Techniques and Tools for Recovering and Analyzing Data from Volatile Memory

Kristine Amari

Techniques and Tools for Recovering and Analyzing Data from Volatile Memory

GCFA Gold Certification

Author: Kristine Amari, Kristine.amari@disa.mil

Adviser: Carlos Cid

Accepted: 26 March 2009

Abstract

There are many relatively new tools available that have been developed in order to recover and dissect the information that can be gleaned from volatile memory, but because this is a relatively new and fast-growing field many forensic analysts do not know or take advantage of these assets. Volatile memory may contain many pieces of information relevant to a forensic investigation, such as passwords, cryptographic keys, and other data. Having the knowledge and tools needed to recover that data is essential, and this capability is becoming increasingly more relevant as hard drive encryption and other security mechanisms make traditional hard disk forensics more challenging. This paper will cover the theory behind volatile memory analysis, including why it is important, what kinds of data can be recovered, *and the potential pitfalls of this type of analysis, as well as techniques for recovering and analyzing volatile data and currently available toolkits that have been developed for this purpose.*

1. Introduction

Computer forensics is an expansive and fast-moving field. New and evolving technologies such as cellular phones, personal digital assistants (PDAs), as well as new and ever-changing operating systems and file systems all require in-depth analysis to determine how best to extract information pertinent to an investigation. In addition, techniques for performing forensics on both new and existing technologies are constantly in development.

In such a dynamic working environment, forensic analysts must be constantly vigilant in order to keep abreast of the latest advances in their field. Many techniques are complex and time-consuming, requiring training and specialized tools. Distinct areas of research and development have emerged within the overarching theme of forensics, and it is increasingly common for examiners to choose a specific area of expertise, such as personal electronic devices or a family of file systems rather than one examiner taking responsibility for all aspects of the analysis.

This specialization allows the study of forensics to branch out in new directions and encourages the development of more technology-specific methodologies and techniques. As technology continues to evolve and become more varied and complex, so must the toolkit available to a forensic examiner.

One relatively new capability available to examiners is memory forensics. As attackers learned that they could leverage volatile memory to store data and execute code instead of or in addition to the hard disk, it became necessary for analysts to take that into consideration and develop their own methodologies for recovering this important information in their investigations. This paper is intended to be a snapshot of the current memory forensic tools and techniques available to forensic analysts. It also aims to provide guidance as to why memory forensics is valuable, and argues that it is in fact essential to the future of forensic analysis.

In this paper, we first discuss the concept of memory forensics and why it is valuable. We then move on to enumerate the types of data that can be extracted from volatile memory in section 4. Later, in sections 5 and 6 we describe current analysis techniques as well as tools that are currently available and the features they provide. Additionally, in section 7, we look at some general guidelines and precautions when performing analyses of volatile memory, as the process involves a set of risks that traditional forensic analysts do not have to consider. Finally we conclude in section 8.

2. Defining Memory Forensics

Memory forensics is a young but fast-growing area of research, and a promising one for the field of computer forensics. Whereas traditional computer forensics involves the study of persistent data storage such as hard drives and USB devices, also known as dead-box-analysis, memory forensics involves the capture and analysis of volatile memory such as RAM.

Data is considered volatile when it is likely to be lost when a machine is rebooted or overwritten during the course of the machine's normal use. Such data, because it is constantly in flux, is often not as structured in the same way that filesystems are, and can be more difficult to predict and parse into meaningful data as a result. Often, however, the artifacts that can be recovered from volatile data are valuable in pushing the investigation forward on all fronts, and many types of artifacts can only be recovered from memory.

The analysis of any volatile memory captured by an incident responder is currently a less precise art than the analysis of a hard disk. Hard disks have a strict pre-defined structure, and analysts know where to look for certain structures and data types on a specific kind of filesystem (FAT32, for instance). Memory, on the other hand, can be allocated and de-allocated to different areas depending on what memory is already being used; for all intents and purposes it is impossible to predict what you will find in volatile memory or where it will be stored.

It is readily apparent that acquiring and analyzing this type of data is more challenging and perilous than dead-box analysis. Because of the less structured approach to storage and the speed at which volatile memory is modified, analysts have to take more precautions when capturing the data and parsing through it. Virtually every action that a user performs on a computer modifies the memory on the machine, which leads to a certain amount of unpredictability in the resulting captures. The pitfalls of performing memory analysis are discussed in detail in section 7.

