

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

digital-forensics.sans.org

DFIR-Memory_V21_7-17

FOR500
Windows Forensics (Formerly FOR408)
GCFE

FOR508
Advanced IR and Threat Hunting
GCSA

FOR518
Mac Forensics

OPERATING SYSTEM & DEVICE IN-DEPTH

INCIDENT RESPONSE & THREAT HUNTING

FOR572
Advanced Network Forensics and Analysis
GNFA

FOR578
Cyber Threat Intelligence

FOR526
Memory Forensics In-Depth

FOR610
REM: Malware Analysis
GEM

FOR585
Advanced Smartphone Forensics
GASF

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

[@sansforensics](#)

[sansforensics](#)

[dfir.to/DFIRCast](#)

[dfir.to/gpl-us-sansforensics](#)

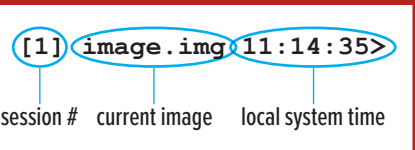
[dfir.to/MAIL-LIST](#)

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
\$ rekall -f image.img pslist
Starting an Interactive Session
\$ rekall -f image.img



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="PsActiveProcessList"
Customize pslist output with filters [1] image.img 11:14:35 pslist 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

DLLIST
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 describe (pstree) - View columns to output
HANDLE
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"
FILES
Scan memory for _FILE_OBJECT handles [1] image.img 11:14:35 filescan output="filescan.txt"

Malicious Code Detection

IDENTIFY SUSPICIOUS PROCESSES BY COMMAND LINE
PS (WITH VERBOSITY) - List processes with path and command line [1] image.img 11:14:35 select _EPROCESS_ppid_cmd_path from pstree() [1] image.img 11:14:35 select _EPROCESS_ppid_cmd_path from pstree() DETECT CODE INJECTION BY VAD ANALYSIS
MALFIND
Find injected code and dump sections Positional Argument: Show information only for specific PIDs
-pid=<pid>
phys_offset=<offset> Provide physical offset of process to scan
epprocess=<process> Provide virtual offset for process to scan
dump_dir=<dir> Directory to save memory sections [1] image.img 11:14:35 malfind epprocess=0x853cf460,dump_dir="/cases"
LDRMODULES
Detect unlinked DLLs Verbose: show full paths from three DLL lists [1] image.img 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
\$ rekall -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 EPROCESS, 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

Identify rogue processes

FOR526@SIFTS rekall -f farief.vmem
The Rekall Digital Forensic/Incident Response Framework 1.6.0 (Gothard).
"We can remember it for you wholesale!"
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License.
See <http://www.rekall-forensic.com/docs/Manual/tutorial.html> to get started.
[1] farief.vmem 18:22:32 select _EPROCESS.cmd from pstree() where _EPROCESS.name = "rundll32.exe" _EPROCESS cmd
0x85212030 rundll32.exe 3276 rundll32.exe "C:\Users\User\AppData\Roaming\txsfxs.dll",DllItemString
0x8520360 rundll32.exe 3416 rundll32.exe "C:\Users\User\AppData\Roaming\colcs.dll",get_user_height_max
Out:18:22:33> Plugin: search (Search)

Analyze process DLLs and handles

FOR526@SIFTS rekall -f farief.vmem dllist lpath | egrep -v 'system32' base size reason dll_path
rundll32.exe pid: 3276
Command line : rundll32.exe "C:\Users\User\AppData\Roaming\txsfxs.dll",DllItemString
0x85202000 0x8c000 65535 C:\Windows\AppPatch\AcLayers.DLL
0x10000000 0xa1000 1 C:\Users\User\AppData\Roaming\txsfxs.dll

Review network artifacts

FOR526@SIFTS rekall -f shells.vmem connections
offset_v local_net_address remote_net_address pid
0x9034440 10.10.10.9:1087 10.10.75.104:4444 3888
0x9030928 10.10.10.9:1034 10.10.75.64:4444 3376
0x8547918 10.10.10.9:1055 10.10.75.104:4444 3340
0x9034440 10.10.10.9:1097 10.10.75.107:4444 3160
0x89006c8 10.10.10.9:1044 10.10.75.64:6817 2256
0x902748 10.10.10.9:1033 10.10.75.64:4444 2104

