APPLIED
INCIDENT
RESPONSE

Analyst Reference

Windows Memory Analysis with Volatility

Version 20200131

Contents

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. We have a Default Windows Process Quick Reference available to help you get started.

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 or by following the instructions listed here.

This document provides a brief introduction to the capabilities of the Volatility Framework and can be used as reference during memory analysis. Those looking for a more complete understanding of how to use Volatility are encouraged to read the book The Art of Memory Forensics 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 and from here.

Volatility 2 vs Volatility 3

Most of this document focuses on Volatility 2. As of the date of this writing, Volatility 3 is in its first public beta release. Volatility 2 is based on Python 2, which is being deprecated. Volatility 3 is a complete rewrite of the framework in Python 3 and will serve as the replacement moving forward. That said, it is not yet fully developed, so Volatility 2 will be kept updated until August 2021. The first full release of Volatility 3 is scheduled for August 2020, but until that time Volatility 3 is still a work in progress and does not yet contain all the features available in Volatility 2. If you wish to experiment with Volatility 3, setup instructions are here, and we provide some notes on usage at the end of this document. Keep in mind that Volatility 3 no longer requires profiles, instead using symbol tables, similar to the approach used by Rekall. So, make sure you download the required symbol tables during installation.

Types of files that can be analyzed

Volatility can process RAM dumps in different formats. It can also be used to process crash dumps, page files, and hibernation files that may be found on traditional forensic images of storage drives. Finally, memory files from virtual machine hypervisors (*e.g.* VMware .vmem files) 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. The data on a page may change right before it is copied and/or right after it is copied. Since RAM has many pages, copying the contents of all of them may take several minutes depending upon the capacity of RAM storage, the type of removeable media being used (*i.e.* USB 3.0/3.1

Generation 1, vs. USB 3.1 Generation 2), 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 will usually not be captured. Tools such as the Pmem suite can also capture the data paged to disk, but your analysis tool may not be able to fully process the paged data. When capturing RAM with any tool, also consider interrogating the system at the command line or capturing data about processes, connections, etc. using an agent such as Velociraptor or an Endpoint Detection and Response (EDR) tool immediately after RAM capture to ensure that you have actionable data collected in case the RAM dump is not able to be parsed correctly by memory forensics tools.

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 lack the context necessary to completely interpret their data since they only contain fragments that were previously stored in RAM. Nonetheless, usable data may 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 value located in the HKLM/SYSTEM/CurrentControlSet/Control/Session Manager/Memory Management key. It is therefore best practice to double check that registry setting when analyzing a disk image for paged RAM data.

You may alo find data that was previously in RAM in crash dumps created by the operating system when problems are encountered. If configured to create crash dumps, your system may place them in the %SystemRoot% folder with the name Memory.dmp. For additional details on configuration options, see this article. Crash dumps are not raw memory captures but have headers containing metadata about the dump. These dumps are designed to be analyzed with the Windows Debugger, WinDbg, but if they are a full memory dump, memory forensics tools may be able to parse these files to provide information about the state of the system at the time the crash occurred.

Windows hibernation files are another potential source of memory data. When a computer goes into hibernation mode, the contents of RAM are compressed and copied to disk in a file called hiberfil.sys in the root of the system drive. This also occurs in support of the Windows fast startup mode, so hiberfil.sys may be found on desktop or laptop computers. The *imagecopy* plugin can convert a hibernation file to a raw memory dump for analysis. Note that the act of hibernating will have an impact on network connections that may have been active when the system went into the reduced power mode, and therefore information about active network connections may be altered.

Virtual Machine (VM) memory can be acquired as if the VM were a running 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. Additional processing with tools like vmss2core may be needed to extract a usable memory dump. You should consult your hypervisor vendor for the proper files and process to extract a memory dump from a snapshot or suspended virtual machine.

Basic Volatility 2 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-2.6.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 (*kdbgscan*) 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 to any further analysis. Note that with Volatility 3, this process is replaced with an autodetection process referred to as Automagic.

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.

