**Memory Forensics Training Writeup**

Hyberfile.sys = found in c\ complete copy of everything in ram when the lid was closed

* If the system is shutdown, you have a chance to review memory. If the system is running, you might have two memory images to analyze.
* Use the imagecopy plugin to convert the hyberfile.sys or crash dumps to a raw format.

Memory.dmp = in c\windows file and contains memory crash dump info

paigefile.sys/swapfile.sys = contains parts of memory that were paged to disk

* Pagefile and infrequently used areas of RAM are compressed, so we won’t be able to pull human-readable strings. The compression algorithm is known so we can use a tool to scan memory for signs of compression and output uncompressed data. **Winmem\_decompress.py** is a tool that can be used to uncompress memory, but cannot be analyzed using Volatility. Use string searches, carving, or YARA scanning for examining uncompressed data. An alternative is to use [Fireeye’s **win10memcompression.py**](https://github.com/mandiant/win10_volatility) as a volatility overlay to decompress memory pages as compression is encountered.

**Step 1 - Obtain Image Profile**

KDBG = data structure whose pointers can be followed to find the process list for a system.

* Can be found by finding the kernel processor control region (KPCR), which points to KDBG, or by searching for well known KDBG signatures.
* Once KDBG is found, it leads to the executive process block list (EPROCESS) by identifying the PsActiveProcessHead pointer (a list of all currently running processes in memory).
  + EPROCESS points to PEB (see below), handles, access tokens, and threads.
  + EPROCESS points to Virtual Address Descriptor (VAD) tree that tracks all memory sections (aka memory pages) assigned to that process.
    - VAD tree allows memory analysis tools to double check what exists in various memory sections for a process vs. what the various lists say are present.
    - Important bc if discrepancies are found, this could indicate code injection.

Each process has its own Process Environment Block (PEB) - holds data structures that define a process such as full path of executable, process spawn command line, and linked list of loaded DLLs.

Volatility requires an OS profile. Kdbgscan and imageinfo are best options for identifying which OS build you are working with.

* kdbg scan searches memory image for matches of known KDBGHeader signatures for a variety of different OS profiles.
* Starting in Win8 the KDBG structure is encrypted, so the plugin needs to find KdCopyDataBlock to derive a decryption key, making kdbgscan a slow plugin.
* Imageinfo provides some information KDBG scan provides but also provides time of memory capture

**Memory forensics process**

* Review processes
* Scrutinize helper objects assigned to processes
  + Loaded DLLs, files, registry keys, mutants, network sockets
* Search for code injection (malware taking over a process)
* Search for rootkits
* Extract suspicious processes, drivers, and memory pages

**Step 2 - Find Processes**

Process information is tracked by the operating system using the Executive Process Block (EPROCESS).

* Process name
* PID/PPID
* Memory offset (location in memory)
* Creation/Termination time
* Threads and Handles assigned to process
* Link to Virtual Address Descriptor tree and link to Process Environment Block

Kernel uses doubly linked lists to track processes. There is a flink (forward link) and blink (back link) in each EPROCESS block that helps link processes. Only currently running processes are found in the doubly linked list. Malware with kernel access can unlink itself using rootkit techniques such as Direct Kernel Object Manipulation. With that said, our tools must scan the entire memory for orphaned process blocks, not just scan the doubly linked EPROCESS list.

When analyzing a process look at

* Image name
  + Spelled correctly? Legitimate Process? Matches System Context?
* Full Path
  + Appropriate path for system executable? Running from user/temp dir?
    - Explorer.exe should always spawn from Windows Directory
    - IEExploer always from program files
    - Be aware of processes spawning from system32 folder
* Parent Process
  + Is the parent process what you would expect?
    - Process spawned from explorer happened after login
    - Process spawned from services happened at boot (persistance)
* Command line
* Start time
  + Did it spawn at boot? Compare to smss.exe or winlogon.exe that starts at boot.
* SID
  + Tells us what level of account spawned the process (system vs user vs LocalService)

Pslist

**Pslist** works by finding the EPROCESS list in the kernel, following the doubly linked list, and parsing formation about each process. This is the most basic plugin and might not find processes hidden by rootkits.

