*Blue Highlight (Important Info)*

*Green Highlight (Links that we might need to refer in future)*

*Pink Highlight (Tools)*

(Gathered From : <https://www.sans.org/reading-room/whitepapers/forensics/techniques-tools-recovering-analyzing-data-volatile-memory-33049>)

There are several different types of processes that may be found in volatile memory. All currently running processes are stored there and may be recovered from the data structures that house them. In addition, hidden processes can be parsed out of memory. Finally, processes that have been terminated may still be residing in memory because the machine has not been rebooted since they were terminated and the space they reside in has not yet been reallocated. These too may be parsed out and analyzed.

If the process is a piece of malware, the open files might lead an investigator to discover where the malware is stored on the disk, where it is writing its output, or what previously clean files the malware may have modified to serve its own purposes. Following these leads can help to turn up other critical information such as what type of output the malware is producing and how it is storing it, or what Windows API calls the malware is using (which can give a better idea of how the malware is working).

Passwords and cryptographic keys are as a general rule never stored on hard disks without some type of protection. When they are used, however, they must be stored in volatile memory and once this occurs they will remain in memory until they are overwritten by other data or the machine is rebooted. When forensic examiners capture volatile memory they can parse through it looking for passwords and keys that may help recover critical data that is password protected or encrypted.

Where to find volatile memory ?

In Windows, there are two common device objects that can be accessed to obtain physical memory: \\.\PhysicalMemory and \\.\DebugMemory. A raw image is usually taken of each of these devices; once that raw image is obtained, the analyst can convert it to Microsoft’s crashdump format and look at it using a debugging tool. The analyst can also use many other tools (including some of those described in section 6 of this paper) to parse and examine the contents of the data dump.

In Unix, the physical memory devices are usually /dev/mem/ and /proc/kcore. Not all filesystems use /proc/kcore so the analyst must know the filesystem in order to understand whether this device exists and should be captured for analysis. The common Unix debugger gdb can be used to analyze the resulting raw memory images, and so can many other freely available and commercial tools (once again, see section 6 for more examples)

How volatile memory works?

First, on both Linux and Windows, anything that the kernel, or central operating system, uses and needs in order to run will be represented as an object. For the purposes of this paper, an object can be understood as consisting of data and methods of manipulating that data. As an example, a process running in memory would be an object, and so would a file.

On Windows, every object used by the kernel has an OBJECT\_HEADER, which is a structure that has information about the object stored inside it. When objects are stored in memory on Windows, there are two ways that the kernel can choose to store them. The kernel has two sets of memory that are structured like heaps – these heap structures are usually called pools. There is a paged pool, which is where most data will be stored, and a non-paged pool, where only important objects that the kernel needs to access frequently are stored. Any data in the paged pool can be placed into a file on the hard disk if the machine is running low on actual physical memory (Carrier, 2005). Process and thread objects, because they are so important and accessed so often, are stored in the non-paged pool, which means that all processes that are running at the time physical memory is captured will be available to the analyst (Schuster, 2006). Processes are stored in Windows in a Virtual Address Descriptor (VAD) tree. This tree describes memory ranges used by currently-running processes, and allows a process’s virtual address space to be reconstructed. This is useful for many reasons – most information associated with a process can be found by walking the VAD tree. In particular, as we will see in section 5.8, it is possible to recover all of the memory-mapped files associated with specific processes using the VAD tree. A visual representation of a VAD tree is shown in the section on VAD tools, section 6.8 (van Baar, Alink, & van Ballegooij, 2008).

In Linux, a computer’s volatile memory is seen by the operating system as one object, and it is stored in a single data structure called pg\_data\_t. This data structure stores information on the size of memory, an address table with page descriptors for memory, and much other additional information. Most data that the CPU uses is stored in physical memory as a page frame, which is 4 KB in size by default. This means that if a file that is 12 KB in size is loaded into memory in its entirety, it would take up three page frames. If a file is 13 KB in size, it would take up four page frames, and the final page frame would have an extra 3 KB that would be unused. When large files are loaded into memory, large pieces of them are often paged to disk to conserve space in the volatile memory. Before a page can be used, it needs to be paged into memory; to keep track of the pages and whether they are paged in or paged out, kernels use page descriptors, which stores information about the state of pages. One of the most important structures in the Linux kernel that deals with volatile memory is the mem\_map\_array. This array holds all of the page descriptors. In Linux, there are usually three memory zones, which are the result of the operating system partitioning the memory up in order to allow the kernel to access all of it (due to hardware constraints that make this impossible if all available memory is treated as one zone). Each zone has its own mem\_map\_array structure, and this structure can be used to find objects within that zone such as processes and files that have been mapped into memory.

Strings Search (memdmp)

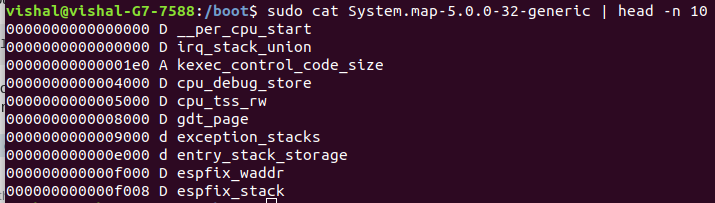
memdump | strings > strings.txt (This will probably crash the system, so define a cap)

Substitute :

XORSearch

Advantage of XORSearch : I can return the typed in string with the key it was encoded with. This way, if we can find all the data in memory encoded/obfuscated with this particular key. (This is applicable if data is obfuscated using trivial functions like XOR or ROL)

In most Linux systems, a map of memory (often located in the /boot directory and named Symbol.map or System.map) can be extremely useful in figuring out where the important locations are. Most important symbols (such as structures and functions) in the Linux kernel are shown there, along with the addresses where they reside.



