack to commit: 0x1000
Size of heap space to reserve: 0x100000
Size of heap space to commit: 0x1000

Data directories
Import Table: 0x17300 (600 bytes)
Resource Table: 0x26800 (10576 bytes)
Exception Table: 0x25800 (224 bytes)
Base Relocation Table: 0x40000 (524 bytes)
Debug: 0x1e100 (56 bytes)
Load Config Table: 0x1e550 (208 bytes)

Import Address Table (IAM): 0x1e5e0 (2424 bytes)

08: 0x400
10: 0x400
12: 0x400
14: 0x400
16: 0x400
18: 0x400
20: 0x400
22: 0x400
24: 0x400
26: 0x400
28: 0x400
30: 0x400
32: 0x400
34: 0x400
36: 0x400
38: 0x400
40: 0x400
42: 0x400
44: 0x400
46: 0x400
48: 0x400
50: 0x400
52: 0x400
54: 0x400
56: 0x400
58: 0x400
60: 0x400
62: 0x400
64: 0x400
66: 0x400
68: 0x400
70: 0x400
72: 0x400
74: 0x400
76: 0x400
78: 0x400
80: 0x400
82: 0x400
84: 0x400
86: 0x400
88: 0x400
90: 0x400
92: 0x400
94: 0x400
96: 0x400
98: 0x400
A0: 0x400
A2: 0x400
A4: 0x400
A6: 0x400
A8: 0x400
AA: 0x400
AC: 0x400
AE: 0x400
B0: 0x400
B2: 0x400
B4: 0x400
B6: 0x400
B8: 0x400
BA: 0x400
BC: 0x400
BE: 0x400
C0: 0x400
C2: 0x400
C4: 0x400
C6: 0x400
C8: 0x400
CA: 0x400
CC: 0x400
CE: 0x400
D0: 0x400
D2: 0x400
D4: 0x400
D6: 0x400
D8: 0x400
DA: 0x400
DC: 0x400
DE: 0x400
E0: 0x400
E2: 0x400
E4: 0x400
E6: 0x400
E8: 0x400
EA: 0x400
EC: 0x400
EE: 0x400
F0: 0x400
F2: 0x400
F4: 0x400
F6: 0x400
F8: 0x400
FA: 0x400
FC: 0x400
FE: 0x400
```

[snip]
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.

```bash
root@kali:~usr/share/volatility# objdump -d -x -h dll_inj\executable.6128.exe >dll_inj)objdump_notepad.txt
```

which confirms the readpe output. Part of this text file is shown below.
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-rtltlsupport-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-l1-1-0.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',
 'urtmon.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.

```python
#---- 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"))
```

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

```plaintext
In [22]: dt["_EPROCESS"]
'_EPROCESS' (1668 bytes)
0x0 : Pcb ['_KPROCESS']
0x20 : ProcessLock ['EX_PUSH_LOCK']
0x2e0 : RundownProtect ['EX_RUNDOWN_REF']
0x2e8 : UniqueProcessId ['unsigned int']
0x2f0 : ActiveProcessLinks ['LIST_ENTRY']
0x300 : AccountingField ['BitField', ['end bit': 2, 'start bit': 1, 'native_type': 'unsigned long']]
0x308 : AffinityPermanent ['BitField', ['end bit': 39, 'start bit': 18, 'native_type': 'unsigned long']]

[snip]
```

```plaintext
[snip]
```

```plaintext
[snip]
```
The 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):

```python
# Get Process' Handles (in Handle_List) and update ALL Processes' handles
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")

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

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):
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:

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

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

```
```

### 3.5 FindDLLInj Testing

#### 3.5.1 Testing Environment

In websites [https://gist.github.com/zmwangx/e728c56f428bc703c6f6](https://gist.github.com/zmwangx/e728c56f428bc703c6f6) and [https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/](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](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/](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](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:

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Test Memory Image Characteristics</th>
<th>Output line of interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>DLL injection in process notepad.exe (pid=6108)</strong>&lt;br&gt;During DLL execution (respective window popped up)&lt;br&gt;Injecting process still active (injectAllTheThings.exe, pid 5248)&lt;br&gt;Image File name: InjectAllThings_before_ok.vmem&lt;br&gt;Output file name: SuspectedDlls - InjectAllThings_before_ok.txt</td>
<td>6108 Suspicious process notepad.exe&lt;br&gt;c:\users\ieuser\desktop\injectallthings-master\x64\release\dllmain.dll 2018-06-04 14:23:27 UTC+0000&lt;br&gt;6108 5304 2018-06-04 14:23:27 UTC+0000 5248&lt;br&gt;This output line is analyzed as:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pid</th>
<th>Description</th>
<th>ImageFileName</th>
<th>Dll</th>
<th>DllLoadTime</th>
<th>TimeDifference</th>
<th>ThreadPid</th>
<th>ThreadTid</th>
<th>ThreadLoadTime</th>
<th>ThreadExitTime</th>
<th>ThreadHandlePid</th>
</tr>
</thead>
<tbody>
<tr>
<td>6108</td>
<td>Suspicious process</td>
<td>notepad.exe</td>
<td>c:\users\ieuser\desktop\injectallthings-master\x64\release\dllmain.dll</td>
<td>2018-06-04 14:23:27</td>
<td>113</td>
<td>6108</td>
<td>5304</td>
<td>2018-06-04 14:23:27</td>
<td>5248</td>
<td></td>
</tr>
</tbody>
</table>
| 2 | Injection in process notepad.exe (pid 6108) achieved  
   During DLL execution (respective window popped up)  
   **Injecting Process terminated**  
   Image File name: InjectAllThings_closeExec.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 UTC+0000 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 UTC+0000 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 15:02:08 UTC+0000 125  
   2560 2540 2018-06-04 15:02:09 UTC+0000 6060  
   2560 Suspicious process notepad.exe  
   c:\users\ieuser\desktop\injectallthethings-master\x64\release\dllmain.dll 2018-06-04 15:02:09 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.

