

June, 2020

# Windows Memory Forensics II

Introduction to Structured Analysis with Volatility.  
Investigative Process Steps. Windows Processes.

## Contents

Structured Analysis and Investigative Process .....	3
Get the OS Map .....	3
IMAGEINFO .....	4
KDBGSCAN .....	4
KDBG Encryption .....	4
Process investigation .....	5
PSLIST .....	6
PSSCAN .....	8
PSTREE .....	10
PSXVIEW .....	12
CMDSCAN .....	15
CONSOLES .....	15
Conclusion .....	16

---

# STRUCTURED ANALYSIS AND INVESTIGATIVE PROCESS

---

After a short introduction into unstructured memory analysis in Part I of the *Windows Memory Forensics* series, now it is time to get more... structured! Let us begin with parsing memory objects. We will discuss two major memory analysis frameworks later in this series: Volatility and Rekall. Rekall is Volatility's fork, based on the development branch from 2011. After two years, results still were not accepted into the Volatility trunk, so the branch turned into a new project – on December 13, 2013, Rekall was born.

Volatility is a framework written in Python. Memory analysis with Volatility consists of running various Python plugins against a memory dump. Each plugin has a specific use-case and can have specific command line options. Where to start an investigation? There are several options on how to approach memory forensics. In these articles, we will roughly follow guidelines published by SANS institute. SANS divides RAM analysis process into the following phases:

1. Identify rogue processes
2. Analyze process DLLs and handles
3. Review network artifacts
4. Look for evidence of code injection
5. Check for signs of a rootkit
6. Dump suspicious processes and drivers

---

## GET THE OS MAP

---

Before we can run any Volatility plugin (except help and info 😊), the specific version of operating system needs to be determined, because there are differences in memory layout for every Windows version, Service Pack, build. For example, memory structures found at a specific offset in Windows 7 are placed to a different location in that of Windows 10 1903 build, OS operations on Windows XP differ from those of Server2012R2. Not to mention, 32- vs. 32-PAE vs. 64-bit OS releases: virtual-to-physical address mapping is tailored for each of them, so Volatility (or any other analysis tool) must take that into account as well. Volatility needs to be aware of what OS version it is looking at, so that plugins can do their work. OS identification is most often done by searching for KDBG structure. KDBG stands for Kernel Debugging Data Block and is (as the name implies) used for debugging purposes. It contains lots of useful information about the investigated system, and its signature is distinct for every Windows operating system, thus finding KDBG reveals what OS we have on the table. Additional info contained in KDBG are debugging symbols – kernel variables for that specific system. These variables will be used by other modules, so determining KDBG's location is the logical first step of any investigation.

## IMAGEINFO

Imageinfo will most likely be the first plugin to be run on any memory image, even if we know what OS we are investigating.

Imageinfo searches the memory image and looks for the KDBG signature. This signature may not be specific to a single version of OS, and Volatility reflects it in the output: you may spot several OS versions in the Suggested Profiles line of the output. Once the KDBG is located, linked list of processes, PsActiveProcessHead, is walked. Each process running on the system holds information about the OS Service Pack in its Process Environment Block, with the most often encountered Service Pack being included in imageinfo's output. Combining those two pieces together, we now have OS version and service pack ready for subsequent use.

Besides OS and SP, imageinfo provides us with the date and time when the memory image had been captured.

As many other Volatility plugins rely on information from KDBG, providing KDBG's offset can speed up their runtime. Invoke the plugin with `-g [offset]` or `--kdbg=[offset]` parameter.

## KDBGSCAN

Imageinfo plugin does have its caveats though: once it finds the first KDBG signature, it stops. The problem with this is that there may be multiple KDBG signatures present in the memory image, and the first hit can be a false positive, thus relying on this information can mislead investigators in further actions. The Kdbgscan plugin scans the entire memory image, addressing the issue mentioned above, and for each KDBG hit it determines the number of processes and loaded modules (based on PsActiveProcessHead and PsLoadedModulesList variables). If the KDBG hit is a false positive, the number of processes and modules will be zero, compared to reasonable number in case of true KDBG.

### KDBG ENCRYPTION

Since Windows 8, the KDBG header is encrypted on running systems for any 64-bit OS release. It is possible to decrypt this header by looking into the function involved in this process – KdCopyDataBlock. Values KiWaitNever and KiWaitAlways can be used to reveal KDBG's content.