3. The Value of Memory Forensics

Performing memory forensics has the potential to contribute significantly to any forensic investigation where such data is available for capture and analysis. Memory forensics is extremely valuable because it overcomes several limitations of traditional forensic analysis, in addition to addressing problems that new technologies such as encryption can cause during the course of a dead-box examination. As technologies continue to evolve, memory forensics will become increasingly critical in order to effectively gather necessary evidence.

Traditional dead-box analysis is limited in several ways. The investigator cannot access encrypted data unless he or she can crack the user's password or recover the key used to encrypt the data. Keys and passwords are very rarely stored on the disk. When the user types in their passwords, or when data is decrypted, however, the passwords and keys are necessarily loaded into and stored in memory; analysis of that memory can allow the analyst to recover them.

Another limitation is imposed by the inability of the physical disk to reveal information about processes that were running in memory, which denies the investigator insight into how applications were being used on the system at the time of the attack. It is also possible for a suspect to hide data in memory, or for a remote attacker who has compromised a system to store tools, data, and other artifacts there rather than on the system's drive.

Furthermore, it is becoming increasingly common for attackers to write viruses, Trojans, and worms that reside only in memory and do not write themselves to the physical disk drive (Szor, 2005). As a result, traditional forensic analysis of the disks will not reveal the code or allow analysts to understand how the attack is being executed or how to mitigate it. As an example, the SQL Slammer worm, the fastest-spreading worm to date, exists only in the memory of the infected box and does not write its own code or any output it produces to the physical disk of a computer it infects (Moore, Paxson, Savage, Shannon, Staniford, & Weaver, 2003).

The techniques referred to above, including data contraception, data hiding, and data destruction, are often referred to as anti-forensic techniques and are discussed in detail in section 5. It is important to note that these techniques were designed by attackers to thwart traditional forensics; hence the value inherent in using memory forensics to supplement a dead-box analysis.

4. Data Found in Volatile Memory

There is a wealth of data available in volatile memory. Processes, information about open files and registry handles, network information, passwords and cryptographic keys, unencrypted content that is encrypted (and thus unavailable) on disk, hidden data, and worm and rootkits written to run solely in memory are all potentially stored there. This section will go into detail about exactly what types of information may be recoverable via memory forensics.

4.1 Processes

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.

4.2 Open Files and Registry Handles

The files that a process has open, as well as any registry handles being accessed by a process, are also stored in memory. Information about the files that a process is using can be extremely valuable. 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).

In the Unix environment, for example, files that are mapped to memory are generally described by an inode structure, which stores information about the memory-mapped file such as modification, access, and change times of the file, the name of the file, and information on the directory the file was executed from if the file happens to be an executable (Burdach, 2005). There is also a field that points to an address_space structure, which usually maps the pages in memory that make up the file to the physical disk blocks where the data resides. As any forensic analyst can see, there is a wealth of information to be gleaned from this data structure alone that could help to track down additional relevant information, or that could be used to validate facts that have already been established via analysis of the physical disks (for example, MAC times).

4.3 Network Information

Information about network connections, including listening ports, currently established connections, and the local and remote information associated with such connections can be recovered from memory. This is useful because tools that are run on the machine itself, such as **netstat**, can be trojanized by a malicious intruder or user to provide false information back to the analyst. When pulling the information directly from a memory dump using the data structures themselves, it is much harder for an attacker to hide their listening backdoor, or the connection to their home server from which they are transferring malware and other harmful or illegal files. Network connection information is one of the most critical pieces of information that can be gleaned from a computer that is being investigated, and it is more reliable when it comes from static analysis of a memory dump.

4.4 Passwords and Cryptographic Keys

One of the most critical advantages of memory forensics is the potential for recovery of user passwords and cryptographic keys that can be used to decrypt files of interest and access user accounts. 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. Password recovery can also allow examiners to access online accounts owned by a suspect such as email and data storage.

4.5 Unencrypted Content

While recovering keys and passwords can lead investigators to encrypted content, it may be possible to recover data from encrypted files without having the key. When the suspect accesses an encrypted file, the content is unencrypted and loaded into memory. This unencrypted content may remain in memory even after the suspect has closed the file, as long as it is not overwritten by something else. Parsing through volatile memory may reveal fragments of files, or even whole files that would otherwise be unrecoverable if the key or password used to encrypt the data could not be discovered.

