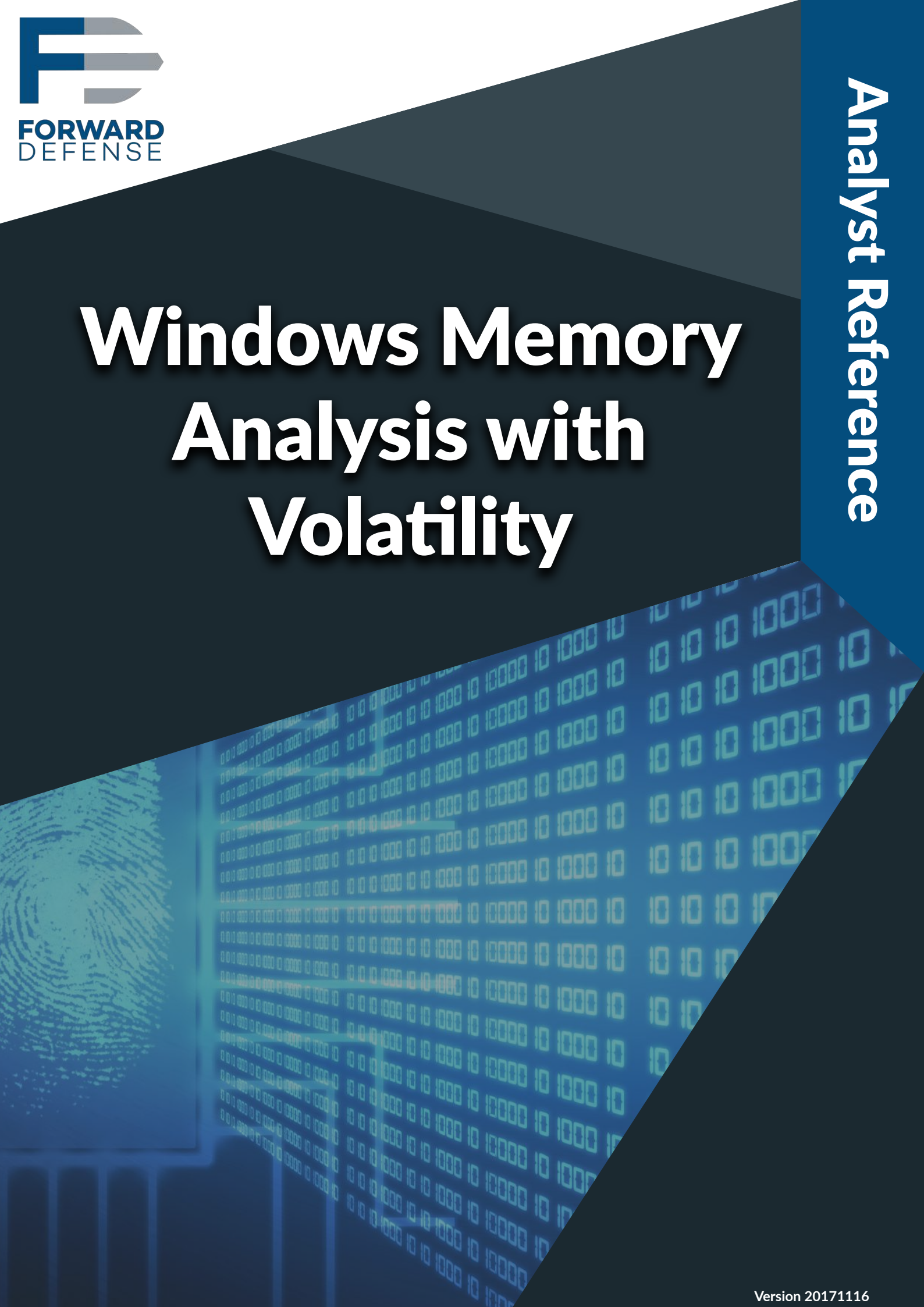
FORWARD DEFENSE

Windows Memory Analysis with Volatility

Analyst Reference



Version 20171116

Table of Contents

Table of Volatility Modules

Before You Start

Memory analysis is most effective when a known-good baseline is established. Where possible, before an incident occurs, collect information on ports in use, processes running, and the location of important executables on important systems to have as a baseline. By comparing results gathered before a compromise to those gathered after an incident, anomalies on the impacted systems will be much easier to detect. Additionally, being familiar with running processes and open ports that are common on Windows systems is helpful for the same reason. Before attempting to examine processes on a system, it is a good idea to familiarize yourself with processes that are typically found on Windows systems, the location of their executable files, the way they are normally initiated (including their parent process), and how many instances of each is normal. To help with that baseline understanding, consult the following article: http://sysforensics.org/2014/01/know-your-windows-processes/.

Installation of Volatility is assumed in these notes. This can be accomplished by using a prebuilt Linux distribution with the tool already installed such as the SANS Investigative Forensic Toolkit (SIFT) found here: https://digital-forensics.sans.org/community/downloads or by following the instructions listed here: https://github.com/volatilityfoundation/volatility/wiki/Installation

These notes are designed to provide a brief introduction to the capabilities of the Volatility Framework and to serve as a quick reference during use. Those looking for a more complete understanding of how to use Volatility are encouraged to read the book The Art of Memory Forensics (http://www.memoryanalysis.net/amf) upon which much of the information in this document is based. More succinct cheat sheets, useful for ongoing quick reference, are also available from here http://downloads.volatilityfoundation.org/releases/2.4/CheatSheet_v2.4.pdf and from here https://digital-forensics.sans.org/media/memory-forensics-cheat-sheet.pdf.

Types of Files That Can Be Analyzed

Volatility can process RAM dumps in a number of different formats. It can also be used to process crash dumps, page files, and hibernation files that may be found on forensic images of storage drives. Finally, RAM files from virtual machine hypervisors can also be processed.