In the case of Win8+ systems, kdbgscan provides different output. It includes offset to KdCopyDataBlock so that KDBG can be decrypted in real time. As in the case of providing offset to KDBG, passing KdCopyDataBlock offset with `-g [offset]` or `--kdbg=[offset]` argument to other Volatility plugins can speed up the processing.

```

vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64
kdbgscan | tee whitepaper/pony/kdbgscan.txt
Volatility Foundation Volatility Framework 2.6
*****
Instantiating KDBG using: Unnamed AS Win8SP1x64 (6.3.9600 64bit)
Offset (V)           : 0xf802190bf530
Offset (P)           : 0x2196bf530
KdCopyDataBlock (V) : 0xf80218ff9908
Block encoded        : Yes
Wait never           : 0x1250c07e0023b219
Wait always          : 0x1d904e4f60438001
KDBG owner tag check : True
Profile suggestion (KDBGHeader): Win8SP1x64
Version64            : 0xf802190bfe60 (Major: 15, Minor: 9600)
Service Pack (CmNtCSDVersion) : 0
Build string (NtBuildLab) : 9600.17415.amd64fre.winblue_r4.1
PsActiveProcessHead  : 0xffffffff802190d8000 (48 processes)
PsLoadedModuleList   : 0xffffffff802190f2250 (137 modules)
KernelBase           : 0xffffffff80218e19000 (Matches MZ: True)
Major (OptionalHeader) : 6
Minor (OptionalHeader) : 3
KPCR                 : 0xffffffff8021911c000 (CPU 0)

```

---

## PROCESS INVESTIGATION

---

Once we have determined the memory image's profile, we can start calling other Volatility plugins. The first step in most investigations is process listing. Process is a container for threads, and threads are workhorses of operating systems. To learn what was running on the system at the time of the memory acquisition, we need a list of processes. Volatility provides different ways to get this list. We will discuss process-related plugins one by one.

Process structure contains various types of data. On a 64bit Windows system, each process has as much as 256TB of virtual memory. The lowest 8TB is reserved for the process memory – so-called “user space”, which leaves the upper 248 TB left for “kernel space”. The user space of process stores the following information:

1. Environment variables
2. Process parameters
3. Heap memory
4. Executable code
5. Process environment block – PEB – includes info about ServicePack version, pointer to list of loaded modules for the process etc.
6. Stack
7. DLLs

## PSLIST

This plugin lists running processes by walking a double-linked list of processes that are maintained by the OS. The pointer to this data structure - PsActiveProcessHead kernel variable - is stored in KDBG, that is why finding the KDBG offset is so important to the investigation.

Below is the output from the pslist plugin run against our Windows 8.1 memory image. Output includes:

- virtual offset to each process,
- process name,
- PID (Process IDentifier) and
- PPID (Parent Process IDentifier),
- number of threads,
- respective session in which the process is running,
- indicator if it is 32bit process on 64bit system,
- start time,
- exit time (if terminated process' data have not been reallocated yet, possibly because there is still some opened handle to it).

This is the same memory sample that we have investigated in Part I of *Windows Memory Forensics* series. You may remember that system was infected with Khaledi malware. We will search for more traces left on the system by this malware.