Psscan

Works by scanning the memory for objects, similar to data carving. **Psscan** has the capability of identifying terminated processes in the memory slack space, as well as processes hidden by rootkits. **Psscan** gives us the page directory base offset (PDB), which allows virtual address space for a process to convert to physical RAM space.

Physical offsets are the actual location of the EPROCESS block in memory. You could observe this information by opening the memory image in a hex editor and navigating to that offset. Virtual memory is what is being used by the running system - they are the location in the virtual memory (physical memory + all of its virtual pages). Some volatility plugins require a virtual offset or physical offset, and if you provide the wrong one you will get the wrong information. If you try to dump a terminated process by PID, try dumping by the physical offset.

Notes on WMI and Powershell

Two primary WMI event consumers abused are CommandLine and ActiveScript consumers. CommandLine kicks off a new WmiPrvSE process to run an executable or Powershell script. Scrcons.exe (script consumers.exe) is the parent of any ActiveScript consumer such as VBScript or Jscript. WmiPrvSE.exe should always be looked into.

Wsmprovhost.exe indicated powershell remoting activity on the target system.

* Enter-PSSession
* Invoke-Command
* New-PSSession
* Any powershell cmdlet with “-computer name” parameter
  + set -service, clear-eventlog, get-wmiobject

If the user executes powershell, the parent process will be explorer.exe, and if powershell runs with a parent of svchost.exe, its an indicator that an admin account was used. Also be aware of powershell being spawned from cmd.exe, which could be indicative of a launch via backdoor or command shells.

**Malprocfind**

Automates checking for common anomalies. Checks for several common system processes (smss, crss, winlogon, services, lsass, svchost, spoolsv, wininit, ect) for anomalies. Failed checks are marked as false.

Proper parent process, Naming permutations, Path, Process priority, Command line, User SID, Sessions identifier, execution time (at boot?) processes spawned by cmd.exe, evidence of process hollowing, suspicious paths (temp folder), missing parents, and expected number of processes.

Baseline

**Baseline** plugin allows the analyst to compare a baseline known good image with a suspect image. It accepts two parameters, the baseline image and the suspect image.

**Step 3 - Analyzing Process Objects**

When analyzing Process objects, review process objects from suspicious processes only, as process objects can be in the hundreds or thousands.

The following process objects can be reviewed:

* DLLs - define the capabilities of a process e.g. if a process needs to communicate via HTTP it will load WININET.dll (some malware will being its own malicious DLLs to take control of a process
* Handles - pointer to a resource
  + File handles - identify which file system or I/O devices are being access by a process
  + Directory handles - not file system directory, instead these are known lists in teh kernel that allow a process to find kernel objects e.g. KnownDlls, BaseNameObjects, Callbacks, Devices, and Drivers.
  + Registry handles - registry keys a process is reading or writing to.
  + Mutex or semaphore handles - these control or limit access to a resource e.g. a mutex might be used by an object to enforce only one process at a time using it e.g. 2 worms use mutexes to mark a compromised system so it does not get reinfected.
  + Event handles - a way for process threads to communicate. Malware occasionally uses unique event handles.
* Threads - a process is just a container for all of the items that do the real work. Multiple threads run within every process interacting with various system objects
* Memory sections - every process has a collection of virtual memory pages where DLLs and files are loaded and code is stored. The Virtual Address Descriptor (VAD) tree maintains a list of all assigned memory sections
* Sockets - Network connection endpoints. Every network socket is assigned to a process.