Keep in mind that since the system is running when RAM is captured, the software tool used to capture the data stored in RAM is not able to make a perfect "Image" in the same way that we can with non-volatile storage devices like hard disks. When capturing RAM, we can only copy each page as it exists at that moment. Therefore, when copying RAM, the data on a page may change right before it is copied and/or right after it is copied. Since RAM will have many pages, copying the contents of each may take several minutes depending upon the capacity of RAM storage, the type of removeable media being used (i.e. USB 2.0 vs. USB 3.0 or 3.1), and other activities occurring on the system at the same time. Therefore, if the copying process takes several minutes to complete, pages copied early in the process may contain totally different data by the time the last page is copied. If too many changes occur during the RAM capture, the resulting RAM dump may end up being corrupted to the point that analysis is not possible. To help minimize this risk, do not interact with the system while capturing RAM any more than is necessary. Also remember that most tools only capture the data that is in RAM at the time of capture, so if data has been paged out to disk, that data may not be captured.

Analysis of memory stored on disk, like crash dumps, page files, and hibernation files, is a bit different than data captured from a RAM dump. Page files generally lack the context necessary to completely interpret their data since they represent only a small amount of data relative to what is stored in RAM. Nonetheless, usable data can be recovered from page files, so they are worth examining. While the normal location of the page file is C:\pagefile.sys on Windows systems, additional or alternate locations can be specified by modifying the PagingFiles registry key located at HKLM/CurrentControlSet/System/ Control/ Session Manager/Memory Management. It is therefore best practice to double check that registry setting when analyzing a disk image for paged RAM data.

Contents of RAM are also stored in a hyberfil.sys hibernation file when a Windows system enters hibernation mode to facilitate the system restarting in its previous state. Windows crash dumps store memory contents to disk to facilitate debugging activities, and these files can likewise be used by examiners to recover memory artifacts from the time when the crash occurred. Much like with page files, not all crash dumps will contain sufficient information for meaningful analysis to occur.

