SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE

Memory Forensics Analysis Poster

Memory analysis is the decisive victory on the battlefield between offense and defense, giving the upper hand to incident responders by exposing injection and hooking techniques that would otherwise remain undetected.

Memory Analysis will prepare your team to:

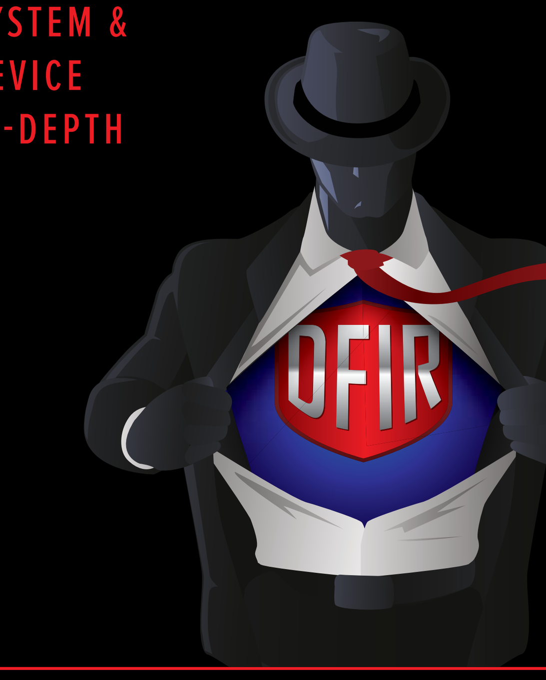
- Discover zero-day malware
- Detect compromises
- Uncover evidence that others miss

The Battleground Between Offense and Defense

DFIR-Memory_v2.1_7-17

digital-forensics.sans.org

SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE



OPERATING SYSTEM & DEVICE IN-DEPTH

FOR500 Windows Forensics (Formerly FOR408) GCFE

FOR518 Mac Forensics

FOR526 Memory Forensics In-Depth

FOR585 Advanced Smartphone Forensics GASF

INCIDENT RESPONSE & THREAT HUNTING

FOR508 Advanced IR and Threat Hunting GCFA

FOR572 Advanced Network Forensics and Analysis GNFA

FOR578 Cyber Threat Intelligence

FOR610 REM: Malware Analysis GREM

SEC504 Hacker Tools, Techniques, Exploits, and Incident Handling GCIH

@sansforensics | sansforensics | dfir.to/DFIRCast | dfir.to/gplus-sansforensics | dfir.to/MAIL-LIST

Rekall Memory Forensic Framework

The Rekall Memory Forensic Framework is a collection of memory acquisition and analysis tools implemented in Python under the GNU General Public License. This cheatsheet provides a quick reference for memory analysis operations in Rekall, covering acquisition, live memory analysis, and parsing plugins used in the Six-Step Investigative Process. For more information on this tool, visit **rekall-forensic.com**.

Getting Started with Rekall

Single Command Example
```
$ rekal -f image.img pslist
```
Starting an Interactive Session
```
$ rekal -f image.img
```
`[1] image.img 11:14:35>` — [1] session # · image.img current image · 11:14:35> local system time

Process Enumeration

PSLIST Enumerate processes
Rekall uses 5 techniques to enumerate processes by default (PsActiveProcessList, sessions, handles, CSRSS, PspCidTable)
```
[1] image.img 11:14:35> pslist
```
Narrow the process enumeration using "method="
```
[1] image.img 11:14:35> pslist method=
"PsActiveProcessHead"
```
Customize pslist output with efilters
```
[1] image.img 11:14:35> select
EPROCESS,ppid,process_create_time from pslist()
order by process_create_time
```
PROCINFO Display detailed process & PE info
```
[1] image.img 11:14:35> procinfo <PID>
```
DESKTOPS Enumerate desktops and desktop threads
```
[1] image.img 11:14:35> desktops verbosity=<#>
```
SESSIONS Enumerate sessions and associated processes
```
[1] image.img 11:14:35> sessions
```
THREADS Enumerates process threads
```
[1] image.img 11:14:35> threads proc_regex= "chrome"
```
DT Displays Specific Kernel Data Structures
```
[1] image.img 11:14:35> dt("_EPROCESS")
```

Extracting Process Details

DLLLIST List of loaded dlls by process.
Filter on specific process(es) by including the process identifier <PID> as a positional argument
```
[1] image.img 11:14:35> dlllist [1580,204]
```
HANDLES List of open handles for each process include pid or array of pids separated by commas
object_types="TYPE" – Limit to handles of a certain type {Process, Thread, Key, Event, File, Mutant, Token, Port}
```
[1] image.img 11:14:35> handles 868, object_types="Key"
```
FILESCAN Scan memory for _FILE_OBJECT handles
```
[1] image.img 11:15:35> filescan output="filescan.txt"
```