4.6 Hidden Data

It is feasible for malicious entities or suspects with data they want to protect to store their data in volatile memory instead of on the hard disk. Because investigators traditionally do not inspect volatile memory, it is a safer place to hide information than on a hard disk. It also makes it easy to destroy incriminating or sensitive information - all the user has to do is pull the plug, and a remote attacker can usually manage to cause a machine to reboot if desired. In addition to hiding files in memory, attackers can also run malicious code from memory instead of storing it on the disk, making it difficult for reverse engineers to obtain copies of programs and figure out how they are working and how to mitigate the threats they pose (Eilam, 2005). This is discussed further in the following section on malicious code. Checking volatile memory for hidden files and code can lead to discovery of critical information.

4.7 Malicious Code

Recently it has become increasingly more popular for attackers to run exploits from memory instead of storing malicious code on the hard disk itself. This is primarily to avoid detection, since current anti-virus software and other malware detection tools are not currently as good at analyzing volatile memory for malicious code as they are at analyzing the hard disk, and some do not have this capability at all. Storing malware in memory also benefits attackers by making it harder for analysts to recover and reverse-engineer their code. An example of a relatively new and revolutionary rootkit that runs in memory and leaves no trace on the affected user's system is Shadow Walker (Sparks & Butler, 2005). Currently the known examples of this type of malware are proof-of-concept, but it seems to be a safe bet that in the future attackers will increasingly trend toward using this type of tool. Furthermore, the lack of real-world examples may be due to the difficulty inherent in detecting and recovering such tools with today’s technologies and forensic methodologies.

5. Current Analysis Techniques

In order to acquire volatile memory and analyze it, first an analyst must have a technique for acquiring memory. They must also know where to locate it and how to obtain a copy of it. These details are described in sections 5.1 and 5.2. Section 5.3 describes the persistence of data in volatile memory, namely how long an analyst can expect data to remain in memory once it has been loaded. The remaining sections describe the techniques used to analyze memory once it has been successfully acquired.

5.1 Acquiring Volatile Memory

There are two methods of acquiring volatile memory: hardware-based acquisition, and software-based acquisition. Both methods, including their pros and cons, will be described in this section. In general, from a forensics perspective, it is better to use hardware-based acquisition because it is more reliable and difficult for an attacker to corrupt, but currently software-based acquisition is the far more popular method due to its cost-effectiveness and ease of availability.

Hardware-based acquisition of memory involves suspending the computer's processor and using direct memory access (DMA) to obtain a copy of memory. It is considered to be more reliable because even if the operating system and software on the system have been compromised or corrupted by an attacker, we will still get an accurate image of the memory because we do not rely on those components of the system. The downside to this method is cost – special hardware must be purchased in order to perform the acquisition of memory in this way. One piece of specialized hardware designed specifically for this purpose is the Tribble card (Carrier & Grand, 2004), which is a PCI card that must be installed in a system before a compromise takes place in order to allow the investigator to capture the memory in a way that is more accurate and reliable than software-based approaches.

Software-based acquisition is most often done (and should always be done) using a trusted toolkit that the analyst brings to the site, but it is also possible to collect volatile memory using tools built in to the operating system (such as **memdump** or **dd** on Unix systems). Whether the tools are trusted or already on the system, it is easy for an attacker to circumvent this technique if they have compromised the operating system of the computer that is being analyzed since the attacker can hide relevant data by modifying system calls and internal system structures. Another downside of software-based acquisition is that executing it in order to capture the memory will alter the contents of the memory, potentially overwriting data that is relevant to the investigation. On the other hand, the tools necessary to perform software-based acquisition are free and readily available.

The next section, section 5.2, describes where volatile memory is found on different types of systems and includes some details on how the acquisition process works.

5.2 Where to Find Volatile Memory

Volatile memory is accessed via different mechanisms depending on the operating system being used, and the hardware in the machine itself.

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.

Another useful way of acquiring physical memory in Windows is to use a registry key to cause a BugCheck trap when a specific key sequence is executed, which causes a memory dump to occur (Microsoft Corporation, 2005). The resulting dump will be in a different, more complex format than a byte-by-byte copy of memory, but it has the advantage of being compatible with Microsoft's debuggers, which can help the analyst better understand the kernel data structures. There are not currently many tools that leverage this format to perform analyses, but it is useful to know about in any case.

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).

Now that we have discussed how to find and acquire memory, we move on to a brief discussion of how long data tends to stay in memory before being overwritten. After that, the next sections discuss techniques for extracting any data that happens to be in the capture obtained by the analyst.