Virtual Machine memory can be acquired as if it were a running on bare metal system or in some cases through the hypervisor itself. For example, VMWare can create memory dump files in its own format by taking a snapshot or suspending the virtual machine. Depending on the version of VMWare and how the files are created, RAM data may be found in files with extensions .vmem, .vmss, or .vmsn.

Volatility Command Syntax

Volatility is written in Python, and on Linux is executed using the following syntax:

vol.py -f [name of image file] –profile=[profile] [plugin]

In the above line, the -f option is used to indicate the name and location of the RAM dump file to be analyzed. The –profile= option is used to tell Volatility which memory profile to use when analyzing the dump. The [plugin] represents the location where the plugin to be used is provided. Volatility is a flexible framework that allows multiple types of plugins to be used to extract information from a RAM dump. Each plugin performs a specific task or set of tasks to create a result. Note that for Windows installations using the Volatility executable, the vol.py in the example line above is replaced with the appropriate executable name, such as volatility-2.5.exe -f [image file name] –profile=[profile] [plugin]

If you are not sure what type of Windows system a RAM image came from, you can ask Volatility to give you additional details about the image with the vol.py -f [image file name]imageinfo command. This will give you suggested profiles to use on that image. To further narrow down the most likely profile, the vol.py -f [image file name] kdbgscan command will use the kernel debugger data block scan plugin to make a profile suggestion based on the KDBG header. Since the profile tells volatility the format and type of memory objects that should be present in the RAM dump, getting the profile correct is an important first step before any further analysis.

Note that for the rest of this document, we will simply refer to the location of the image file as [image] and the profile for that image as [profile]. We also will stick to the Linux command syntax, however, the same general format can be used on Windows workstations if that is where you chose to run Volatility. You would simply need to substitute the name of your Volatility executable as noted above, and make substitutions such as findstr for grep where such examples are provided.

Analyzing Network Connections

A common starting point when examining a system for malicious behavior is identifying rogue network connections. The primary Volatility plugin for determining network connections in Windows systems beyond Windows XP is the netscan plugin. It will carve through the memory dump looking for artifacts from network activity, which means it may find sessions that were active or inactive at the time of the RAM dump. The syntax is

vol.py -f [image] –profile=[profile] netscan

Sometimes this plugin is unable to find all the information necessary to reconstruct all the active sessions due to data being paged out at the time of the dump. Additionally, it may recover partially deleted data regarding old connections and/or generate false positive results. As a result, it is a good idea to run commands like netstat -anob at the time of volatile data collection. To have a point of comparison. Keep in mind that tools like netstat may be fooled by malware is running on the live system, so the netscan plugin may detect hidden network activity that netstat misses. Comparing the results of both commands is therefore a best practice when possible.

While memory analysis can provide valuable insight into network activity, and provide some of the best evidence regarding which code on the system was responsible for it, don't discount the need to collect network forensics evidence as well. Consider correlating evidence from memory dumps with network-based evidence such as log files and live packet captures. Tools like Security Onion (https://securityonion.net) provide a great platform for network security monitoring, with tools like Bro, Squil or Suricata often being the initial detection mechanism for suspicious network activity.

Analyzing Processes

A process can be thought of as a container that holds executable program code, imported libraries, allocated memory, execution threads, and other elements necessary for a computer program to function. A process receives its own allocation of memory and enables an instance of a computer program to run on the system. Malicious code can run on a victim system either as its own process or by injecting code into the context of an already running process. Therefore, analysis of processes is an important aspect of memory forensics. For additional information about processes on a Windows system, consult Windows Internals Part 1 (now in its 7th exceptional edition) by Mark Russinovich, *et. al.* (https://docs.microsoft.com/en-us/sysinternals/learn/windows-internals).