Offset(V)	Name	PID	PPID	Thds	Hnds	Sess	Wow64	Start	Exit
0xffffe0001a624040	System	4	0	106	0	----	0	2020-05-29 14:40:46	UTC+0000
0xffffe0001baaf040	smss.exe	280	4	2	0	----	0	2020-05-29 14:40:46	UTC+0000
0xffffe0001bc1b900	csrss.exe	356	348	8	0	0	0	2020-05-29 14:40:47	UTC+0000
0xffffe0001a694500	wininit.exe	412	348	1	0	0	0	2020-05-29 14:40:47	UTC+0000
0xffffe0001a698280	csrss.exe	420	404	10	0	1	0	2020-05-29 14:40:47	UTC+0000
0xffffe0001a717080	winlogon.exe	460	404	2	0	1	0	2020-05-29 14:40:47	UTC+0000
0xffffe0001a6f2080	services.exe	488	412	3	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001a713500	lsass.exe	496	412	10	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001bd2b900	svchost.exe	564	488	11	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001b8bf500	svchost.exe	596	488	10	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001bd1f900	dwm.exe	688	460	7	0	1	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d21d900	VBoxService.exe	744	488	11	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d2df080	svchost.exe	832	488	20	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d21f900	svchost.exe	856	488	39	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d330240	svchost.exe	904	488	19	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d321480	svchost.exe	952	488	13	0	0	0	2020-05-29 14:40:48	UTC+0000
0xffffe0001d34d080	svchost.exe	348	488	17	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d211900	spoolsv.exe	676	488	9	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d20d900	svchost.exe	252	488	22	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d445080	svchost.exe	1148	488	6	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d4dd500	svchost.exe	1280	488	15	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d4ce900	MsMpEng.exe	1328	488	20	0	0	0	2020-05-29 14:40:49	UTC+0000
0xffffe0001d57a400	NisSrv.exe	1696	488	5	0	0	0	2020-05-29 14:40:50	UTC+0000
0xffffe0001d6126c0	explorer.exe	1864	1856	66	0	1	0	2020-05-29 14:41:00	UTC+0000
0xffffe0001d609080	taskhostex.exe	1888	856	8	0	1	0	2020-05-29 14:41:00	UTC+0000
0xffffe0001d5df900	svchost.exe	1244	488	1	0	0	0	2020-05-29 14:41:00	UTC+0000
0xffffe0001d6f5900	dllhost.exe	972	564	8	0	1	0	2020-05-29 14:41:00	UTC+0000
0xffffe0001d721900	SearchIndexer.	2076	488	16	0	0	0	2020-05-29 14:41:01	UTC+0000
0xffffe0001bd1c080	audiodg.exe	2460	832	4	0	0	0	2020-05-29 14:41:04	UTC+0000
0xffffe0001b123080	svchost.exe	2500	488	0	----	0	0	2020-05-29 14:41:06	UTC+0000
2020-05-29 14:46:13	UTC+0000								
0xffffe0001d759900	VBoxTray.exe	2568	1864	11	0	1	0	2020-05-29 14:41:10	UTC+0000
0xffffe0001b8f5080	Autoruns.exe	2644	1864	3	0	1	1	2020-05-29 14:41:14	UTC+0000
0xffffe0001d94e900	mmc.exe	2916	2904	11	0	1	0	2020-05-29 14:41:22	UTC+0000
0xffffe0001b095900	dllhost.exe	2104	564	3	0	1	0	2020-05-29 14:43:11	UTC+0000
0xffffe0001b015080	cmd.exe	1988	1864	1	0	1	0	2020-05-29 14:43:37	UTC+0000
0xffffe0001aaa6080	conhost.exe	2112	1988	2	0	1	0	2020-05-29 14:43:37	UTC+0000
0xffffe0001d4e1340	ielowutil.exe	2620	564	7	0	1	1	2020-05-29 14:44:06	UTC+0000
0xffffe0001aa9a080	iexplore.exe	1040	564	18	0	1	0	2020-05-29 14:44:06	UTC+0000
0xffffe0001aa3b540	iexplore.exe	180	1040	20	0	1	1	2020-05-29 14:44:06	UTC+0000
0xffffe0001abc0180	SearchProtocol	208	2076	6	0	1	0	2020-05-29 14:44:11	UTC+0000
0xffffe0001abb9900	SearchFilterHo	320	2076	2	0	0	0	2020-05-29 14:44:11	UTC+0000
0xffffe0001abdb900	iexplore.exe	2364	1040	19	0	1	1	2020-05-29 14:44:11	UTC+0000
0xffffe0001ce97080	FTK Imager.exe	1248	1864	22	0	1	0	2020-05-29 14:44:51	UTC+0000
0xffffe0001cea1080	WMIADAP.exe	1228	856	4	0	0	0	2020-05-29 14:44:52	UTC+0000
0xffffe0001ce89080	WmiPrvSE.exe	2896	564	6	0	0	0	2020-05-29 14:44:52	UTC+0000
<b>0xffffe0001ab2e080</b>	<b>Kip1.exe</b>	<b>1552</b>	<b>1724</b>	<b>4</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>2020-05-29 14:45:08</b>	<b>UTC+0000</b>
0xffffe0001ce94080	iexplore.exe	764	1040	72	0	1	1	2020-05-29 14:45:09	UTC+0000
0xffffe0001a80d900	FlashUtil_Acti	3032	564	5	0	1	0	2020-05-29 14:45:13	UTC+0000

In the excerpt above, the process named Kip1.exe is an eye-catcher. What is it supposed to do on the system? Is it a known application, or is it a piece of malware?