5.3 Persistence of Data Stored in Memory

In this section we try to gain some insight into the lifespan of data that is stored in volatile memory – for example, once a program is loaded into memory when it is run, how long can we expect it to remain in memory? We are also interested to see whether other data follows similar patterns, or if other types of information that are commonly stored in memory have entirely different lifecycles.

Intuitively, we can conclude that as long as a program continues to run it will continue to reside in memory. But what happens after the program exits and the memory it inhabited is deallocated? This area of study is not well mapped out because of how unpredictable memory is due to the multitude of factors that can influence when and where memory is allocated, deallocated or overwritten. One of the few papers that features a study of this information is *Data Lifetime is a Sytems Problem* (Garfinkel, Pfaff, Chow, & Rosenblum, 2004). Figure 1 shows a graph from their research on memory persistence on the Solaris operating system.

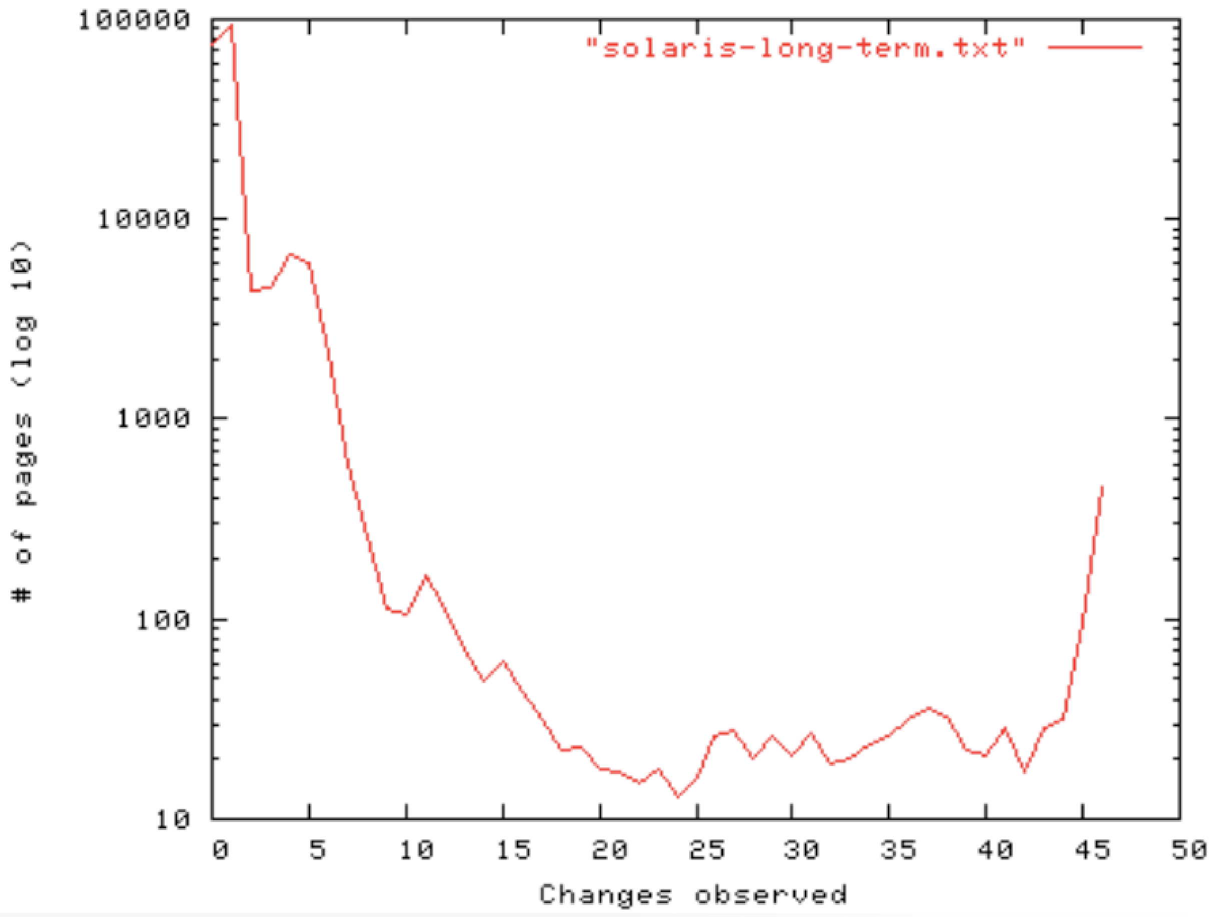


Figure 1: A graph of the number of changes in memory over time on a Solaris 8 machine set up as a DNS server with 768MB of RAM (Garfinkel, Pfaff, Chow, & Rosenblum, 2004).