DLLlist

When using **DLLlist**, you can use the -p (process ID) option to limit the information to a comma separated list of processes. The Dlls for each process number is in the hundreds, so use DLLlist on a subset of processes you are unsure about. DLllist provides:

* Base offset (can be used with dlldump to extract a specific DLL)
* DLL size
* Load count
* Load time (can help detect dlls loaded after execution time e.g anomalies like DLL injection)
* DLL file path
* Full command line parameters

The plugin reports information only found in the process environment block, so it won’t identify reflective DLL injection, which circumvents updating the PEB list. Use LDRmodules plugin to provide additional information to reveal more advanced attacks.

**Getsids**

Because every process inherits an access token from the account spawning it (token is a collection of SIDs that describe the user permissions and user groups), a review of the process tokens can help ID malware. Be aware of processes named as legitimate system processes but spawned under a user SID. Use the information in the SAM or SOFTWARE registry hive to match up user SID with suspicious processes to pinpoint compromised accounts.

Notes on SIDs

Each running process is assigned an access token that contains user account SIDs, SIDs for every group that user is a member of, and a list of privileges and ACLs related to the user account. Nearly all Windows system processes are spawned by built-in windows accounts. Find SIDs for a specific process by using the -p flag. The three most popular systems accounts are:

* LocalSystem - S-1-5-18
* LocalService - S-1-5-19
* NetworkService - S-1-5-20

Once you find a potentially compromised account, you can pivot on other processes that were spawned with that user account

* vol.py -f imagexx.raw --profile=Win10x64\_16299 **getsids** | grep -i accountxx

Handles

**Handles** are reviewed for specific suspicious processes. They can be used to confirm suspicion, specifically file and registry handles. The -p flag is used to search for a subset of processes, the -s flag removes unnamed handles (typically only used internally to the process and are uninteresting in memory forensics).

Use -t to specify handles (note: case sensitive)

* Process, Thread, Key (registry keys), Event, File, Mutant, Semaphore, Token, WmiGuid, Port, Thread, Directory, WindowStation, IOCompletion, Timer.

Note on Files/Named Pipes:   
Having a named pipe to another workstation y seems anomalous. Named pipes are increasingly being used by advanced adversaries and malware to tunnel command and control and data transfer over SMB/RPC, replacing (somewhat) noisier TCP socket connections.

**Filescan** and **mutantscan** plugins are used for more rigorous searching of file handles and mutants. They search for the markers indicating FILE\_OBJECTS and \_KMUTANT objects and return the respective results.

Note on mutants  
Because mutants are named to blend in, they are mostly used after the malware has been identified and used as an IOC to search for other compromised systems.

**Netscan**

We can use netscan (use connscan for legacy windows) to review relevant connections such as CNC

* 389: Lightweight Directory Access Protocol (LDAP)
* 443: HTTP over SSL/TLS
* 445: SMB (Microsoft File Sharing)
* 3389: Microsoft Remote Desktop Protocol
* 5985: PowerShell Remoting
* 8080: Alternative port for HTTP (Similar to port 80; Stark Research Labs employs a web proxy. Any traffic using port 8080 to 172.16.4.10 is traversing that proxy)

Notes on network connections:

Network sockets have a creation time, which can be useful to pivot on e.g. related processes and threads created at the same time. We also have the ability to see older terminated network sockets that might persist in memory. All network sockets and TCP connections are tied directly to a process. Take note on network connections from processes that should only communicate locally or not at all. Final note - network collection is the only OS dependent plug in: use netscan for Win7+, conscan/sockscan for WinXP/Server 2003.