The structural representation of a process is actually similar between most common operating systems, and the general methodology for recovering the list of running processes from memory is also essentially the same at a high level.

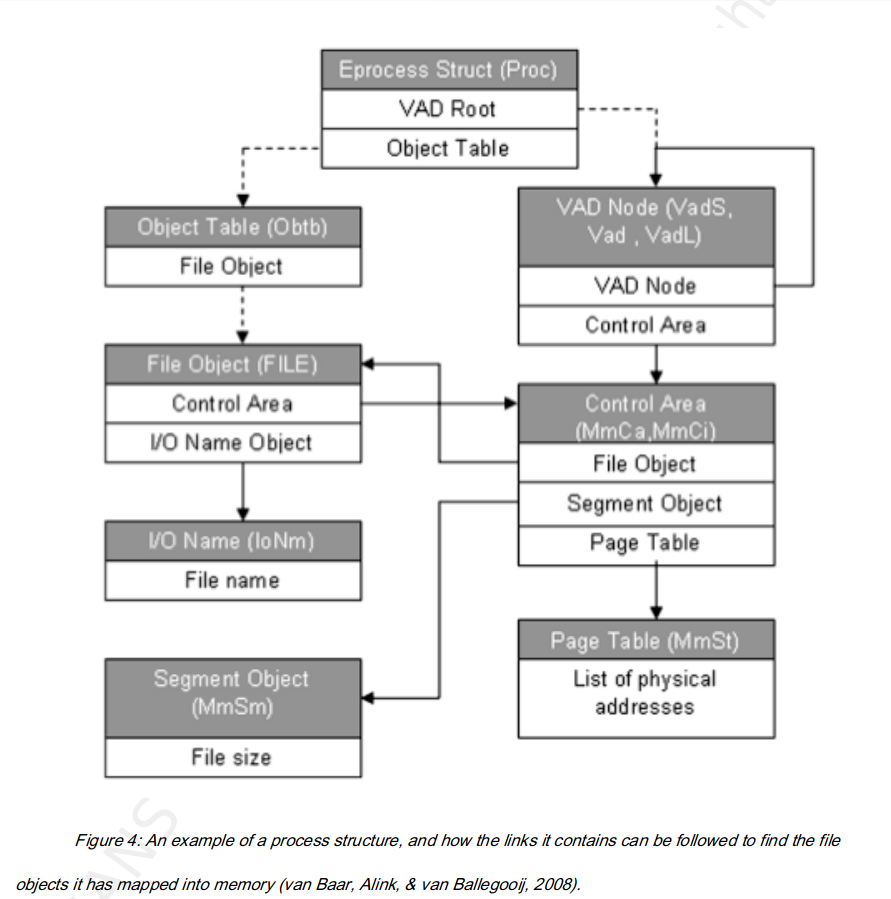
On most Linux flavors, a process descriptor is used to store information about the current state of every running process, and serves as a representation of that process. This structure is called a task\_struct, and is used to represent all types of processes from those that are invoked by a user to kernel threads. The list of currently running processes is a doubly-linked list that strings together all of the existing process descriptors.

Finding the running processes is not quite as straightforward in Windows as it is on Linux, unfortunately, because analysts don’t have a map of memory to start their search for the linked list of processes from. Instead, an analyst must usually find a starting point by looking at global kernel variables that point to the start of the list of processes.

There are several excellent references available for analysts interested in all of the low-level details, including (Burdach, 2005) for Linux processes, and (Schuster, 2006) for Windows processes.

Discovering Memory-Mapped Files in Windows and Linux

The best place to start is the root of the VAD tree, which is a set of structures that describes a processes’ memory ranges. One of the structures in the VAD tree is called an object table, which lists the private objects that are in use by a process – these can be files, registry keys, and events. The memory-mapped files associated with each process can be recovered by walking the VAD tree and pulling out the objects of interest – in this case files, but potentially other objects as well. (Look at the dig below) (**For Windows**)



For in-depth coverage of this topic, an analyst may start with the paper Forensic Memory Analysis: Files Mapped in Memory by van Baar et. al. (van Baar, Alink, & van Ballegooij, 2008) and use Windows Internals (Russinovich & Solomon, 2005) or Understanding the Linux Kernel (Bovet & Cesati, 2006) for more information on the data structures used for the recovery process.

A useful tool for recovering memory-mapped files from a memory dump is VADtools.

Just incase if the PMs delink its previous process after changing states (Effectively removing it from the doubly linked list but there is a chance the process data still resides in memory)

All types of objects, such as processes or files, have patterns to them – for example the “header” of every process object will contain some constants that will be the same for every process in memory. In order to find processes that aren’t in the doubly-linked list used to reference processes that are currently running, the analyst needs to go through the entire image of memory and search for these constant values and use that as a guide to point out process objects that would otherwise be missed

(We can find tools to parse through the image of memory for a particular header. Instead, we can also write custom script according to need.)

**Current Page 40**

Important Note :

A related problem is that when this happens and information related to capturing the memory is put into RAM, the analyst is mixing the results of the analysis with the data that was previously stored on the system. This makes it necessary for the analyst to differentiate between what was put on the system as a result of collecting the data, and what was there before. An analyst should make sure to familiarize his or herself with what the footprint of running the tools used to capture volatile memory looks like in order to quickly eliminate that data and keep from wasting valuable time trying to figure out whether it is relevant or not.

Solution :

Use an Ubuntu VM and transmit the memdmp data over network to your local machine