Malicious Code Detection

IDENTIFY SUSPICIOUS PROCESSES by COMMAND LINE

PSTREE (WITH VERBOSITY) – List processes with path and command line
```
[1] be.aff4 11:14:35> describe(pstree) - View columns to output
[1] be.aff4 11:14:35> select _EPROCESS,ppid,cmd,path from pstree()
```
DETECT CODE INJECTION by VAD ANALYSIS

MALFIND Find injected code and dump sections
- <pid> Positional Argument: Show information only for specific PIDs
- phys_eprocess= Provide physical offset of process to scan
- eprocess= Provide virtual offset for process to scan
- dump_dir= Directory to save memory sections
```
[1] be.aff4 11:14:35> malfind eprocess=0x853cf460,dump_dir="/cases"
```
LDRMODULES Detect unlinked DLLs
- verbosity= Verbose: show full paths from three DLL lists
```
[1] be.aff4 11:14:35> ldrmodules 1936
```

Windows® Memory Acquisition (winpmem)

CREATING AN AFF4 (Open **cmd.exe** as Administrator)
```
C:\> winpmem_<version>.exe -o output.aff4
```
INCLUDE PAGE FILE
```
C:\> winpmem_<version>.exe -p c:\pagefile.sys -o output.aff4
```
EXTRACTING THE RAW MEMORY IMAGE FROM THE AFF4
```
C:\> winpmem_<version>.exe output.aff4 --export PhysicalMemory -o memory.img
```
EXTRACTING TO RAW USING REKALL
```
$ rekal -f win7.aff4 imagecopy --output-image="/cases/win7.img
```
OTHER WINPMEM OPTIONS
```
view aff4 metadata (-V)| elf output (--elf)
```

Counters to Memory Forensics: Modern Anti-Analysis Techniques

Subverting Memory Acquisition

Dementia by Luka Milkovic

An impressive advancement in "anti-analysis" research was presented by Luka Milkovic at the 29th Chaos Communication Congress in December 2012. His tool, Dementia, evades memory capture by intercepting NtWriteFile() calls through the use of inline hooking and a file system mini-filter. The buffer of a memory acquisition tool is manipulated so that any reference to the target process and its kernel objects is removed and the resultant memory image file has no evidence of this running process.

For more on this, visit: **https://events.ccc.de/congress/2012/Fahrplan/attachments/2231_Defeating%20Windows%20memory%20forensics.ppt**

Anti-Analysis: Spinning the Wheels of the Forensic Examiner

Attention Deficit Disorder by Jake Williams

Another anti-memory analysis POC is ADD (Attention Deficit Disorder), written by Jake Williams. This tool creates fake EPPROCESS, TCP_Endpoint, and FILE_OBJECT structures in memory that lead the examiner down rabbit holes where files may appear to be loaded into system memory or where network connections to rogue IP/domains may appear to exist. As with the arms race of malware sophistication and the reversing skills of our ninja malware engineers, anti-analysis techniques will continue to push the edge of forensic detection.

For more on this, visit: **http://malwarejake.blogspot.com/2014/01/analysis-of-add-ref-image-part-1.html**