In Windows, processes are represented by an executive object in the kernel of the operating system called an _EPROCESS. Volatility uses a variety of methods to find and examine these objects. Inside the _EPROCESS structure is the Process Environment Block (PEB) that contains lists of the process' loaded DLLs, the command line used to launch the process, the path to its associated executable, and pointers to its assigned memory regions. The memory assigned to each process can also hold valuable information, and Volatility can be used to examine each process' allocated memory.

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, causing most tools on a live system to fail to detect the unlinked 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 the memory in a RAM dump looking for the known structure of a process object's tag 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 [dump_file] --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 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 display a list of processes in a tree format to show which process 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:

vol.py -f [dump_file] --profile=[profile] pstree

As mentioned earlier, Volatility is not constrained to only using the doubly linked process list to identify allocated processes. The 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 been 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 running previously but terminated before the dump was captured. A process scan can be run with the syntax:

vol.py -f [dump_file] --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 a useful investigative approach and make for useful graphs for report purposes. To accomplish this, a command like the following can be used:

vol.py -f [dump_file] --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 detect processes that are maliciously hiding their presence. For such cross-comparative analysis, the command *vol.py -f [dump_file] --profile=[profile] psxview* uses multiple methods for detecting processes and displays which processes are and are not detected with

each method. This comparison can help detect 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, or processes that have terminated not being detected by methods that only track running processes. To help account for these expected variations, the command *vol.py -f [dump_file] --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 Process with Volatility

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 must be taken to confirm those suspicions and to determine the nature of the process itself. Several 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 account 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 context in 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 [dump_file] --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 additional privileges to allow a malicious process to perform additional tasks. To list the privileges assigned or enabled for a process use the following command:

vol.py -f [dump_file] --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 for that process, an indicator of whether each privilege is enabled, a note as to whether the system enabled the privilege by default or if it was explicitly enabled, 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 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 likely used or intended to use. The --silent option can be added to show only those privileges that were explicitly 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 [dump_file] --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 specified type of handle. Examples include *key*, *file* and *thread*. To list only the handles to registry keys, use the command:

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

File objects can obviously represent files stored on disk, but they 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.

Therefore, searching for these device handles 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, assigned to hard drives or even mapped network drives, along with the time when the mapping was created

vol.py -f [dump_file] --profile=[profile] symlinkscan

If you know that an adversary is storing data in a certain file, you can search through all the process file handles to determine which process was 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 [dump_file] --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 [dump_file] --profile=[profile] envars

This will list all environment variables for all processes that were running at the time of the dump. The plugin can be restricted to a single process with the *-p [PID]* switch as seen previously with handles and other plugins. Finally, the *--silent* option can be employed to have Volatility compare the results of the *envars* plugin to a list of known, normal values, and 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-link 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 starting 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 several 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 [dump_file] --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 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. Different counts will be present for other methods of loading DLLs into the process' memory space.

To help detect DLLs that have unlinked from 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 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 [dump_file] --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*. As attackers seek to evade endpoint protection systems, they will often inject malicious code directly into the process space of an otherwise benign process. This allows them to keep their malicious code from being written to disk where it is more likely to be scanned by antivirus or other endpoint defenses. The *malfind* plug-in is designed to help you detect such injected code.

Memory is allocated in units known as pages. Although pages may vary in size from system to system, 4,096 bytes is a common value. The concept of a page is like a cluster on disk, in that a page is the smallest unit that can be allocated in memory and a cluster is typically the smallest unit on disk that can be allocated by the operating system. Each page must be provided with permissions indicating whether the data contained within it can be read, executed, or written. DLLs are typically loaded with permissions indicating that they can be read, but if they are written to, a new copy must be made and the changes made only on that copy (copy on write). This allows multiple processes to share a single instance of a DLL in memory, but if one of the processes attempts to make a change to that DLL it must copy its own instance of the DLL into its process memory space before it is allowed to make changes. This avoids one process modifying code that may be in use by other processes in the case of a shared DLL.