Plist plugin is efficient and fast. Providing the location of KDBG (or KdCopyDataBlock in case of Windows 8+ systems) can speed up the process even more:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64  
pslist -g 0xf80218ff9908
```

Based on the strange name or the unusual number of instances of some known processes it may be possible to immediately spot an anomaly on the system. However, this approach has its faults. Relying solely on the KDBG location may spoil the process if KDBG is corrupted for some reason. Pslist can also be fooled by an attacker: As the plugin follows a double linked list of processes, the attacker is able to hide malicious processes by modifying pointers in kernel memory so that the process does not appear in the list. This modification does not affect process performance, so it is quite a reliable way to hide in plain sight.

Fortunately, there are other plugins available for which to solve this seemingly problematic issue.

## PSSCAN

Psscan serves a similar purpose as the pslist plugin – get the list of running processes. However, the way psscan fulfills the task is completely different. Instead of a walking OS structure, psscan scans (as the name implies) an entire memory image, looks for a process object pool tag and parses data – EPROCESS structure - found at that location. As it does not rely on an active process list maintained by OS kernel, psscan can reveal terminated processes with their data structures not yet overwritten!

Comparing pslist plugin output with psscan's when run on the same image:



Offset(P) Time exited	Name	PID	PPID	PDB	Time created
0x000000010834e900 UTC+0000	mmc.exe	2916	2904	0x000000010efe3000	2020-05-29 14:41:22
0x000000019e209080 UTC+0000	taskhostex.exe	1888	856	0x0000000100214000	2020-05-29 14:41:00
0x000000019e2126c0 UTC+0000	explorer.exe	1864	1856	0x0000000100043000	2020-05-29 14:41:00
0x000000019e2f5900 UTC+0000	dllhost.exe	972	564	0x0000000102867000	2020-05-29 14:41:00
0x000000019e321900 UTC+0000	SearchIndexer.	2076	488	0x00000001034ad000	2020-05-29 14:41:01
0x000000019e359900 UTC+0000	VBoxTray.exe	2568	1864	0x000000010848e000	2020-05-29 14:41:10
0x000000019f645080 UTC+0000	svchost.exe	1148	488	0x000000019f088000	2020-05-29 14:40:49
0x000000019f6ce900 UTC+0000	MsMpEng.exe	1328	488	0x00000001c109f000	2020-05-29 14:40:49
0x000000019f6dd500 UTC+0000	svchost.exe	1280	488	0x00000001c0f8f000	2020-05-29 14:40:49
0x000000019f6e1340 UTC+0000	ielowutil.exe	2620	564	0x00000001c61d6000	2020-05-29 14:44:06
0x000000019f77a400 UTC+0000	NisSrv.exe	1696	488	0x00000001c7007000	2020-05-29 14:40:50
0x000000019f7df900 UTC+0000	svchost.exe	1244	488	0x0000000102367000	2020-05-29 14:41:00
0x00000001a460d900 UTC+0000	svchost.exe	252	488	0x00000001a0ab0000	2020-05-29 14:40:49
0x00000001a4611900 UTC+0000	spoolsv.exe	676	488	0x00000001a1412000	2020-05-29 14:40:49
0x00000001a461d900 UTC+0000	VBoxService.ex	744	488	0x00000001aa41a000	2020-05-29 14:40:48
0x00000001a4730240 UTC+0000	svchost.exe	904	488	0x00000001a2aa1000	2020-05-29 14:40:48
0x00000001a474d080 UTC+0000	svchost.exe	348	488	0x00000001a4ba5000	2020-05-29 14:40:49
0x00000001a970a080 UTC+0000	Autoruns.exe	2644	1864	0x0000000108af5000	2020-05-29 14:41:14
0x00000001b097d500 UTC+0000	svchost.exe	596	488	0x00000001a8fa0000	2020-05-29 14:40:48
0x00000001be489080 UTC+0000	WmiPrvSE.exe	2896	564	0x0000000119065000	2020-05-29 14:44:52
0x00000001be494080 UTC+0000	iexplore.exe	764	1040	0x000000011ba96000	2020-05-29 14:45:09
0x00000001be497080 UTC+0000	FTK Imager.exe	1248	1864	0x0000000119239000	2020-05-29 14:44:51
0x00000001be4a1080 UTC+0000	WMIADAP.exe	1228	856	0x0000000119c98000	2020-05-29 14:44:52
0x00000001d181b900 UTC+0000	csrss.exe	356	348	0x00000001b96cd000	2020-05-29 14:40:47
0x00000001d191c080 UTC+0000	audiodg.exe	2460	832	0x000000010677e000	2020-05-29 14:41:04
0x00000001d191f900 UTC+0000	dwm.exe	688	460	0x00000001a983a000	2020-05-29 14:40:48
0x00000001d192b900 UTC+0000	svchost.exe	564	488	0x00000001a86d8000	2020-05-29 14:40:48
0x00000001da6af040 UTC+0000	smss.exe	280	4	0x00000001bf052000	2020-05-29 14:40:46
0x000000021589a080 UTC+0000	iexplore.exe	1040	564	0x000000000481d000	2020-05-29 14:44:06
0x00000002158a6080 UTC+0000	conhost.exe	2112	1988	0x000000019fc2a000	2020-05-29 14:43:37
0x000000021592e080 UTC+0000	Kip1.exe	1552	1724	0x000000011923b000	2020-05-29 14:45:08

Already, the first column is different: pslist provided the virtual offset to the process structure, whereas psscan listed the physical offset. This makes perfect sense though, as psscan goes through “raw” bytes of memory image and does not care about virtual-to-physical memory address translation. To perform this, the location of PDB is included in the output. (PDB, or Page Directory Base, also known as DTB – Directory Table Base, contains indexes used in virtual-to-physical address translation. This topic exceeds the scope of this whitepaper.) As a result of scanning the memory, processes are not listed in time sequential order. Again, due to the scanning nature of the plugin, this is no surprise: psscan adds processes into the output as they are found in the image, regardless of timestamps contained in the process structure. A small disadvantage of using psscan is that it takes longer than pslist. Psscan’s ability to find terminated and unlinked processes compensates for this.

Psscan can produce a graph of parent-child process relationships if we configure it to write output into a DOT file. After it is finished, open the dot file in some graphical viewer (for example xdot) to explore which process spawned which child and look for any anomaly.

Invoking psscan with .dot output:

```
$ vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64
psscan --output=dot --output-file=psscan.dot
$ xdot psscan.dot
```

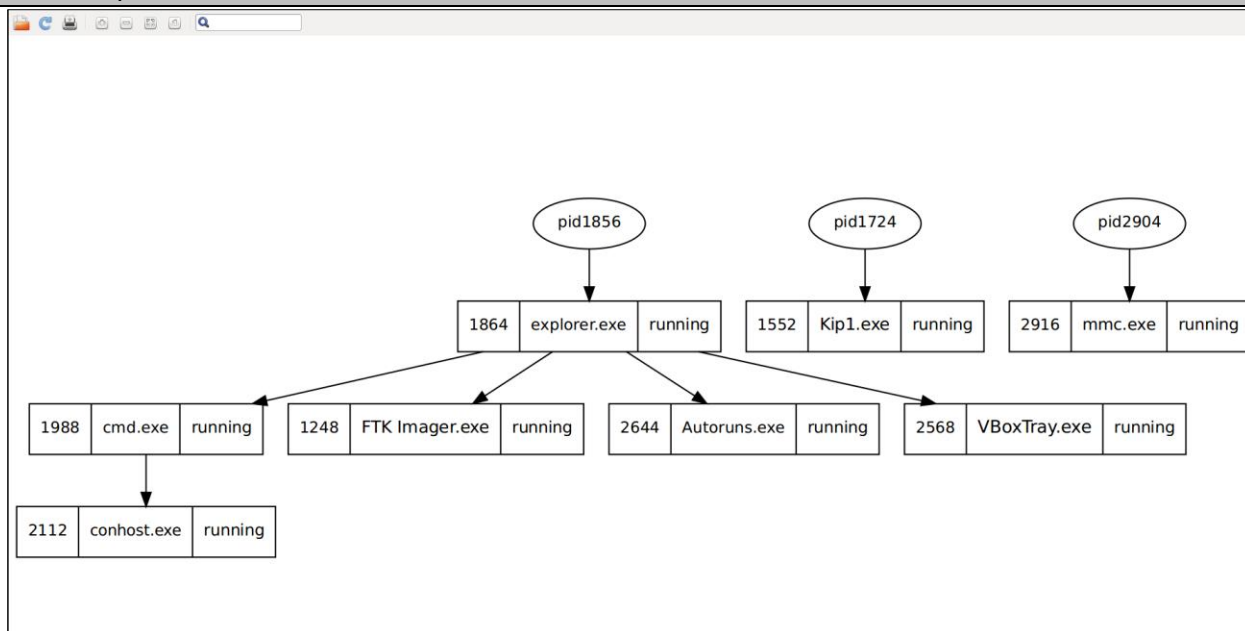


Figure 1: psscan, excerpt from DOT output.

## PSTREE

Not only can the presence or absence of a particular process be an indicator of adverse activity, but anomalies in process relationships can be just as valuable of a pointer. The pstree plugin allows us to investigate parent-child relationships of processes listed in a

double linked list, pointed to by PsActiveProcessHead, which means it shares the same pros and cons as pslist. Tree view is useful to spot outliers, such as system processes being spawned by user processes (for example, Explorer.exe should never be spawning any instance of svchost.exe), or other unexpected parent-child relationships (for example, why should iexplore.exe spawn cmd.exe?).

Name	Pid	PPid	Thds	Hnds	Time
0xfffffe0001d4dd500:svchost.exe 14:40:49 UTC+0000	1280	488	15	0	2020-05-29
0xfffffe0001d330240:svchost.exe 14:40:48 UTC+0000	904	488	19	0	2020-05-29
0xfffffe0001d721900:SearchIndexer. 14:41:01 UTC+0000	2076	488	16	0	2020-05-29
. 0xfffffe0001abb9900:SearchFilterHo 14:44:11 UTC+0000	320	2076	2	0	2020-05-29
. 0xfffffe0001abc0180:SearchProtocol <snip> 14:40:48 UTC+0000	208	2076	6	0	2020-05-29
0xfffffe0001d21f900:svchost.exe 14:40:48 UTC+0000	856	488	39	0	2020-05-29
. 0xfffffe0001cea1080:WMIADAP.exe 14:44:52 UTC+0000	1228	856	4	0	2020-05-29
. 0xfffffe0001d609080:taskhostex.exe 14:41:00 UTC+0000	1888	856	8	0	2020-05-29
0xfffffe0001d34d080:svchost.exe 14:40:49 UTC+0000	348	488	17	0	2020-05-29
. 0xfffffe0001a694500:wininit.exe 14:40:47 UTC+0000	412	348	1	0	2020-05-29
.. 0xfffffe0001a6f2080:services.exe 14:40:48 UTC+0000	488	412	3	0	2020-05-29
... 0xfffffe0001d445080:svchost.exe 14:40:49 UTC+0000	1148	488	6	0	2020-05-29
... 0xfffffe0001d21d900:VBoxService.ex 14:40:48 UTC+0000	744	488	11	0	2020-05-29
... 0xfffffe0001d20d900:svchost.exe 14:40:49 UTC+0000	252	488	22	0	2020-05-29
.. 0xfffffe0001a713500:lsass.exe 14:40:48 UTC+0000	496	412	10	0	2020-05-29
. 0xfffffe0001bc1b900:csrss.exe <snip>	356	348	8	0	2020-05-29
0xfffffe0001a624040:System 14:40:46 UTC+0000	4	0	106	0	2020-05-29
. 0xfffffe0001baaf040:smss.exe 14:40:46 UTC+0000	280	4	2	0	2020-05-29
0xfffffe0001d94e900:mmc.exe 14:41:22 UTC+0000	2916	2904	11	0	2020-05-29
0xfffffe0001a717080:winlogon.exe 14:40:47 UTC+0000	460	404	2	0	2020-05-29
. 0xfffffe0001bd1f900:dwm.exe 14:40:48 UTC+0000	688	460	7	0	2020-05-29
0xfffffe0001a698280:csrss.exe 14:40:47 UTC+0000	420	404	10	0	2020-05-29
0xfffffe0001ab2e080:Kip1.exe 14:45:08 UTC+0000	1552	1724	4	0	2020-05-29

The great thing about pstree is that it allows requesting a more verbose output, by invoking the -v switch. Pstree will include three more lines to the output for each process

with the -v switch: audit and path with path to the process's executable, and cmd with full command line that has been used at process startup. This information can be invaluable to the investigation process.

Looking at the pstree output below (note that this is different memory sample as in example before), we can see that wscript.exe has been used to run the VisualBasic script named Judgement\_04212020\_2313.vbs.