Evasion of Malicious Code Detection Techniques

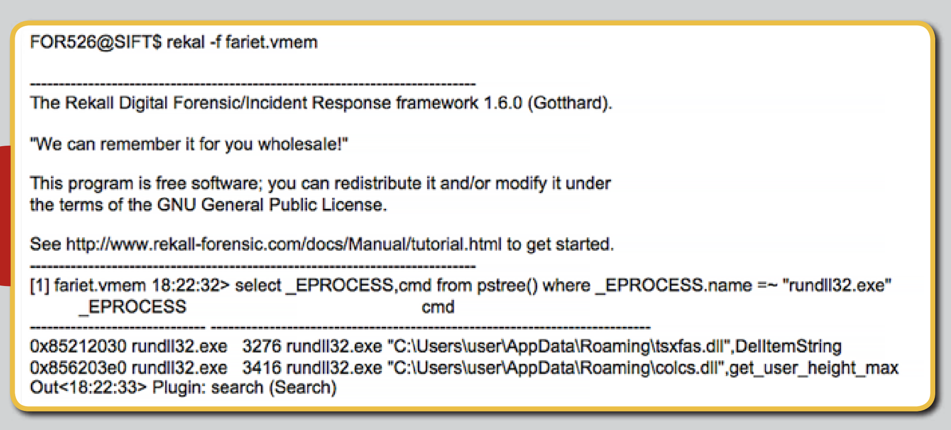
Gargoyle by Josh Lospinoso

One of the methods we use to identify code injection (see Step 4 above) is to look for executable memory that is not mapped to disk. Gargoyle implements a unique proof of concept evasion technique, writing malicious code into read/write only memory, then using an Asynchronous Procedure Call based on a timer that calls a ROP gadget to invoke VirtualProtectEx to change protections to RWX. After Gargoyle executes, it again calls VirtualProtectEx to return to RW protections to further evade detection.

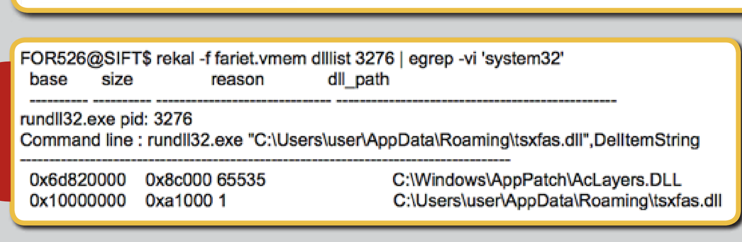
For more on this, visit: **https://github.com/JLospinoso/gargoyle**