For malicious code to be injected into the memory space of a running process, the page holding that memory must allow new code to be written to that page. For the code to then be of any use to the attacker, the code must be able to be read and executed as well. Normally, if a page of memory contains executable code, that code will have been loaded into memory from disk, so the code in the page is backed by a file on disk, and the location from which is was loaded is recorded in RAM. When a page is marked with read, write, and execute permissions, but there is no associated file on disk to explain from where that code came, that is indicative of code having been injected into the process maliciously. The *malfind* plug-in automates the detection of pages that are marked with read, write, and execute permissions and are also not backed by a file on disk.

Although the plug-in helps identify potentially suspicious pages within a process's memory, it is up to you to complete the analysis and confirm that the pages discovered contain executable code. One of the easiest ways to identify executable code is by the presence of the MZ header at the beginning of the page. This header is used by Windows systems to identify executable files. Even if the MZ header is not present, the page may still contain executable code, so the *malfind* plug-in will display the hexadecimal and ASCII representations of the data as well as display the assembly language instructions that data would represent if it was intended as executable code. It is up to the human analyst to decide whether the data contained in the page is executable code or simply other types of data that would not be harmful to the system. Note that the *malfind* plugin only displays the first 64 bytes of each memory address it identifies. Malware authors may avoid putting an MZ header or obvious code at the beginning of the memory segment to avoid detection, so it may be necessary to dump the memory for further examination. This can be done by adding the *--dump-dir=[directory]* option to the *malfind* command to dump each memory segment that it finds out to disk for further analysis, as discussed in the next section.

You can also try the *hollowfind* plugin to help identify malicious code that is using process hollowing techniques. The following syntax searches for processes that exhibit signs of process hollowing and

saves the associated memory segments to a directory called hollowfind_dir (which must already be created):

vol.py -f [dump_file] --profile=[profile] hollowfind -D ./hollowfind_dir

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, including antivirus software, 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. While many attackers obfuscate code when it is stored on disk, when it is in RAM we are able to capture more of it in a decoded state, which may help in its identification. The *procdump* plugin can be used to dump contents of process memory.

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

The above command will dump the entire contents of the process memory to a file in the directory specified by *--dump-dir=* option. With athe ddition 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. Like the *procdump* plugin, the *dlldump* plugin can be used to dump specific DLLs from within a process' memory space and the *moddump* plugin can dump driver memory.

Detecting kernel loaded drivers

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 drivers loaded by the kernel. The syntax is as follows:

vol.py -f [dump_file] --profile=[profile] modules

If a driver 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 drivers and provide a list based on its manual scan. However, because it relies on manual scanning and interpretation of memory data, it may result in false positive results. The syntax is:

vol.py -f [dump_file] --profile=[profile] modscan

Registry artifacts in memory

Since many elements of the Windows registry are updated or frequently read by the OS, it is 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 [dump_file] --profile=[profile] hivelist

Malware will often use autostart extensibility points (ASEPs), places in the registry or elsewhere that cause executable code to be launched automatically as a system starts, a user logs in, or other defined event. 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 subkeys, value names and data stored within a registry key. The syntax for this plugin is:

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

where *"Path\To\Key"* represents that name (and optionally portions of the path) to the specific key that you desire to examine. If the key name specified exists in multiple places, each instance will be printed.

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 [dump_file] --profile=[profile] printkey -K currentcontrolset

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 from the SAM hive can be dumped from memory for external password cracking. Volatility can 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 [dump_file] --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 [dump_file] --profile=[profile] lsadump

Network connections

One of the more important aspects of examining a system for malicious behavior is identifying rogue network connections. While memory analysis can provide great insight into the type of activity occurring, and provide 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 provide a great platform for network security monitoring of suspicious activity, with tools like Zeek and Suricata often being the initial detection mechanism for suspicious network activity.

The primary plugin for determining network connections in Windows systems is the *netscan* plugin. It will carve through the memory dump looking for artifacts from network activity, which means it may find both sessions that were active or inactive at the time of the RAM dump. The syntax is

vol.py -f [dump_file] --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 that 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.

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 key HKLM\SYSTEM\CurrentControlSet\Services. Each configured service will normally have a subkey, and details of the executable or DLLs needed by that service 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 commands like 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 [dump_file] --profile=[profile] svcscan

You can optionally include the *--verbose* option for additional details.