Listing Processes

On Windows systems, the kernel tracks the currently active processes using a doubly-linked list. Each running process is found in this list, and therefore most standard Windows calls to list processes accomplish this by walking this list and printing each process found in it. Some malware will attempt to hide by delinking its process from this list. In those instances, most live tools run on the system will fail to detect the malware process. When working with a memory dump, different approaches can be taken to locate processes. For example, each process has a fixed format header that contains a key or tag of "Proc" on Windows systems. By searching through all the memory in a RAM dump for the known structure of a process object's header and other attributes, Volatility can detect processes that are not linked in the standard doubly-linked process list. By using and comparing different methods of identifying processes, an examiner can identify processes that were attempting to hide their presence.

One of the easiest ways to get a list of processes that were running at the time a RAM dump was made is:

vol.py -f [image] –profile=[profile] pslist

The pslist plugin walks the doubly-linked list of processes in the same way as most commands that run on the live system. It therefore provides a useful baseline of what would have been seen by commands like pslist or tasklist when the system was running, but will not give any information about processes that were hidden by removing themselves from the process list or those that had already terminated before the dump was captured.

The pstree plugin will place the list of processes in a tree format to show which processes spawned other processes and make their parent/child relationship clearer. However, it also relies on walking the doubly-linked process list, and therefore suffers from the same limitations as the pslist plugin. It can, however, be a useful command to run, particularly to understand the relationship between processes. For example, if a particular process is identified as malicious, understanding what other processes it spawned helps identify other processes that may be acting maliciously (fruit of the poisonous tree, if you will). Also, a process that spawns in an abnormal way (such as explorer.exe being used to launch a svchost process) may also signal anomalies caused by malicious activity. The syntax to run the pstree module is simply:

vol.py -f [image] –profile=[profile] pstree

As mentioned earlier, Volatility is not limited to using only the doubly-linked process list to identify processes. The entire memory dump can be scanned for known signatures of process objects, and anything that matches that pattern can be displayed. This is an extremely helpful method to find processes that have delinked from the process list to avoid detection. Since it does not rely on the doubly-linked process list, it can also uncover information about processes that were once running but that terminated before the dump was captured. A process scan can be run with the syntax:

vol.py -f [image] –profile=[profile] psscan

The output from the psscan plugin does not provide the hierarchical view of the parent/child relationship in the way that the pstree plugin does. To get a similar effect, you can output the results of psscan into a dot file, and use a program like graphviz to display it graphically. This can be both an informative investigative approach and also makes illustrative graphs for report purposes. To accomplish this, a command like the following can be used:

vol.py -f [image] –profile=[profile] psscan --output=dot --output-file=processes.dot

This command will create the list of process in the dot format. To then convert that to a format such as JPEG, the dot command can be used as follows:

dot -Tjpg processes.dot > processes.jpg

There are many structures within a Windows system that need to track running processes. While the doubly-linked process list is the most commonly used method for enumerating running processes, it is also the most likely to be targeted by processes that are attempting to evade detection. As a result, comparing the results of the doubly-linked list to other structures within the operating system and other methods of detecting processes can help identify processes that are maliciously hiding their presence. For such comparative analysis, the command vol.py -f [image] –profile=[profile] psxview uses multiple methods for detecting processes and lists which processes are and are not detected by each method. This comparison can help identify processes that are maliciously trying to avoid detection. Some methods will not detect certain processes, such as those that were started before the object upon which the detection method relies. Similarly processes that have terminated will not be detected by methods that only track running processes. To help account for these expected variations, the command

vol.py -f [image] –profile=[profile] –apply-rules psxview

Will show True when a method detects the process, False when the method does not detect the process, and Okay when the process is expectedly absent due to a known limitation of the method being used. Keep in mind that only the psscan method will detect terminated processes.

Examining A Specific Process

Obtaining a list of processes from a memory dump file can be an important way to identify suspicious activity on a system. However, once a process has been identified as potentially malicious, additional steps are often needed to confirm those suspicions and to determine the nature of the process itself. A number of different methods can be used to learn more about a particular process.