Six-Step Investigative Methodology

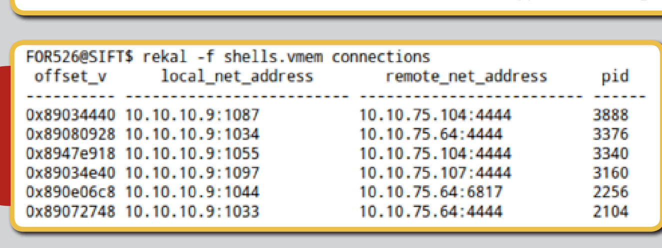
1 Identify rogue processes



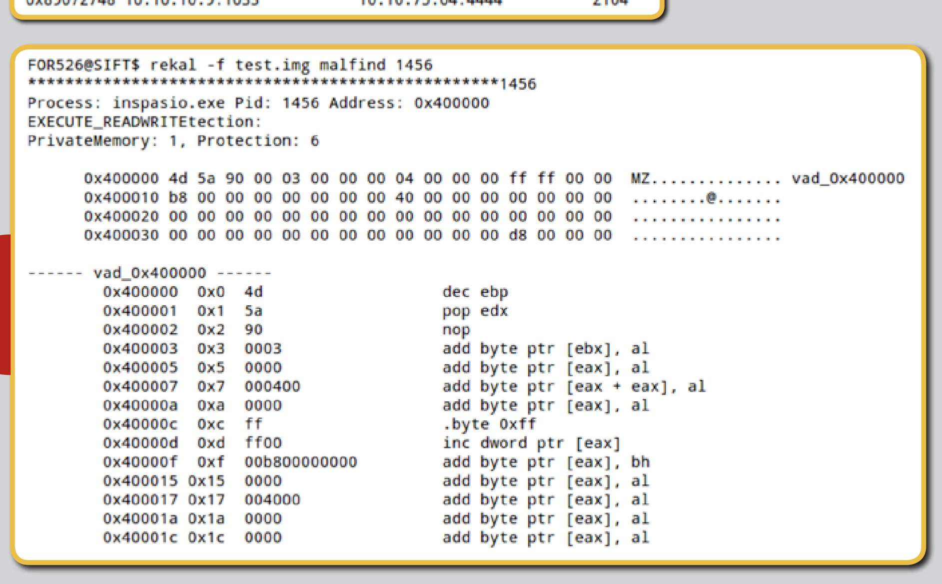
2 Analyze process DLLs and handles



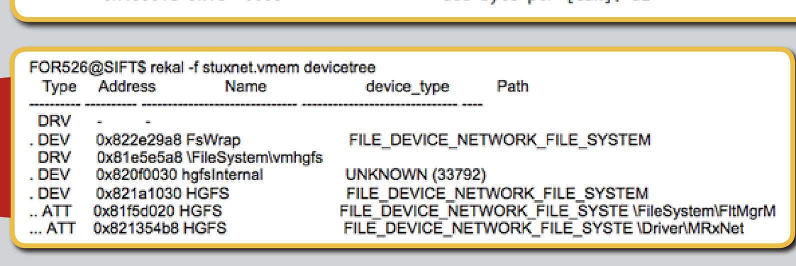
3 Review network artifacts



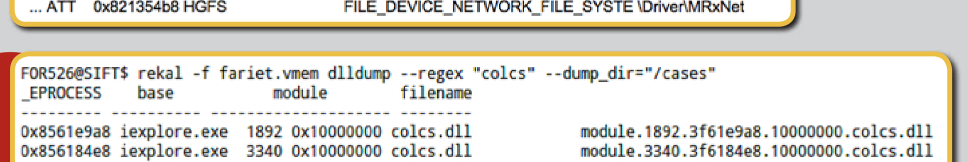
4 Look for evidence of code injection



5 Check for signs of a rootkit

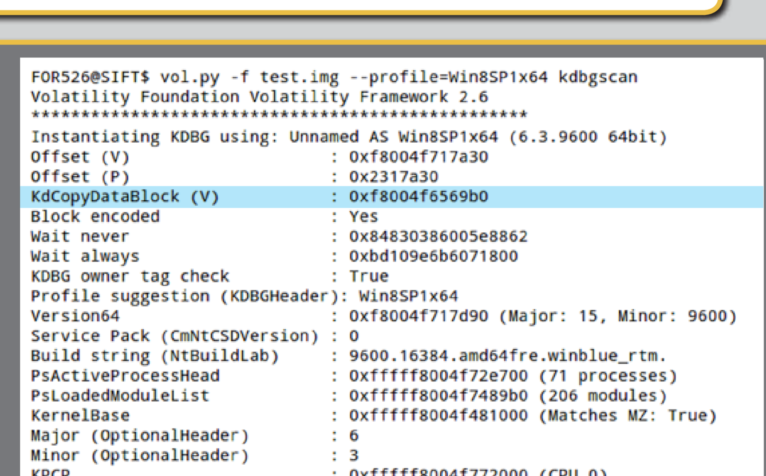


6 Dump suspicious processes and drivers



Tip

Tip for Parsing a Memory Image with an Encoded KDBG:
Windows 8 and later (x64) encode the KDBG, a key structure tremendously useful for memory forensics. To more easily analyze these memory images, an examiner should pay attention to the offset for the KdCopyDataBlock, identified with **kdbgscan**, to speed Volatility's ability to identify the KiWaitNever and KiWaitAlways values and interpret the KDBG data structure.



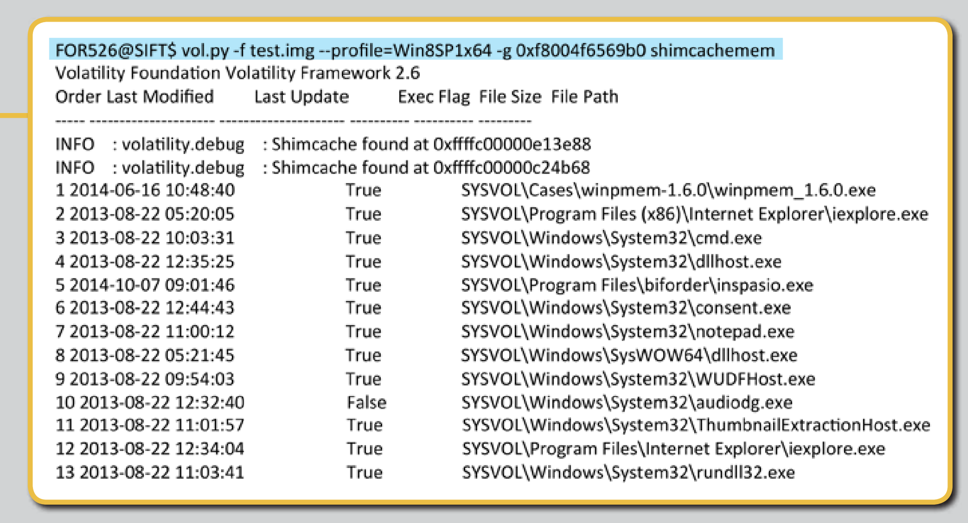
Advances in Memory Forensics

Recover Memory-Resident Evidence of Execution: *Shimcachemem*

by Fred House, Andrew Davis, and Claudiu Teodorescu

The use of shimcache artifacts in many investigations has been limited because data is not updated in the registry until the system is shut down. As a winning submission to the 2015 Volatility plugin contest, these researchers authored a parsing plugin that extracts these entries from the Application Compatibility Cache database in module or process memory. Despite changes in structure and the method of organization of these entries across versions of Windows, `shimcachemem` supports versions from WinXPSP2 to Windows2012R2.

```
$ vol.py -f test.img --profile=Win8SP1x64 -g 0xf8004f6569b0 shimcachemem
```