Look for evidence of code injection

FOR526@SIFTS rekall -f test.img malfind 1456

Process: inspasio.exe Pid: 1456 Address: 0x400000
EXECUTE_READWRITE: 0x400000
PrivateMemory: 1, Protection: 6
0x400000 4d 5a 90 00 03 00 00 04 00 00 ff ff 00 00 MZ..... vad_0x400000
0x400010 b8 00 00 00 00 00 00 00 00 00 00 00 00@.....
0x400020 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400030 00 00 00 00 00 00 00 00 00 00 00 00 00
..... vad_0x400000
0x400000 0xd 4d pop ebx
0x400001 0x1 5a file name
0x400002 0x2 90 nop
0x400003 0x3 0003 add byte ptr [eax], al
0x400004 0x4 0009 add byte ptr [eax], al
0x400007 0x7 00040d add byte ptr [eax+eax], al
0x40000a 0xa 0000 add byte ptr [eax], al
0x40000c 0xc ff ,byte off
0x40000d 0xd ffff inc dword ptr [eax]
0x40000f 0xf 00b080000000 add byte ptr [eax], bh
0x400015 0x15 0000 add byte ptr [eax], al
0x400017 0x17 004000 add byte ptr [eax], al
0x40001a 0x1a 0000 add byte ptr [eax], al
0x40001c 0x1c 0000 add byte ptr [eax], al
0x40001e 0x1e 0000 add byte ptr [eax], al

Check for signs of a rootkit

FOR526@SIFTS rekall -f stuxnet.vmem devicetree
Type Address Name device_type Path
DEV 0x22e29a8 FaWin FILE_DEVICE_NETWORK_FILE_SYSTEM
DEV 0x21e0e0a8 FileSystemVmhgfs UNKOWN (33792)
DEV 0x2202030 hfsNameMail
DEV 0x21a1020 HGFS FILE_DEVICE_NETWORK_FILE_SYSTEM
ATT 0x819e020 HGFS FILE_DEVICE_NETWORK_FILE_SYSTEM FileMgrM
ATT 0x21354b8 HGFS FILE_DEVICE_NETWORK_FILE_SYSTEM UserMgrM

Dump suspicious processes and drivers

FOR526@SIFTS rekall -f farief.vmem dldump --regex "colcs" --dump_dir="/cases"
0x8561e9a8 explorer.exe 1892 0x10000000 colcs.dll module: 1892.3f61e9a8.10000000.colcs.dll
0x856184e8 explorer.exe 3340 0x10000000 colcs.dll module: 3340.3f6184e8.10000000.colcs.dll
0x856203e0 rundll32.exe 3416 0x10000000 colcs.dll module: 3416.3f6203e0.10000000.colcs.dll

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 supply the offset for the KdCopyDataBlock, identified with **kdbgsan**, to speed Volatility's ability to identify the KiWaitNever and KiWaitAlways values and interpret the KDBG data structure.

```
FOR526@SIFTS vol.py -f test.img --profile=Win7SP1x86 kdbgsan
Volatility Foundation Volatility Framework 2.6
*****
Instantiating KDBG using: Unnamed AS Win7SP1x64 (6.3.9600.64bit)
Offset (v) : 0xf8004717a30
Offset (p) : 0x2317a30
KdCopyDataBlock (v) : 0xf8004f6569b0
Block encoded : Yes
Wait never : 0x8483038600e8862
Wait always : 0xb019e0eb6071800
KDBG owner tag check : True
Profile suggestion (KDBGHeader): Win7SP1x64
Version64 : 0xf8004717a30 (Major: 15, Minor: 9600)
Service Pack (CMNCSVersion): 0
Build string (NTBuildLab): 9600.16384.amd64fre.winblue_rtm.
PsActiveProcessHead : 0xfffff800472e700 (71 processes)
PsLoadedModuleList : 0xfffff8004748900 (216 modules)
KernelBase : 0xfffff8004772000 (Matches MZ: True)
Major (OptionalHeader): 6
Minor (OptionalHeader): 3
MPCR : 0xfffff8004772000 (CPU 0)
KPCR : 0xfffff8000207e000 (CPU 1)
```