Look out for

* Processes not web browser communicating via 80, 443, or 8080
* Browsers not communicating via 80, 443, or 8080
* Connections to unexplained internal/external IP address.
* Naked IP connections (connections to an IP, not a domain)
* RDP connections (note, external RDP connections are usually routed through a VPN concentrator)
* DNS requests to unusual domains
* Workstation to workstation connections.
  + Workstations don’t typically RDP, map shares, or authenticate to other workstations. It’s usually a workstation to server. This activity can indicate lateral movement.

vol.py -f imagexx.img --profile=Win10x64\_16299 netscan | egrep -i 'CLOSE|ESTABLISHED|Offset'

**Step 4 - Looking for Code Injection**

Code injection is easy to spot via memory analysis, but not other normal analysis means. Running a process inside another process allows malware to inherit access to the injected process’s memory section and permissions (e.g. cred dumpers injected into LSASS to get access to hashes and tickets). Code is often injected into persistent processes (e.g. not internet explorer - the user might close the browser and the malware will lose access).

Two popular forms of code injection are DLL injection and process hollowing.

DLL injection

To perform DLL injection, the attacker must have admin or debug privileges on the system (the attacking process token must have SeDebugPrivilage. Admin accounts have this privilege by default). The attacking process can allocate space in a running process, write a DLL file to load in the new space, and then create a new thread to load the DLL into the running process using native Windows function calls such as VirtualAllocEx() and CreateRemoteThread(). Alternatively, the attacking process can force a running process to load a DLL by hooking its filter function using the SetWindowsHookEx() function. Metasploit and Powershell can perform reflective injection by creating its own loader, bypassing many common API functions (which are heavily policed by security software), which results in code running that is not registered with any of the host process lists. Most DLLs are loaded from the system32, program files, or WinSxS folder. Outliers may be deemed necessary for further investigation.

Process Hollowing

During process hollowing, malware starts a copy of a legitimate system process, but in a suspended state. Some of the original code is deallocated and replaced with malicious code. Finally the process is started, and items like process name, path, and command line remain unchanged (camouflaging the malware). During process hollowing, the injected process image EXE is not backed with a file on disk.

**Ldrmodules**

Detects unlinked DLLs (common advance malware process). Each process is represented by the EPROCESS block which links to the PEB, which contains three doubly linked lists for tracking a process’s loaded DLLs. Ldrmodules outputs the process ID, name, PEB InLoadOrderModuleList (‘InLoad’), PEB InINitilazationOrderModule List (‘InInit’), PEB InMemoryOrderModule List (‘InMem’), and VAD Tree MappedPath. Unlinking a DLL from any of the PEB lists will hide injected DLLs and will not appear in plugins like dlllist. Unlinking DLLs does not require system or debug privileges. **Ldrmodules** scans all of a process’s memory section for portable executable headers and compares the results with information present in the three PEB lists, as well as comparing the DLL path pulled from the VAD tree for comparison. A “true” in a column means the DLL is present, a false means a DLL was not present in the list.

Some legit DLLs will not be present in some of the lists e.g. lsass.exe will be missing from the InInitilizedOrder list. No information in the “Mapped Path” column indicates the DLL was not loaded from disk (we expect some legit DLLs loaded using the windows API, hence no full path information). This is usually an indicator of DLL injection, even if the unnamed DLL is present in all three linked lists. Use the -v flag to gain more information (PEB keeps full path information) or plan to dump the memory section for analysis.

Be on the lookout for processes with no MappedPath. This is an indication that the binary was not loaded from disk (not loaded using the Windows API) and was injected. This helps malware circumvent antivirus and forensic analysis.

Ldrmodules | grep false will give us a quick view of loaded executables and any other output that was not present in at least one of the PEB lists.

False positives might include unloaded DLLs that have not yet been removed from process memory or files with strange extensions (many other file extensions share the PE header format).

Malfind

Reflective Code Injection is a way for code to run in a remote process without using the Windows Loader. By excluding the LoadLibrary, code is not registered in standard in-memory lists and is hidden from most security tools, as well as evading loading code from disk. Powersploit, and Empire all use reflective injection. Merterperter (metasploit payload) is also a good example of malware that uses reflective DLL injection.