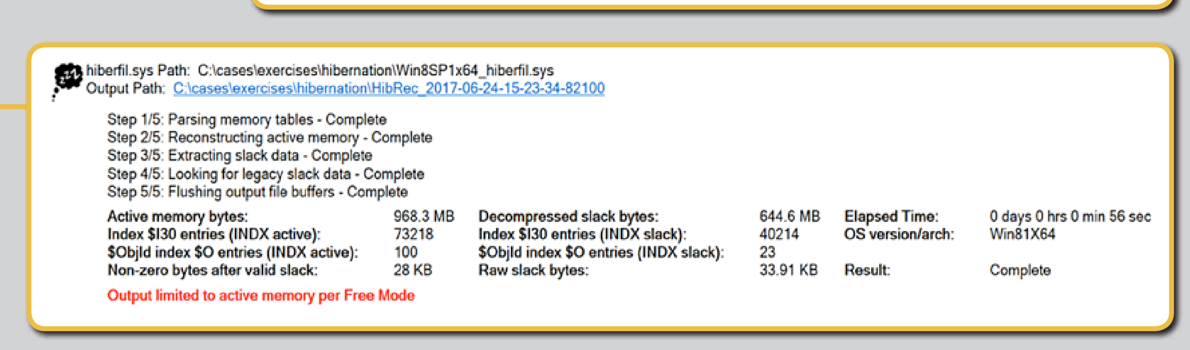


Decompress Win 8+ Hiberfil.sys and Carve Hibernation Slack: *Hibernation Recon*

Hibernation Recon by Arsenal Recon

Hibr2Bin by Comae Technologies

Hibernation files can be a treasure trove of forensic artifacts in investigations of all types. We encountered a hurdle to our analysis when Windows 8 introduced the LZ Huffman XPRESS compression method for storing the contents of physical memory for a hibernating machine. Our tools at the time could not decompress, barring us from unearthing system state analysis for the time of hibernation. Arsenal Recon and Comae Technologies introduced decompression tools recently that allow examiners to analyze this dataset.



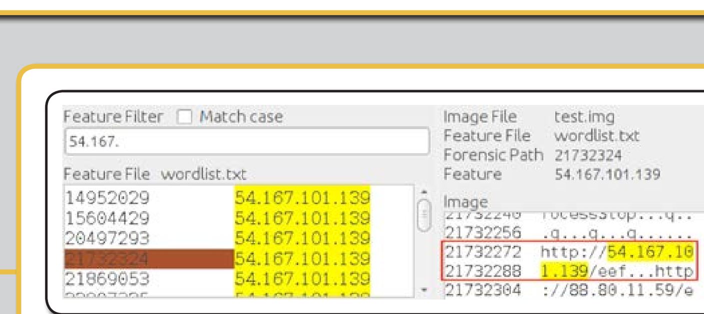
Physical to Virtual Address Translation

strings by Volatility Framework

ptov or **pas2vas** by Rekall

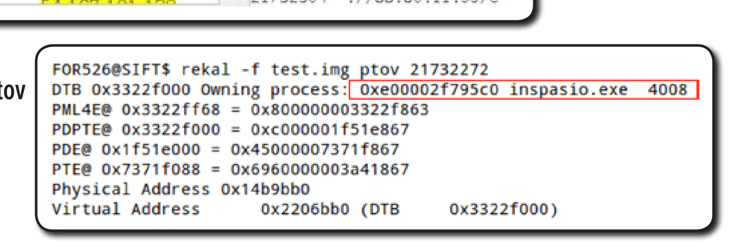
To map keywords identified by Bulk_Extractor or the strings tool, to their owning process or kernel module, we must perform physical to virtual address translation. Both Rekall and Volatility offer plugins that provide this ptov functionality. With Volatility, we can invoke the `strings` plugin. Rekall has two different plugins that offer physical to virtual address translation, `ptov` and `pas2vas`. These plugins employ different methods in determining which process has been allocated the frame in physical memory where the keyword lies. Regardless of the method used, the end result is a reverse lookup of keyword to owning process.

```
$ rekal -f test.img ptov 21732272
```