For a process to access other elements of the system it must first acquire a handle to the objects that it wants to manipulate. Whether reading a file, writing to a registry key, or opening a connection to a remote share, the process must have permission to access the object and secure a handle to that object. Permissions are determined based on the user or group that is attempting to perform an action, and the permissions that have been assigned to that user and/or the groups of which it is a member. A process is assigned a security token based on the user or service account context from which it was run. This token lists the user and/or groups for which the process is working, which in turn determines which files it may access and other security permissions. The operating system uniquely refers to each user or group with a numeric Security Identifier (SID). To determine the SIDs that are associated with a process' token, use the following command:

vol.py -f [image] –profile=[profile] getsids -p [PID]

Where PID is the process identifier of the process that you wish to examine.

In addition to permissions, a process may also be assigned privileges by the operating system to perform certain tasks. Privileges include things like the ability to bypass file permissions in order to read files to make backup copies, the ability to access memory of any process to perform debugging operations, the ability to shutdown or restart the system, or the ability to load kernel drivers. These privileges are determined in accordance with local computer policies set by the system administrator. Malware will frequently attempt to enable privileges to allow a malicious process to perform additional tasks. To list the privileges assigned or enabled for a particular process use the following command:

vol.py -f [image] –profile=[profile] privs -p [PID]

Where PID is once again the process identifier of the process that you wish to examine. The output of this command will list the various privileges that are present (allowed) for that process, an indicator of whether each privilege is enabled, a note as to whether the system enabled the privilege by default, and a description of what the privilege allows the process to do. Before a privilege may be used, it must first be enabled. Therefore, your analysis should pay particular attention to enabled privileges, particularly those that were not enabled by default as they indicate a privilege that the malware bothered to specifically enable and has therefore likely used or intended to use. The –silent option can be added to only show those privileges that were enabled.

In addition to understanding the permission and privilege context of a process, it is important to understand which handles it has opened to other system resources. A handle is a mechanism used by the operating system to allow access from one resource to another, and to ensure that different resources are not attempting to make conflicting changes at the same time. Specifically, a handle controls access to kernel objects that represent other resources on the system like files, registry keys, processes, etc. To list the handles opened by a process use the command:

vol.py -f [image] –profile=[profile] handles -p [PID]

A process may have many handles opened, so the -t option can be used to restrict the output to a certain type of handle. Examples include key, file and thread. To list only the handles to registry keys, use the command:

vol.py -f [image] –profile=[profile] handles -p [PID] -t key

File objects can obviously represent files stored on disk, but can also be used to represent network connections. The type of device involved should be apparent when looking at the path to the object. Some items that may be less obvious include:

- \Device\Ip \Device\Tcp and \Device\Afd\Endpoint all refer to handles for network connections.
- \Device\LanmanRedirector and \Device\Mup both refer to handles to SMB network shares.

Searching for these devices may help you locate indications of network activity by the process being examined. Alternatively, the following command can be used to identify drive letter assignments, such as the C or D drives being assigned to hard drive or optical drives, but also assignment of drive letters to mapped network drives, along with the time when the mapping was created.

vol.py -f [image] –profile=[profile] symlinkscan

If you know that a malicious process is storing data in a certain file, you can search through all the process file handles to determine which process is using that file. For example, if the file name was hiddenfile.txt, you can use the following command to identify processes that may be using that file:

vol.py -f [image] –profile=[profile] handles -t file | grep hiddenfile.txt

In addition to handles, it may be of use to examine the environment variables set by a process. The basic syntax of the envars plugin is:
vol.py -f [image] –profile=[profile] envars

This will list all environment variables for all processes that were running at the time of the dump. However, the plugin can be restricted to a single process with the -p [PID] switch as seen previously with plugins like handles, privs, etc. Finally, the –silent option can be employed to have volatility compare the results of the envars plugin and compare it to a list of known, normal values and then only display items that do not match the known values as programmed into the module.