To find Reflective Code Injection, **ldrmodules** follows the VAD tree of each process and reviews the receptive memory section. **Malfind** will search for unusual permissions (Page\_Execute\_ReadWrite) and if the file is unmapped (not backed with a file on disk). A memory section with no code should not be with executable permissions. Finally it checks if the actual code is present in memory (via PE file or shellcode).   
  
**Malfind** will dump to a directory assigned by the user which can then be scanned with anti-malware signatures or disassembled. The dumped files are named according to the physical offset of the injected process and the location of the suspicious memory page within the process memory space.

Use strings to get a field triage of the dumped memory sector

strings -a -t d -e l process.0xffff8c88af601580.0x2be0000.dmp >> p.uni

cat p.uni

(the -e l option provides Unicode strings, you can run the command again without that option to see ASCII strings)

Malfind | grep -B4 MZ | grep process   
This will output only hits with MZ signature (PE) as well as showing the four lines before any hit (showing the line that includes the process name), and finally grep out any lines that include Process. This compound command is a quick list of processes with a high likelihood of code injection.

Code without the MZ header but with PUSH EBP - MOV EPB, ESP is assembly language prologue that denotes a function. This is still code and should be analyzed.

**Hollowfind**

Attempts to identify process hollowing. **Hollowfind** compares information stored in the process environment block (PEB) with information that should match with the process’s virtual address descriptor (VAD). Similar to malfind, it looks for unusual memory sections.

**Threadmap**

Provides more robust identification of process hollowing techniques and defeats possible memory forensic countermeasures. **Threadmap** uses threads as a primary information source (threads are located in the kernel and are required to run code, so if you want to inject code to execute, you must have a thread pointing to it). Each thread is mapped to its respective VAD entry, and threadmap subjects the VAD entry and code it contains to a series of tests to ID anomalies.

**Step 5 - Rootkit and Hooking Detection**

Rootkits subvert the kernel to hide processes, files, registry entries, and network connections. Hooking legitimate system functions and redirecting their output is the most common means for a rootkit to hide itself (like a detour). A malicious process or driver redirects the logical code flow in order to manipulate user input. There are various ways to accomplish this in Windows:   
  
Hooking the SSDT (System Service Descriptor Table). The kernel uses the SSDT as a lookup table for system functions, where each table entry points to a function code. An SSDT hook patches on or more of these pointers to a location the rootkit controls. SSDT patches are global - the hooks will be valid for any process on the system. On Windows, every SSDT entry will point instructions to either the system kernel (ntoskrnl.exe) or the GUI driver (win32k.sys). We only need to identify the address ranges for those two legitimate system files and filter for any entries that do not point within them.

We can identify SSDT hooks using the **ssdt** plugin.

ssdt | egrep -v ‘(ntoskernl | win32k)’ will grep out ntoskrnl.exe and win32k.sys

Use **moddump** to export suspicious drivers from memory for review.

The IDT (Interrupt Descriptor Table) maintains a table of addresses to functions handling process interrupts and executions. An interrupt happens whenever a process needs access to the kernel. If malware controls the IDT, it can manipulate any kernel-level input. Because interrupts have been deprecated, they are no longer useful for malware to hook. Legacy malware will use this technique. F+ include: ntoskrnl.exe and hal.dll. **Idt** plugin works similar to **ssdt** and can help identify IDT hooks.

IAT (Import Address Table). This is where malware tries to access the import address table - a table specific to each process that lists what Windows API functions it uses from specific DLLs and the address where they can be found. IDT and SSDT hooks are increasingly scanned for and blocked by system protection apps, so attackers like to favor IAT hooks as they are simple. Malware will try to access the IAT to overwrite addresses and redirect functions of the process to use malicious code instead. IAT can be overwritten in user mode (not within kernel) and thus effects are not global. Legit apps can create IAT hooks, but the plugin **apihooks** can help us spot malicious hooks.

Some important parameters for **apihooks** -p (PID); -r (skips kernel mode checks); -Q (scan only critical processes and dlls).