Name	Pid	PPid	Thds	Hnds	Time
0xfffffe00078e2c080:wininit.exe	416	352	1	0	2020-05-18
14:03:45 UTC+0000					
audit: \Device\HarddiskVolume2\Windows\System32\wininit.exe					
cmd: wininit.exe					
path: C:\Windows\system32\wininit.exe					
. 0xfffffe0007ac06080:services.exe	492	416	4	0	2020-05-18
14:03:45 UTC+0000					
audit: \Device\HarddiskVolume2\Windows\System32\services.exe					
cmd: C:\Windows\system32\services.exe					
path: C:\Windows\system32\services.exe					
.. 0xfffffe0007b1dd900:svchost.exe	1324	492	15	0	2020-05-18
14:03:50 UTC+0000					
audit: \Device\HarddiskVolume2\Windows\System32\svchost.exe					
cmd: C:\Windows\system32\svchost.exe -k iissvcs					
path: C:\Windows\system32\svchost.exe					
.. 0xfffffe0007b1ef900:spoolsv.exe	652	492	8	0	2020-05-18
...					
.. 0xfffffe00079d9a900:wscript.exe	1012	2592	4	0	2020-05-18
14:23:55 UTC+0000					
audit: \Device\HarddiskVolume2\Windows\System32\wscript.exe					
cmd: wscript					
C:\Users\Analyst\Desktop\!vzorky\Judgement_04212020_2313_2\Judgement_04212020_2313\Judgement_04212020_2313.vbs					
path: C:\Windows\system32\wscript.exe					

Running a more verbose pstree on our Kpot sample reveals nothing groundbreaking about suspicious Kip1 process:

0xfffffe0001ab2e080:Kip1.exe	1552	172	4	0	2020-05-29	14:45:08	UTC+0000
audit: \Device\HarddiskVolume2\ProgramData\Kip1.exe							
cmd: C:\ProgramData\Kip1.exe							
path: C:\ProgramData\Kip1.exe							

## PSXVIEW

EPROCESS kernel structure is not the only (kernel) object that holds information about processes running on the system. From the nature of Windows OS, this information is maintained at several other locations. Comparing information gathered from all of the locations can reveal anomalies that can be an indication of the system being infected. Psxview is a Volatility plugin designed for this purpose. Besides a walking process list, it collects data from:

- csrss.exe system process, which has a handle to any process, starting after it,

- session manager – each logon session has its own list of processes, belonging to that session,
- PSpCidTable, used by CPU scheduler and pointed to (surprisingly) by KDBG,
- ETHREAD kernel structure – scanning memory for threads and tracking to which process the thread belongs can reveal hidden processes, running on the system,
- Desktop threads – maintained by Windows Stations; we can use this thread list to trace back the process they belong to.

After querying the above-mentioned sources, psxview shows tabular output with True/False statements about which process has been found using which method.

There are some exceptions of the rules, so the mere fact the process is not present in any of the above-mentioned resources does not mean it is malicious. To have Volatility apply known anomalies to psxview output, invoke it with -R switch. Exceptions include, for example, System and smss.exe processes cannot be tracked by csrss as they have already started before it; nor do they have a corresponding logon session, or desktop threads. Lsass.exe is going to be "False" in Desktop threads. When the plugin is run with -R option, known gaps are labeled as "Okay".

See the output below.