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Test Memory Image Characteristics</th>
<th>Output line of interest</th>
</tr>
</thead>
</table>
| 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 |
<table>
<thead>
<tr>
<th>Step</th>
<th>Event Description</th>
<th>Output (other than warnings)</th>
</tr>
</thead>
</table>
| 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 | **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 | **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-eaa27c2-3_DifferentProcs.vmem Output file name: SuspectedDllsWin10_preview-eaa27c2-3_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 14:13:48 56 4328 2144 2018-08-31 14:13:48 UTC+0000 5056 |
| Injection in three different Processes | 408 Suspicious process explorer.exe  
| One of them terminated (pid 6052) | c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:14:42 26  
| injecting Processes still active | 408 3908 2018-08-31 14:14:42 UTC+0000 5372  
| Image File name: MSEdge - Win10_preview-eaaa27c2-3_DifferentProcs-1_Term.vmem | 4328 Suspicious process mspaint.exe  
| Output file name: | c:\users\ieuser\desktop\injectproc-master\injectproc-master\x64\debug\mbox.dll 2018-08-31 14:13:48 56  
| SuspectedDlls- Win10_preview-eaaa27c2-3_DifferentProcs-1_Term.txt | 4328 2144 2018-08-31 14:13:48 UTC+0000 5056  
| Note: The OneProcess parameter is set to False |

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 to 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


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


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


[37] 0x00sec, 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


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


[54] CodeMachine, Catalog of key Windows kernel data structures, https://www.codemachine.com/article_kernelstruct.html, 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 volatility, http://akovid.blogspot.com/2014/02/volatility-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


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).

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

<table>
<thead>
<tr>
<th>Name</th>
<th>PID</th>
<th>CPU</th>
<th>User name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Idle Process</td>
<td>0</td>
<td>65.68</td>
<td>0</td>
<td>NT AUTHORITY\SYSTEM</td>
</tr>
<tr>
<td>System</td>
<td>4</td>
<td>5.21</td>
<td>128 KB</td>
<td>NT AUTHORITY\SYSTEM</td>
</tr>
<tr>
<td>smss.exe</td>
<td>295</td>
<td></td>
<td></td>
<td>Windows Session Manager</td>
</tr>
<tr>
<td>Memory Compression</td>
<td>1368</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>5.59</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>csrss.exe</td>
<td>392</td>
<td>0.02</td>
<td>1.26 MB</td>
<td></td>
</tr>
<tr>
<td>wininit.exe</td>
<td>464</td>
<td></td>
<td>940 KB</td>
<td>Windows Start-Up Application</td>
</tr>
<tr>
<td>csrss.exe</td>
<td>472</td>
<td>0.03</td>
<td>1.32 MB</td>
<td>Client Server Runtime Process</td>
</tr>
<tr>
<td>winlogon.exe</td>
<td>540</td>
<td></td>
<td>1.68 MB</td>
<td>Windows Logon Application</td>
</tr>
<tr>
<td>dm.exe</td>
<td>832</td>
<td>0.12</td>
<td>43.08 MB</td>
<td>Desktop Window Manager</td>
</tr>
<tr>
<td>explorer.exe</td>
<td>3544</td>
<td>0.67</td>
<td>36.91 MB</td>
<td>Windows Explorer</td>
</tr>
<tr>
<td>MSASCui.exe</td>
<td>5384</td>
<td></td>
<td>2.04 MB</td>
<td>Windows Defender notification</td>
</tr>
<tr>
<td>vmmelds.exe</td>
<td>5404</td>
<td>0.33</td>
<td>0.38 B/s</td>
<td>VMware Tools Core Service</td>
</tr>
<tr>
<td>Process Hacker Portable</td>
<td>7455</td>
<td>0.07</td>
<td>36.72 MB</td>
<td>Process Hacker Portable [Port...</td>
</tr>
<tr>
<td>Process Hacker</td>
<td>7504</td>
<td>1.51</td>
<td>15.1 MB</td>
<td>Process Hacker</td>
</tr>
<tr>
<td>notepad.exe</td>
<td>6400</td>
<td></td>
<td>2.33 MB</td>
<td>Notepad</td>
</tr>
<tr>
<td>cmd.exe</td>
<td>6224</td>
<td></td>
<td>1.56 MB</td>
<td>Windows Command Processor</td>
</tr>
<tr>
<td>config.exe</td>
<td>6660</td>
<td></td>
<td>5.67 MB</td>
<td>Microsoft\Windows\System 32\Network Host</td>
</tr>
<tr>
<td>cohost.exe</td>
<td>3896</td>
<td>0.19</td>
<td>4.85 MB</td>
<td>Console Window Host</td>
</tr>
<tr>
<td>sshd.exe</td>
<td>3948</td>
<td></td>
<td>5 MB</td>
<td></td>
</tr>
<tr>
<td>MsdCtrlRun.exe</td>
<td>4196</td>
<td></td>
<td>3.02 MB</td>
<td>Microsoft Malware Protection</td>
</tr>
<tr>
<td>OneDrive.exe</td>
<td>7728</td>
<td></td>
<td>10 MB</td>
<td>MSEDGEWIN10\U\User</td>
</tr>
<tr>
<td>notepad.exe</td>
<td>7594</td>
<td></td>
<td>1.20 MB</td>
<td>MSEDGEWIN10\U\User</td>
</tr>
</tbody>
</table>