Similar to IAT hooks, inline/trampoline hooks also intercept process function calls. The difference is that inline hooks modify the functions themselves, adding a jump (redirect) to malicious code. These changes can be made to only the copies in memory, never touching the disk. A “Hooking Module : <unknown>” might indicate the DLL that made the “JMP” is jumping into a memory section that is not mapped to disk, indicating an injected memory section.

Legit hooks:

Setupapi.dll | mswsock.dll | sfc\_os.dll | adsldpc.dll | advapi32.dll | secur32.dll | ws2\_32.dll | iphlpapi.dll | ntdll.dll | kernel32.dll | user32.dll | gdi32.dll

IRPs (I/O Request Packets) are how the OS interacts with hardware drivers. Hooking these functions gives malware the ability to manipulate network traffic, disk reads, and keyboard entries. Many legit drivers and processes hooks IRPs, so look for suspicious driver names and paths. **driverirp** plugin works similar to **ssdt** and can help identify IRP hooks.

Direct Kernel Object Manipulation (DKOM)

DKOM allows malware to make changes to kernel objects directly within memory. Changes do not get written to disk, making AV detection difficult. Popular attacks include unlinking objects from standard kernel lists (e.g. unlinking evil.exe from its EPROCESS doubly linked list). The process will continue to run, but standard tools will not display the process (including task manager and volatility **pslist**, but will populate with **psscan**).

**Psxview** is a compound plugin that helps the analyst identify anomalies between plugins by providing a cross-view analysis of different plugins. The cross-view starts by querying the system at a high-level (e.g. using the Windows API) and reading linked lists in memory, then diving deeper by comparing results with low-level data gathering (e.g. the MFT from raw disk or scanning all of the memory for errant process structures). Differing results can be an indicator of a rootkit hiding something at a high level.

* **Pslist** and **psscan**
* **Thrdproc** - review all threads found in memory and collects processes using the thread parent process identifier
* **Pspcid** - the PspCid table is another kernel object that keeps track of process and thread PIDs
* **Csrss -** csrss.exe keeps a handle to each process started after it (so there will be no entries for smss.exe (system process) or csrss.exe)
* **Session** - list of processes belonging to each logon session
* **Deskthrd** - identifies processes via threads attached to each Windows desktop

The **psxview** table has a column for each source and displays “false” for the process if its not found in that source (or “true” if it is). Some legitimate factors might cause a process to generate a false e.g. csrss information is only good for processes started after csrss execution. Session and Deskthrd are no available for smss.exe and anything started before it. Use the -R flag to filter out common false positives. When such legit anomaly is detected, it will mark that item with “Okay”.

Because malware can use drivers to introduce code into the kernel, we need to investigate loaded drivers. **Modules** plugin provides the contents of the linked list identifying currently loaded modules.

Loaded drivers

When analyzing drivers, consider using the **baseline** plugin. **Baseline** compares services, processes, and drivers to a known good memory sample (must be provided), and can help cut down the amount of analysis necessary.

**Modscan** plugin scans memory for any instances of pool tags associated with memory pages containing drivers (a deeper dive than **modules** and can identify old drivers still present in memory slack). **Modules** and **modscan** provide driver name, size, and location.

**Devicetree** can give us a different view as it provides visual information about the layering of drivers. Consider comparing loaded drivers with information gathered on systems with evidence of hooking. The two regularly go hand in hand, and might be the best bet for finding well-hidden kernel modules.