```

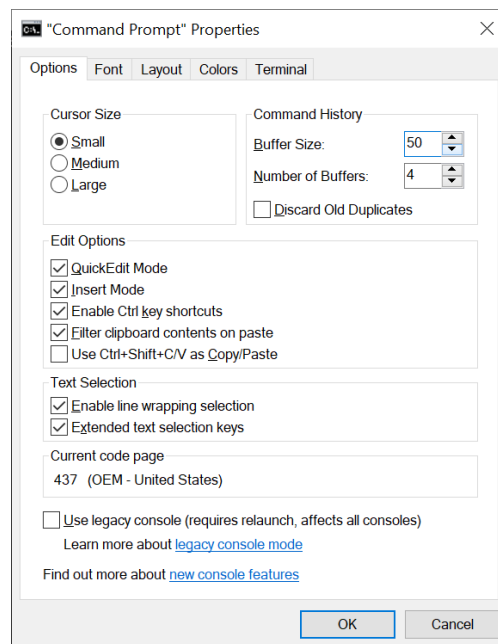
vol.py -f /mnt/hgfs/AnalystVMShare/memdump-pony-2.mem --profile Win8SP1x64 psxview -R
Volatility Foundation Volatility Framework 2.6
Offset(P)          Name                PID pslist psscan thrdproc pspcid csrss session deskthrd
ExitTime
-----
0x0000000219a94500 wininit.exe         412 True   True   True   True   True   True   True
0x0000000219af2080 services.exe        488 True   True   True   True   True   True   True
0x00000001be494080 iexplore.exe       764 True   True   True   True   True   True   True
0x00000001a461f900 svchost.exe         856 True   True   True   True   True   True   True
0x0000000217815080 cmd.exe             1988 True   True   True   True   True   True   True
0x00000001d191c080 audiodg.exe        2460 True   True   True   True   True   True   True
0x00000002158a6080 conhost.exe        2112 True   True   True   True   True   True   True
0x000000019e359900 VBoxTray.exe       2568 True   True   True   True   True   True   True
0x0000000219b17080 winlogon.exe       460 True   True   True   True   True   True   True
...
0x0000000219b13500 lsass.exe           496 True   True   True   True   True   True   True
0x00000001d192b900 svchost.exe         564 True   True   True   True   True   True   True
0x00000001be4a1080 WMIADAP.exe        1228 True   True   True   True   True   True   True
0x000000019e2126c0 explorer.exe       1864 True   True   True   True   True   True   True
0x00000001a460d900 svchost.exe         252 True   True   True   True   True   True   True
0x00000001a4730240 svchost.exe         904 True   True   True   True   True   True   True
0x000000019f6ce900 MsMpEng.exe        1328 True   True   True   True   True   True   True
0x000000021904f900 FlashUtil_Acti    3032 True   True   True   True   True   True   True
0x000000019f645080 svchost.exe        1148 True   True   True   True   True   True   True
0x00000001d191f900 dwm.exe             688 True   True   True   True   True   True   True
0x00000001a4611900 spoolsv.exe        676 True   True   True   True   True   True   True
0x000000019e321900 SearchIndexer.    2076 True   True   True   True   True   True   True
0x00000001be489080 WmiPrvSE.exe      2896 True   True   True   True   True   True   True
0x00000001a970a080 Autoruns.exe      2644 True   True   True   True   True   True   True
0x00000001b097d500 svchost.exe         596 True   True   True   True   True   True   True
0x000000019e2f5900 dllhost.exe        972 True   True   True   True   True   True   True
0x00000001be497080 FTK Imager.exe    1248 True   True   True   True   True   True   True
0x0000000217895900 dllhost.exe        2104 True   True   True   True   True   True   True
0x00000001a474d080 svchost.exe         348 True   True   True   True   True   True   True
0x000000021592e080 Kip1.exe           1552 True   True   True   True   True   True   True
0x00000002159b9900 SearchFilterHo    320 True   True   True   True   True   True   True
0x000000010834e900 mmc.exe            2916 True   True   True   True   True   True   True
0x00000002159db900 iexplore.exe      2364 True   True   True   True   True   True   True
0x000000019e209080 taskhostex.exe   1888 True   True   True   True   True   True   True
0x00000001a461d900 VBoxService.ex    744 True   True   True   True   True   True   True
0x00000001da6af040 smss.exe           280 True   True   True   True   True   Okay   Okay
0x0000000219a98280 csrss.exe          420 True   True   True   True   True   Okay   True
0x0000000219a24040 System             4 True   True   True   True   True   Okay   Okay
0x00000001d181b900 csrss.exe          356 True   True   True   True   True   Okay   True
0x0000000217923080 svchost.exe       2500 True   False Okay   Okay   Okay   True   Okay
2020-05-29 14:46:13 UTC+0000

```

## CMDSCAN

When discussing psxview, we have already mentioned that one of the sources of information on running processes is csrss.exe. This process can be a great source of additional information in multiple ways. It is able to reveal more about actions performed by an attacker if s/he happened to use the console shell – cmd.exe. The process csrss.exe on Windows XP/2003/Vista/2008, or the process conhost.exe on Windows 7 systems store information about command history – meaning they maintain a list of the most recently executed commands. The number of these commands depends on how the terminal's MaxHistory value is set on the machine being analyzed, which by default is 50. This can be adjusted either in console Properties, accessed from the GUI, or by setting registry value (per user) HKCU\Console\HistoryBufferSize.

Plugin cmdscan scans the memory images and searches for COMMAND\_HISTORY structures. Based on its scanning nature, it can find information from both opened and terminated consoles. Below in this plugin's output we can see that wscript.exe (called indeed from command shell) was used to run a malicious Visual Basic script. Except for the command run, plugin details include the name of the consoles host process (if it is csrss.exe or conhost.exe), name of the application (process) using cmd.exe, the location of the command history buffer and the application process handle.



## CONSOLES

With cmdscan we can get information about commands that have been run on the system. However, we can only see one part of the story – what has been written to the console – we cannot determine what the output of those commands was. To help with this, plugin consoles comes in handy. Although its name does not imply it, this is another scanning plugin, this time targeting CONSOLE\_INFORMATION structures in an investigated memory image. This structure stores the entire screen buffer – both input and output, thus enabling us to see the same results as an attacker. Besides this, cmdscan lists the title of the original console window, the name and PID of attached processes, and any aliases associated with the executed command.

In the example below, consoles show the running VisualBasic script from command line using WScript.exe. However, as the script did not provide any output, we will not get any new information on what the script was used for.