Bulk_Extractor

Rekall's ptov

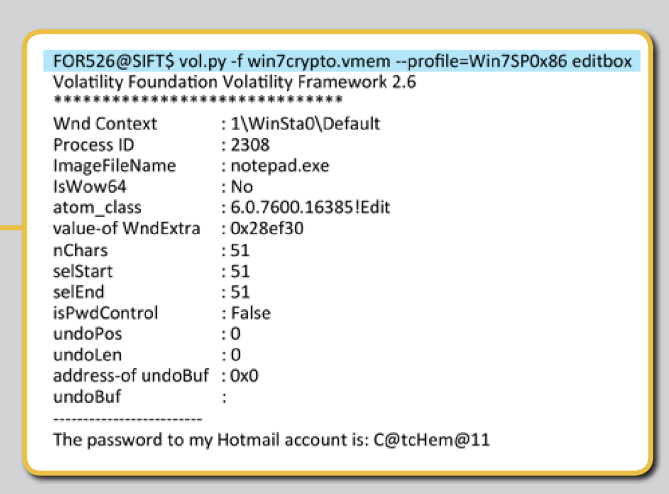


Recover Text from Windows Edit Controls

editbox by Adam Bridge

Extracting the relevant contents of applications with Edit controls, such as notepad was a difficult challenge until the introduction of the `editbox` plugin. Based on the research of Adam Bridge, we can now uncover urls fields, undo buffers, and undo text entered in the Run dialogue box.

```
$ vol.py -f memory.img --profile=<profile> editbox
```



Identify Known Malware Based on Import API Fuzzy Hashing: *impfuzzy*

impfuzzy by JPCERTCC

Signatures for malicious binaries extracted from the file system are not applicable to memory analysis, due to changes that occur when a PE file is loaded into memory. By using fuzzy hash of the Import API table, as performed by `impfuzzy`, we can identify the presence of previously signatured malware in new memory samples.

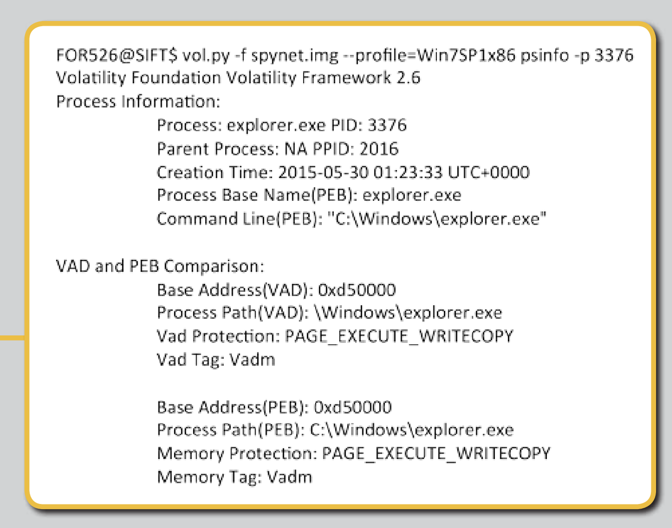
```
$ vol.py -f memory.img --profile=<profile> impfuzzy -p <pid>
```