When analyzing a process, it is important to know which DLLs (Dynamic Linked Libraries) are imported into the process itself. A DLL contains executable code that can provide a process with specific functionality, so understanding which DLLs a process incorporates may give insight into its capabilities. In addition, malicious software may inject rogue DLLs into otherwise benign processes to introduce malicious activity without standing up a new process on the system, so examining processes for the presence of malicious DLLs or other code injection is an important analysis step. Volatility supports this type of analysis with a few different plugins.

Within a process' memory space is the Process Environment Block or PEB. The PEB contains a number of different items of interest including but not limited to:

- The path to the process' executable on disk;
- The command line used to invoke the process;
- Three different lists of DLLs associated with the process:
    - One that lists the order in which each DLL was loaded into the process;
    - One that lists the DLLs based on their order in process memory;
    - One that lists the order in which they are executed by the program code.
- The standard input, output, and error for the process;
- The process' working directory.

Most tools that run on a live system determine the DLLs used by a process by consulting the first of the three DLL lists stored in the PEB, which tracks the order in which each DLL is loaded. As a result, malware will sometimes modify that list to hide the presence of a DLL. Volatility has a plugin that also parses this same list, which can be run with the following command:

vol.py -f [image] –profile=[profile] dlllist -p [PID]

This plugin will list any executable code module, including the program itself. The program executable will load first and should therefore be the first item in the results of this plugin. As a side benefit, the original command line and any arguments used to originally launch the process is also pulled from the PEB and displayed by this module. Ntdll.dll and kernel32.dll are frequently found DLLs that load early in the invocation of many processes. After that, the Import Address Table (IAT) of the process is used to begin loading other DLLS as specified for the process. The dlllist plugin will report a load count of -1 (0xFFFF) for items loaded from the IAT. Other counts will be present for other methods of loading DLLs into the process' memory space.

To help detect DLLs that have unlinked for the load order list in the PEB, Volatility also has a ldrmodules plugin. This plugin acts similarly to the psxview plugin for processes in that it will enumerate the results of DLLs listed in all three lists in the PEB and present a comparison of the results. This helps an analyst to detect anomalies that may be indicative of an attempt to hide the presence of a DLL. In addition, the ldrmodules plugin also manually scans the process' executive object in kernel memory looking for signatures of DLLs or other types of executable code modules and presents a list of all items that it detects. In this way, even if the process memory itself has been tampered with, the lists of modules stored about the process in kernel memory can be used to help identify any tampering. One thing to be aware of in the output from this plugin is that the executable itself will by default only appear in two out of the three PEB lists since it is not a separately loaded DLL but is rather the main executable code. The ldrmodules plugin can be run with the following syntax

vol.py -f [image] –profile=[profile] ldrmodules -p [PID]

Another plugin that may come in handy in detecting malicious code that has been injected into a process is malfind. This plugin looks for suspicious memory areas within a process and displays them along with their associated assembly code so that an analyst can determine if it is suspicious.

Examining Running Services

Services are processes that run automatically, typically do not require user interaction, and run in predefined user context rather than the context of a user that manually launches them. The Services Control Manager (SCM) is responsible for starting and managing services. The SCM is implemented by services.exe. Services are typically defined in the registry HKLM\SYSTEM\CurrentControlSet\services key. Each configured service will normally have a subkey and details of any DLLs needed by that services can be found in its respective subkey.

Volatility uses a scanning technique to detect services in memory dumps, even those that use unusual loading methods or actively try to avoid detection. Again, having results of tasklist /SVC as comparison is a good practice, so running such commands at collection time is a good idea.

To scan for services, use the svcscan plugin, with the following syntax:

vol.py -f [image] –profile=[profile] svcscan

You can optionally include the –verbose option for additional details.

Dumping Process Memory for Further Analysis

Volatility can dump the contents of a process' or DLL's memory space for further analysis with other reverse engineering tools if desired. Keep in mind that variables that exist within the program are instantiated with actual values when a process is running. Additionally, some parts of the process memory space may be paged to disk or otherwise inaccessible at the time of the memory dump. Finally, code that is packed or encoded on disk may be unpacked in memory. As a result, the information dumped from a memory dump is not going to be identical to the information stored in the associated executable on disk. For this reason, traditional hash matching techniques and many signature based detection mechanisms will not work when run against process memory extracted by Volatility. Nonetheless, antivirus tools may be able to offer insight into the nature of a process recovered from a RAM dump and reverse engineering tools may be able to determine the actions taken by the process.