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 [dump_file] --profile=[profile] notepad

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

vol.py -f [dump_file] --profile=[profile] cmdscan

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

vol.py -f [dump_file] --profile=[profile] consoles

The *timeliner* plugin can be used to extract time-stamped objects from the memory image, including process creation times, thread creation times, compile times, socket creation times, and registry timestamps. Output to a body file for inclusion in a larger timeline can be achieved as follows:

vol.py -f [dump_file] --profile=[profile] timeliner --output=body --output-file timeliner_out.body

Notes on Volatility 3

Volatility 3 is a complete rewrite of the framework, building on lessons learned and implementing many previous feature requests. When the first full release happens in August of 2020, the team says it will have all the features of Volatility 2, plus new ones. One of the most noticeable new features from a user perspective is the removal of *--profile* from the required syntax. Much as Rekall does, Volatility 3 will now scan an image to determine its version automatically (or Automagically, to stick to the Volatility vocabulary), query a local set of symbol tables (or download and process the required information to create a new symbol table if it isn't present) and then analyze the image accordingly. Therefore, this process may take a minute and generate a lot of output the first time you run a plugin on a new sample. After the first run, the required symbol table is cached, so future runs take less time. What this change means in practice is that if you have a memory dump that Volatility 2 and/or Rekall cannot process, particularly if it's a newer version of the target OS, giving Volatility 3 a try makes sense.

The names of the plugins have also been redesigned, and not all the plugins from Volatility 2 are present or function in the same way in Volatility 3. Some of the differences are because Volatility 3 is still in beta, but some is the result of new design decisions. Plugins are now broken up into modules that are organized by the operating system to which they apply (Linux, macOS or Windows). A full list can be found here, but examples include:

- *linux.pslist.PsList*
- *mac.netstat.Netstat*
- *windows.dlllist.DllList*

Note that the plugin's class uses a case sensitive, camel case, naming convention (*PsList* in the first example). The module name (*linux.pslist* in the first example) is lower case. Officially, to run a plugin you should specify the full name of the module and its class (*linux.pslist.PsList*) when you indicate the desired plugin at the command line. However, at least in the first public beta, testing shows that just the module name (*linux.pslist*) seems to work as well. There does not appear to be any tab autocompletion, wildcarding, or other convenience feature to help with typing the plugin names in the first public beta. *vol.py -h* does produce a useful help file with a list of the plugins and a one-line summary of the function of each. The new syntax to run Volatility 3 is therefore:

vol.py -f [dump_file] [plugin]

There are some other changes in syntax to watch out for. *-p* is no longer used to specify a process ID, instead *--pid* is used. As in this example:

vol.py -f [dump_file] windows.dlllist --pid PID

Note that the full name of the plugin is *windows.dlllist.DllList*, but the shorter name of just the module (*windows.dlllist*) seems to produce the same result. Also note that in Volatility 3, the *DllList* plugin does not display the command line from the PEB like the Volatility 2 *dlllist* plugin does. Instead, a separate Volatility 3 plugin (*windows.cmdline.CmdLine*) provides that capability. Other Volatility 3 plugins such as *windows.pslist.PsList*, *windows.psscan.PsScan*, and *windows.pstree.PsTree* behave similarly to their Volatility 2 counterparts but with some differences in output, such as column order and spacing. Many of the core Volatility 2 plugins are missing from the Volatility 3 first public beta, but they will presumably be added (or plugins with similar capabilities added) before the August 2020 first full release.

Volatility 3 plugins produce a TreeGrid object as output, and output rendering is controlled with the *-r* or *--renderer* switch. The default is to display tab separated values to standard out. Other options currently include CSV, JSON and others. To declare a different output renderer, include the *-r* (or *--renderer*) option before the plugin name. To get CSV output and redirect it to a file called pslist_out.csv, the following syntax would work:

vol.py -f [dump_file] -r csv windows.pslist > pslist_out.csv

A "dot" renderer does not yet seem to be supported in the first public beta.

For a demo of the Volatility 3 first public beta, check out Richard Davis' 13Cubed Shorts video found here.

https://www.linkedin.com/company/threathunting
https://www.twitter.com/threathunting_