Comprehensive Process and VAD Analysis

psinfo by Monnappa K A

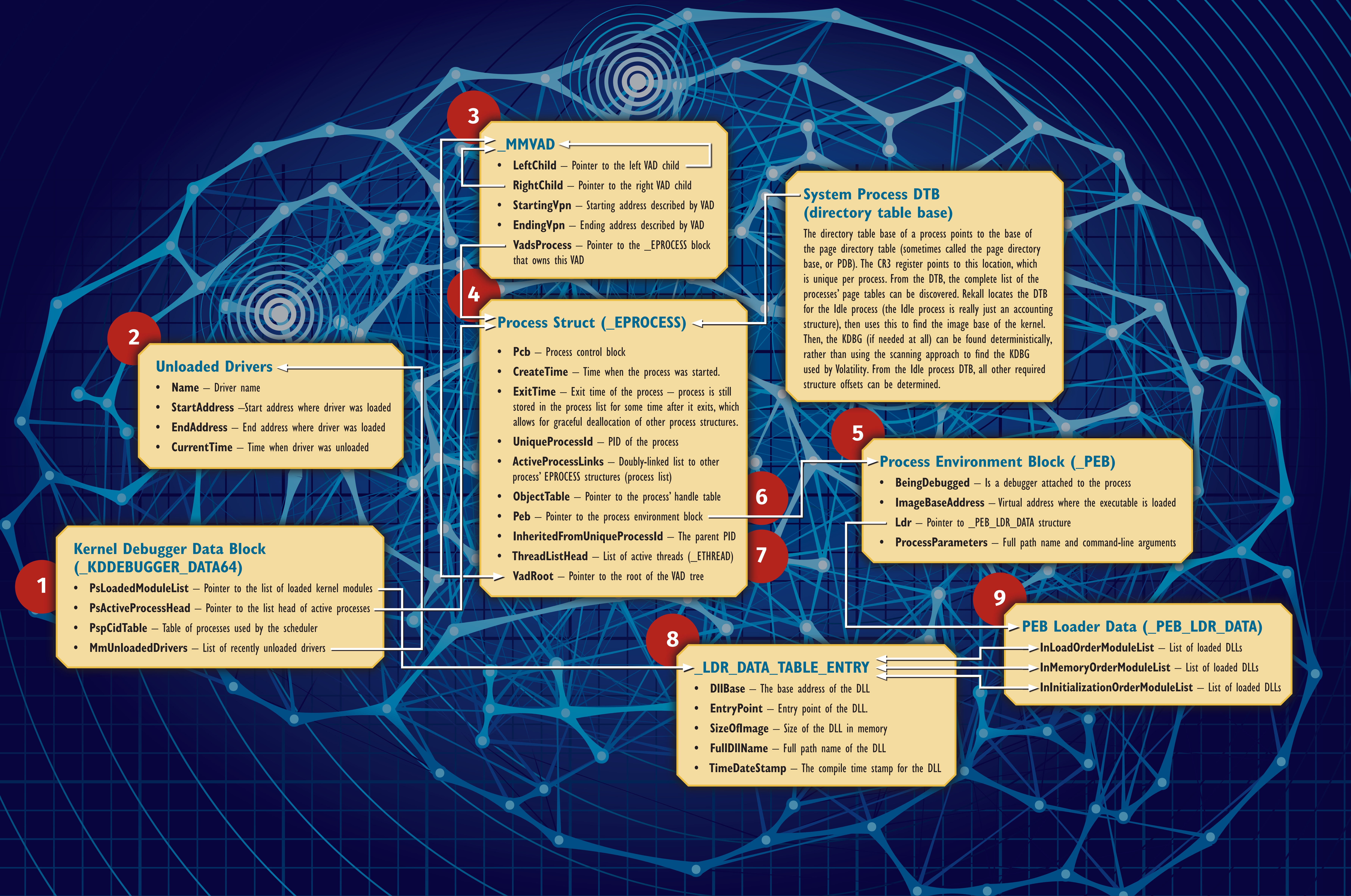
Often during memory analysis, an examiner will enumerate processes multiple ways in order to gain insight into its functions and characteristics. Instead of requiring multiple runs of different plugins, `psinfo` provides process and VAD analysis in one.

```
$ vol.py -f memory.img --profile=<profile> psinfo -p <pid>
```



What Lies Within: Windows Memory Analysis

We are in a cybersecurity arms race as incident responders, faced with a growing sophistication of threats, posed by actors both internal and external to our environment. Our ability to effectively and efficiently detect and contain malicious actors inside our environment hinges on visibility into the current system state of our endpoint. The details uncovered through memory analysis allows us to baseline normal functions and spot significant anomalies indicative of malicious activity. This poster provides insight into the most relevant Windows internal structures for forensic analysis. Though there are far more members of each structure than shown here, these are the most pertinent for spotting malicious activity and subversion.



Security Protections

Kernel Patch Protection (aka PatchGuard)

Modern x64 Windows implements a functionality called Kernel Patch Protection (sometimes referred to as PatchGuard). KPP checks key system structures, including (but not limited to) the doubly-linked lists that track most objects on Windows. In particular, KPP makes the DKOM rootkit technique of unlinking a process from the process list obsolete. When KPP detects an unauthorized modification, it causes a BSOD to halt the system. As a result, Windows kernel mode rootkits now use kernel callbacks, Asynchronous Procedure Calls (APCs), and Deferred Procedure Calls (DPCs) to run code instead of the old "launch a process and use DKOM to hide it" technique.

Kernel Object Obfuscation

Just as we do in memory forensics, many rootkits have relied on the KDBG to locate key operating system structures. As of Windows 8, the KDBG is encrypted to prevent rootkits from easily locating it. This does not impact operations since the KDBG is not used during normal system operation. If the system crashes, the KeBugCheck routine decrypts the KDBG before storing the crash dump data in the page file (making the KDBG available for debugging purposes). Kernel object headers are also encrypted in Windows 10. While intended to interfere with rootkits, this also has the effect of inhibiting some scanning plugins.