**Moddump** can be used to extract the driver for analysis.

**Step 6 - Dump Suspicious Processes and Drivers**

**Rundown:**

Step 1 - Obtain Image Profile

* KDBGscan and Imageinfo - Identifies which OS we are working with

Step 2 - Review Processes

* Pslist - displays the processes following the doubly linked list
* Psscan - scans memory object for processes (more comprehensive than pslist)
* Malprocfind - automates checking common processes for anomalies
* Baseline - provides a differential output between a baseline known good image and a suspected compromised image

Step 3 - Scrutinize Helper Objects assigned to processes

* Dlllist - Outputs DLLs for each process as well as base offset, full file path, and command line parameters
* Getsid - shows access tokens assigned to each process
* Handles - displays handles for each process
  + Process, Thread, Key (registry keys), Event, File, Mutant, Semaphore, Token, WmiGuid, Port, Thread, Directory, WindowStation, IOCompletion, Timer.
* Filescan and mutant scan - more rigorous searching of file handles and mutants
* Netscan/connscan (for older versions of windows) - can show current and closed (in memory slack) network sockets as well as the associated times and processes

Step 4 - Search for Code Injection

* Ldrmodules - scans all of a process’s memory section for PE headers and compares the results with information PEB list. A “true” in a column means the DLL is present, a false means a DLL was not present in the list.
* Malfind - Searches for usual permissions (Page\_Execute\_ReadWrite) and if the file is unmapped (not backed with a file on disk)
* Hollowfind - compares information stored in the process environment block (PEB) with information that should match with the process’s virtual address descriptor (VAD).
* Threadmap - Provides more robust identification of process hollowing techniques and defeats possible memory forensic countermeasures.

Step 5 - Search for Rootkits

* Ssdt - Finds system functions not mapped to (ntoskrnl.exe) or the GUI driver (win32k.sys).
* IDT - Finds system interrupt and execution functions not mapped to IDT
* Apihooks - Searches for Process Import Address Table overwrites
* Driverirp - Searches for drivers hooking I/O Request Packets
* PSXview - compound plugin that helps the analyst identify anomalies between plugins by providing a cross-view analysis of different plugins. The cross-view starts by querying the system at a high-level (e.g. using the Windows API) and reading linked lists in memory, then diving deeper by comparing results with low-level data gathering (e.g. the MFT from raw disk or scanning all of the memory for errant process structures). Differing results can be an indicator of a rootkit hiding something at a high level.
* Modscan - scans memory for drivers including inside memory slack
* Devicetree - visual representation about the layering of drivers
* Moddump - Used for extracting the driver for analysis

Step 6- Extract Processes, Drivers, and Memory Page

* Dlldump - dump dlls from a process
* Moddump - dump a kernel driver to an executable file sample
* Procdump - dump a process to an executable file sample
* Memdump - dump all addressable memory for a process into one file
* Cmdscan - provide command history typed by a user within a console application
* Consoles - provide what is displayed within the console for apps like cmd and powershell
* Dumpfiles - extract cached files from memory
* Filescan - search for file objects in memory
* Shimcachemem - extract app compat cache artifacts buffered in memory

**Step 6 - Extract Suspicious Processes, Drivers, and Memory Pages**   
We can’t always wrap up our memory analysis with just one tool - we can get very close to understanding how a system was compromised and how the malware works, but to completely understand we must perform static and dynamic malware analysis. Once we narrowed down our analysis, we need to dump the artifacts for further investigation.