The general properties for the legitimate process are:
and for the malicious one are:

Both processes use 0x7ff6f3070000 as base address. The different memory characteristics are demonstrated below and are consistent with the ones described in Methodology of Detection section.
The other characteristics such as priority and environment variables, are similar:
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

<table>
<thead>
<tr>
<th>PID</th>
<th>PID</th>
<th>Thds</th>
<th>Mems</th>
<th>Sess</th>
<th>Wave64</th>
<th>Start</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>408</td>
<td>408</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Dynamic Link Library Listing

<table>
<thead>
<tr>
<th>Base</th>
<th>Size</th>
<th>LoadCount</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>Windows\System32\notepad.exe</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>Windows\System32\cmd.exe</td>
</tr>
</tbody>
</table>

Profile:

Volatility Foundation Volatility Framework 2.6

python vol.py -f /media/sf_Shared/KALY/MSEdge - Win10_preview-eaaa27c2-onreplNew.vmem --profile=WinDbg14393

notepad.exe pid: 7284
Command Line: "C:\Windows\System32\notepad.exe"
Virtual Address Descriptor information

Module linked lists and VAD

Checking memory permissions
Hollowfind plugin

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

Extracting executables, Comparing executables sizes

Malfind

---

70
Threadmap plugin

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

6.1.3 Injection Using Process Hollow project

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

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.
Next, the corresponding thread that loaded the DLL was found out. It has Thread Id 5304, which is the one that executes LoadLibraryW API.
The injection process is injectAllthethings.exe with process id 5248.

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

Current context: mstsc.exe @ 0xffffffff72a5900, pid=3268, ppid=584 DTB=0xb1800000
Current context: NisSrv.exe @ 0xffffffff72a49000, pid=356, ppid=584 DTB=0xb3160000
WARNING : volatility.debug : Unexpected string: Invalid Address 0x7f62051d18, instantiating String
WARNING : volatility.debug : Unexpected string: Invalid Address 0x7f62052960, instantiating String
WARNING : volatility.debug : Unexpected string: Invalid Address 0x7f62052980, instantiating String
WARNING : volatility.debug : Unexpected string: Invalid Address 0x7f620560a0, instantiating String
WARNING : volatility.debug : Unexpected string: Invalid Address 0x7f6205f000, instantiating String

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:

and the last lines are:
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,

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

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 environment --> 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=[]

nt_header = dos_header.get_nt_header()

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 ----
DllLoaded=[]
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:
    # DllLoaded layout
    # mod.LoadTime-previous_load : time difference between current and previous loaded module (in seconds)
    previous_load=mod.LoadTime

if len(DllLoaded) == 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 DllLoaded:

    # 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) ----
        #---- 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) + ' ' + Warning: Dll in WhiteList' + '

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"))

#---- 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 )
           and thread_item[4] == " ":

            FoundInHandleList=False
for handle_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 --> suspicious
        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 --> suspicious
        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:
if FoundInHandleList==False:
    Out_SuspectedDll.write(str(AllProcessSuspectedDlls[i][0][1]) + ' ' + 'Warning: No handle found' + ' ' + str(AllProcessSuspectedDlls[i][0][0]) + ' ' + str(AllProcessSuspectedDlls[i][0][2]) + ' ' + str(AllProcessSuspectedDlls[i][0][6]) + ' ' + str(AllProcessSuspectedDlls[i][0][4]) + ' ' + str(AllProcessSuspectedDlls[i][1][0]) + ' ' + str(AllProcessSuspectedDlls[i][1][1]) + ' ' + str(AllProcessSuspectedDlls[i][2]) + '')

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