FOR526: Memory Forensics In-Depth

AUTHORS:
Alissa Torres *@sibertor*
Jake Williams *@malwarejake*

In today's enterprise investigations, memory forensics plays a crucial role in unraveling the details of what happened on the system. Recent large-scale malware infections have involved attackers implementing advanced anti-analysis techniques, making the system memory the battleground between offense and defense. Skilled incident responders use memory forensics skills to reveal "ground truth" of malicious activity and move more swiftly to remediation.

Learn more about **FOR526: Memory Forensics In-Depth** at www.sans.org/FOR526

1) PsLoadedModuleList

The PsLoadedModuleList structure of the KDBG points to the list of loaded kernel modules (device drivers) in memory. Many malware variants use kernel modules because they require low level access to the system. Rootkits, packet sniffers, and many keyloggers use may be found in the loaded modules list. The members of the list are _LDR_DATA_TABLE_ENTRY structures. Stuxnet, Duqu, Regin, R2D2, Flame, etc., have all used some kernel mode module component – so this is a great place to look for advanced (supposed) nation-state malware. However, note that some malware has the ability to unlink itself from this list, so scanning for structures may also be necessary.

REKALL PLUGINS: modules, modscan

2) Unloaded Modules

The Windows OS keeps track of recently unloaded kernel modules (device drivers). This is useful for finding rootkits (and misbehaving legitimate device drivers).

REKALL PLUGINS: unloaded_modules

3) VAD

VADs (Virtual Address Descriptors) are used by the memory manager to track ALL memory allocated on the system. Malware and rootkits can hide from a lot of different OS components, but hiding from the memory manager is unwise. If it can't see your memory, it will give it away!

REKALL PLUGINS: vad, vaddump

4) _EPROCESS

The _EPROCESS is perhaps the most important structure in memory forensics. The _EPROCESS structure has more than 100 members, many of them pointers to other structures. The _EPROCESS gives us the PID and parent PID of a given process. Analyzing PID relationships between processes can reveal malware. For more information, see the SANS DFIR poster "Know Normal, Find Evil." The _EPROCESS block also contains the creation and exit time of a process. Why would the OS keep track of exited processes? The answer is that when a process exits, it may have open handles which must be closed by the OS. The OS also needs time to gracefully deallocate other structures used by the process. The ExitTime field allows us to see that a process has exited but has not yet been completely removed by the OS. Note that the task manager and other live response tools will not show exited processes at all, but they are easy to see with use of memory forensics!

REKALL PLUGINS: pslist, psscan, pstree

5) Process Environment Block

The PEB contains pointers to the _PEB_LDR_DATA structure (discussed below). It also contains a flag that tells whether a debugger is attached to a process. Some malware will debug a child process as an antireversing measure. Finally, the PEB also contains a pointer to the command line arguments that were supplied to the process on creation.

REKALL PLUGINS: ldrmodules, dlllist, pstree verbosity=10

6) ObjectTable

For a process in Windows to use any resource (registry key, file, directory, process, etc.), it must have a handle to that object. We can tell a lot about a process just by looking at its open handles. For instance, you could potentially infer the log file a keylogger is using or persistence keys used by the malware, all by examining handles.

REKALL PLUGINS: handles, object_types

7) ThreadListHead

Where are the thread list structures on the poster? Sorry, we just don't have room to do them justice – but most investigations don't require us to dive into thread structures directly. Threads are still important. though. In Windows, a process is best thought of as an accounting structure. The Windows scheduler never deals with processes directly, rather it schedules individual threads (inside a process) for execution. Still, you'll find yourself using process structures more in your investigations.

REKALL PLUGINS: thrdscan, threads

8) _LDR_DATA_TABLE_ENTRY

This structure is used to describe a loaded module. Loaded modules come in two forms: the kernel module (aka device driver) and dynamic link libraries (DLLs), which are loaded into user mode processes.

REKALL PLUGINS: modules, ldrmodules, dlllist

9) PEB Loader Data

This structure contains pointers to three linked lists of loaded modules in a given process. Each is ordered differently (order of loading, order of initialization, and order of memory addresses). Sometimes malware will inject a DLL into a legitimate Windows service, then try to hide. But they'd better hide from all three lists or, you'll detect it with no trouble.

REKALL PLUGINS: ldrmodules

Note that many internal OS structures are doubly-linked lists. The pointers in the lists actually point to the pointer in the next structure. However, for clarity of illustration, we have chosen to show the type of structure they point to. Also, note that the PsActiveProcessHead member of the KDBG structure points to ActiveProcessLinks member of the _EPROCESS structure. However, for clarity, we depict the pointer pointing to the base of the _EPROCESS structure. We feel that this depiction illustrates this more clearly.