If desired, the procdump plugin can be used to dump contents of process memory.

vol.py -f [image] –profile=[profile] procdump -p [PID] –dump-dir=[directory/]

The above procdump will dump the entire contents of the process memory to a file in the directory specified by –dump-dir= option. With addition of the –memory switch, any memory that is not able to be dumped due to paging or other reasons is filled in with zeros to keep the relative location of other objects consistent with the original process. Similar to the procdump plugin, the dlldump and moddump plugins can be used to dump specific DLLs from within a process' memory space.

Detecting Kernel Loaded DLLs

If determining which modules the kernel has loaded is of interest in your analysis, the modules and modscan plugins can be used. The following command walks the doubly-linked list of loaded kernel drivers found in the LDR_DATA_TABLE_ENTRY structures and provides the name and path of DLLs loaded by the kernel. The syntax is as follows:

vol.py -f [image] –profile=[profile] modules

If a DLL has been removed from that list, the modules plugin will not find it. However, the modscan plugin will scan the memory dump for the tags or signatures of kernel loaded DLLs and provide a list based on its scan. Because it relies on pattern matching and interpretation of memory data, it may result in false positive results. The syntax is

vol.py -f [image] –profile=[profile] modscan

Registry Artifacts in Memory

Since many elements of the Windows registry are updated or read frequently by the OS, it is very common to capture registry key data in a RAM dump. Volatility has a hivelist plugin to list registry hives, including their path on disk. There may also be a hive listed by Volatility as "[no name]" that represents pointers to other hives and is normal. The syntax for this command is

vol.py -f [image] –profile=[profile] hivelist

Malware will often use autostart locations, places in the registry or elsewhere that causes executable code to be launched automatically as a system starts or a user logs in. Since many of these locations are in the registry it may benefit your analysis to look at specific keys for evidence of malware. The printkey plugin provides the ability to view the text data stored within a registry key. The syntax for this plugin is

vol.py -f [image] –profile=[profile] printkey -K "Path\To\Key"

where "Path\To\Key" represents that path to the specific key that you desire to examine.

For example, to determine the current control set being used by the system, the current control set key can be examined with the command

vol.py -f [image] –profile=[profile] printkey -K currentcontrolset

The registry also stores multiple lists of recently used programs, including the userassist key that tracks recently run programs and the time of their execution. While the exact location of the userassist key varies depending on the version of Windows involved, the userassist plugin is able to locate and parse the data in the userassist key into readable text. The syntax is

vol.py -f [image] –profile=[profile] printkey -K "Path\To\Key" userassist

Similarly, information about executables that were previously present on the system can be gleaned from the shimcache and shellbags keys of the registry. The shimcache and shellbags plugins respectively will parse and present this information.

If needed, password hashes can be dumped from memory for external password cracking. Volatility is able to obtain the system key from the SYSTEM hive and use it to extract the hashes from the SAM hive. The syntax of the command is

vol.py -f [image] –profile=[profile] hashdump

Additional user password data may be recoverable from the LSA Secrets stored in the registry. Again, Volatility automates that extraction with the lsadump plugin, with the following syntax:

vol.py -f [image] –profile=[profile] lsadump

Special Use Plugins

If you notice that a suspect had the notepad.exe application open, there is a volatility plugin that can recover typed text from the notepad.exe process memory. Use the following syntax,

vol.py -f [image] –profile=[profile] notepad

To search for commands previously entered into a command shell, try running the *cmdscan* plugin as follows:

vol.py -f [image] –profile=[profile] cmdscan

Additionally, the consoles plugin uses an alternate method to perform a similar search:

vol.py -f [image] –profile=[profile] consoles