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=Win7SP1x64 -g 0xf8004f6569b0 shimcachemem

```
FOR526@SIFTS vol.py -f test.img --profile=Win7SP1x64 -g 0xf8004f6569b0 shimcachemem
Volatility Foundation Volatility Framework 2.6
Order Last Modified Last Update Exec Flag File Size File Path
INFO : volatility.debug : Shimcache found at 0xffff0000e13e88
INFO : volatility.debug : Shimcache found at 0xffff0000c24b68
1 2014-06-16 10:48:40 True SYSVOL\Cases\winpmem-1.6.0\winpmem_1.6.0.exe
2 2013-08-22 05:20:05 True SYSVOL\Program Files (x86)\Internet Explorer\explorer.exe
3 2013-08-22 10:03:31 True SYSVOL\Windows\System32\cmd.exe
4 2013-08-22 12:35:25 True SYSVOL\Windows\System32\dlhost.exe
5 2014-10-07 09:01:46 True SYSVOL\Program Files\lbforder\inspasio.exe
6 2013-08-22 12:44:43 True SYSVOL\Windows\System32\consent.exe
7 2013-08-22 11:00:12 True SYSVOL\Windows\System32\notepad.exe
8 2013-08-22 05:21:45 True SYSVOL\Windows\System32\WUDFHost.exe
9 2013-08-22 09:54:03 True SYSVOL\Windows\System32\WUDFHost.exe
10 2013-08-22 12:32:40 False SYSVOL\Windows\System32\audiiodg.exe
11 2013-08-22 11:01:57 True SYSVOL\Windows\System32\ThumbnailExtractionHost.exe
12 2013-08-22 12:34:04 True SYSVOL\Program Files\Internet Explorer\explorer.exe
13 2013-08-22 11:03:41 True SYSVOL\Windows\System32\winll32.exe
```

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.

```
liberfil.sys Path: C:\cases\exercises\hibernation\Win8SP1x64_hiberfil.sys
Output Path: C:\cases\exercises\hibernation\HiberRecon_2017-08-24-1530-34-82100
Step 1/5: Parsing memory tables - Complete
Step 2/5: Reconstructing active memory - Complete
Step 3/5: Extracting slack data - Complete
Step 4/5: Looking for legacy slack data - Complete
Step 5/5: Flushing output file buffers - Complete
Active memory bytes: 968.3 MB
Index $30 entries (INDX active): 73218
$30jnd index $0 entries (INDX active): 100
Non-zero bytes after valid slack: 28 KB
Decompressed slack bytes: 644.6 MB
OS version/arch: Win8/64
CBJid index $0 entries (INDX slack): 23
Raw slack bytes: 3391 KB
Result: Complete
Output limited to active memory per File Mode
```

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.

\$ rekall -f test.img ptov 21732272

```
Feature Filter [ ] Match case Image File test.img
[54.167.101.139] Feature File wordlist.txt
[14952829] Feature Path 21732272
[15684429] [54.167.101.139] [21732272] http://54.167.101.139/
[20497293] [54.167.101.139] [21732272] http://54.167.101.139/
[21896853] [54.167.101.139] [21732272] http://54.167.101.139/
[21925294] [54.167.101.139] [21732272] http://54.167.101.139/
Rekall's ptov
FOR526@SIFTS rekall -f test.img ptov 21732272
DB: 0x3322f000 Owing process : 0xe90002f795c0 inspasio.exe 4008
PML4E: 0x3322f000 = 0xc0000003322f693
POPE: 0x3322f000 = 0xc000001f51e867
PDE: 0x1f51e000 = 0x4500000731f1867
PTE: 0x731f088 = 0xc0000000341867
Physical Address 0x14b9bb0
Virtual Address 0x2206bb0 (DTB) 0x3322f000
```

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

```
FOR526@SIFTS vol.py -f win7crypto.vmem --profile=Win7SP0x86 editbox
Volatility Foundation Volatility Framework 2.6
*****
Wind Context : 1\WinSta0\Default
Process ID : 2308
ImageFileName : notepad.exe
IsWow64 : No
atom_class : 6.0.7600.16385>Edit
value-of-WndExtra : 0x28ef30
nChars : :51
selStart : :51
selEnd : :51
isPwdControl : False
undoPos : :0
undolen : :0
address-of-undoBuf : 0x0
undoBuf :
The password to my Hotmail account is: @tChem#11
```

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>

```
FOR526@SIFTS vol.py -f spyenet.img --profile=Win7SP1x86 psinfo -p 3376
Volatility Foundation Volatility Framework 2.6
Process Information:
Process: explorer.exe PID: 3376
Parent Process: NA PPID: 2016
Creation Time: 2015-05-30 01:23:33 UTC+0000
Process Base Name(PEB): explorer.exe
Command Line(PEB): "C:\Windows\explorer.exe"
```

```
VAD and PEB Comparison:
Base Address(VAD): 0x50000
Process Path(VAD): C:\Windows\explorer.exe
Vad Protection: PAGE_EXECUTE_WRITECOPY
Vad Tag: Vadm
Base Address(PEB): 0x50000
Process Path(PEB): C:\Windows\explorer.exe
Memory Protection: PAGE_EXECUTE_WRITECOPY
Memory Tag: Vadm
```

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: thrdsan, 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.