* Dlldump - dump dlls from a process
  + By default dumps all DLLs, so limit by specifying a PID. You can further specify a specific dll by using its base offset with the -b flag (pare -P with -B). Use the -r flag for regex (case sensitive).
  + There is no guarantee the DLL will be in memory as the DLL might have been paged out.
* Moddump - dump a kernel driver to an executable file sample
  + Use the -r flag for regex (case sensitive). Use the -b flag for base offset (use modules plugin to find base offset).
* Procdump - dump a process to an executable file sample
  + By default dumps all process within EPROCESS doubly linked list (won’t dump unlinked processes). Use the -o flag to specify the offset of unlinked processes found with psscan.
* Memdump - dump all addressable memory for a process into one file
  + Uses the processes page table to collect all memory pages belonging to the process in a single file. It will output the executable code and any loaded DLLs, memory-mapped files, and global kernel memory pages.
  + Effective for file carving (less fragmentation) as well as string searching for domain names, IP addresses, user-typed data, and passwords.
  + Use the -p flag to specify processes in a comma-separated list.
  + Use sister plugin **vaddump** to save each memory section owned by the process in separate files. Helpful for AV or YARA scanning.
* Cmdscan - provide command history typed by a user within a console application
* Consoles - provide what is displayed within the console for apps like cmd and powershell
  + Crss.exe (XP) and conhost.exe (Win7+) are responsible for drawing and maintaining text consoles like cmd.exe and powershell.exe By default cmd.exe keeps 50 entries within its history buffer.
  + Cmdscan shows user input, consoles shows input and output (both sides of conversation)
  + Consoles only shows input/output buffer information from when the dump happened. Cmdscan can recover current and old remnants of command history.
  + Use the -P flag to specify a process PID
* Dumpfiles - extract cached files from memory
  + Will extract a collection of files such as executables, DLLs, registry hives and .dat databases, and other text files such as PDFs and Office documents.
  + -Q to specify physical offset of File\_Object; -r to specify regex (add -i for case sensitivity); -n to use original filename in output
  + There is no guarantee the file will be in memory - the whole file or parts of the file might be paged out (unavailable regions will be zerod out)
* Filescan - search for file objects in memory
  + Complements dumpfiles. Dumpfiles only identifies File\_Objects in teh VAD tree or in process handles list and might not recover close or maliciously manipulated File\_Objects. NTFS special files like $MFT and $LogFile are not present in VAD tree and won’t be recovered by dumpfiles. Interesting finds in filescan output can be dumped using dumpfiles by using the physical offset (-q flag)
* Shimcachemem - extract app compat cache artifacts buffered in memory
  + Important because shimcache is stored in memory and not written to disk until system reboot.
  + –output=csv to output to a csv; -c strips UNC path prefexes and replaces SYSVOL with C: for analysis conveniences.

**Notes on using strings/grep**A standard practice is to use strings twice: once for Unicode (- e l flag) and once for ASCII strings, then combining the two into a single file for review. Use the -t d flag to get the exact byte offset (location) of the string, that way if you find a hit you can go back in the original data stream and look at other information around the hit. Once you have the strings output, you can also grep the output for more information, or view it manually (use the utility Regex Coach for assistance with grep).

Strings -a -t d filexxxx > strings.txt

Strings -a -t d -e l filexxx >> strings.txt

Sort strings.txt > sorted\_strings.txt

Useful grep options:   
-i ignores case sensitivity

-A *num* print number of lines after pattern match

-B *num* prints number of lines before pattern match

-f *file name* file with list of dirty words to run against

**For analysis information consider using the following plugins:**

**svcscan**  - Scan for Windows Service record structures -v Show service DLL for svchost instances

* # vol.py svcscan -v

**hivelist** - Find and list available registry hives - will give offset for hivedump

* # vol.py hivelist

**hivedump** - Print all keys and subkeys in a hive

* -o Offset of registry hive to dump (virtual offset)
* # vol.py hivedump –o 0xe1a14b60
* Write the output to a .txt file and cat for interesting strings

**printkey** - Output a registry key, subkeys, and values -K “Registry key path”

* # vol.py printkey –K “Microsoft\Windows\CurrentVersion\Run”

**dumpregistry** - Extract all available registry hives

* -o Extract using virtual offset of registry hive --dump-dir Directory to save extracted files
* # vol.py dumpregistry --dump-dir ./output

**userassist** - Find and parse userassist key values

* # vol.py userassist

**hashdump** - Dump user NTLM and Lanman hashes

* # vol.py hashdump autoruns - Map ASEPs to running processes -v Show everything

**cmdscan -** Scan for COMMAND\_HISTORY buffers

* # vol.py cmdscan

**For more information:**

https://github.com/volatilityfoundation/volatility/wiki/Command-Reference

https://www.sans.org/posters/memory-forensics-cheat-sheet/