```
*****
ConsoleProcess: conhost.exe Pid: 2608
Console: 0x7ff7671c7220 CommandHistorySize: 50
HistoryBufferCount: 1 HistoryBufferMax: 4
OriginalTitle: Pr?kazov? riadok
Title: Pr?kazov? riadok
----
CommandHistory: 0x143ae85bd0 Application: cmd.exes (x8????Flags: Allocated,
Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x143ae566d0
Cmd #0 at 0x143ae5d5b0: wscript
Cmd #1 at 0x143ae5c250: wscript
C:\Users\Analyst\!vzorky\Judgement_04212020_2313_2\Judgement_04212020_2313\Ju
dgement_04212020_2313.vbs
----
Screen 0x143ae5bd60 X:3 Y:0
Dump:
```

---

## CONCLUSION

---

In this section we completed the first phase of memory analysis – getting information about the memory image and process exploration. We went through various Volatility plugins that can be used to populate a list of running processes or process objects carved from memory. As an investigator, we should cross-check our output of plugins that approach the search differently. We should not rely solely on output from pslist and pstree as they can be fooled.

When going through process list, we should focus on:

- unknown process names,
- anomalies in parent-child relationship,
- unusual number of instances of system processes,
- suspicious process command line,
- outliers in process start time,
- processes with unexplained “False” entries in psxview output.

These steps can reveal evidence of various malicious activity on the system. However, more sophisticated attackers may be able to remain hidden. Either way, the memory analysis process continues. Next on the list will be the investigation of process objects: handles, DLLs, files, mutexes and more. Stay tuned for Part III of *Windows Memory Forensics* series!