This graph shows the number of days that the machine has been running on the x-axis, and the number of pages that change on the y-axis. In this study, 86% of memory did not change.

Furthermore, Garfinkel et. al. (Garfinkel, Pfaff, Chow, & Rosenblum, 2004) showed that metadata about processes and other objects can survive in physical memory for more than 14 days while the system is in use. Their study showed that even in the most extreme case they saw during their research, there were still 23 KB of tracked data on the system after 14 days, and on that same system, after an additional 14 days, there were still 7 KB of tracked data. This may not seem like much, but 7 KB is still enough to contain passwords, cryptographic keys, parts of incriminating files, and other fragments of data that an analyst might find useful.

Even more interesting was the work that was done to study the effect of rebooting a machine on that machine's physical memory. Garfinkel et. al. found that contrary to popular belief, a soft reboot (one that does not turn the power off completely) will leave most of the data in RAM intact. Even more surprisingly, on some hardware a hard reboot (one where the power is completely turned off) did not clear the memory completely either – specifically, they found that on some hardware data persisted in volatile memory after up to 30 seconds without power.

This is encouraging news for forensic analysts, but it is important to remember that there are a multitude of factors that contribute to data persistence in volatile memory. The type of operating system is a major factor, as is how much memory is available to swap data in and out of memory. The less data is available, the more often the operating system will have to overwrite it. The less efficient the operating system is in allocating the available chunks of memory, the more sporadic the allocation will be across memory as a whole. In addition, the level of activity on the machine itself plays a huge role. The more work the machine performs the more it will be loading data into memory and swapping it back out. Finally, as mentioned earlier in this section, certain types of data may be more likely to be swapped out or overwritten if the system needs more space in memory for new processes that it needs to run, or other data it needs to store.

5.4 How Volatile Memory Works

In order to understand some of the information in the following sections, it will be necessary to have at least a basic understanding of how volatile memory works. This section will attempt to outline the central concepts necessary to understand how memory generally works on both Windows and Linux. For a deeper understanding of this material, two sources were particularly helpful in writing this paper: *Windows Internals* for the Windows operating systems (Russinovich & Solomon, 2005), and *Understanding the Linux Kernel* for Linux operating systems (Bovet & Cesati, 2006).

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 VADtools, 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.

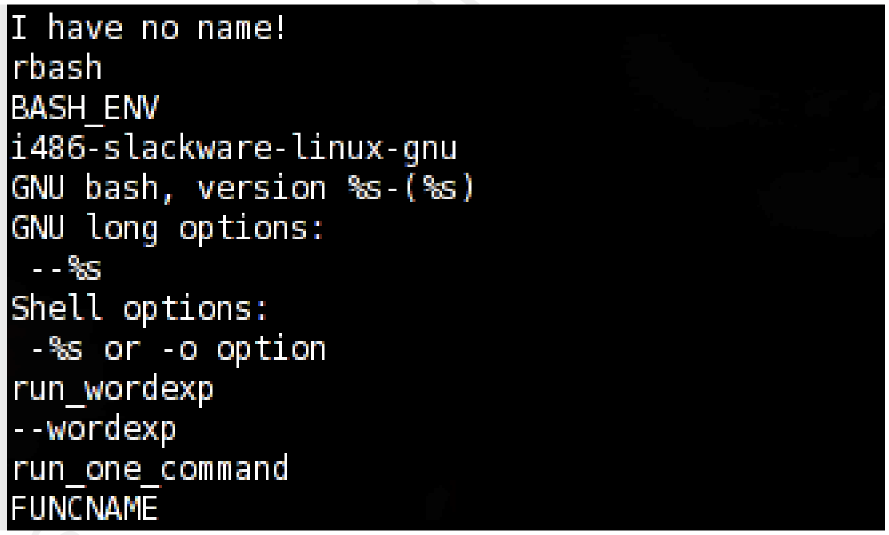
One of the most important structures in the Linux kernel that deals with volatile memory is the mem_map_array. This array holds all of the page descriptors. In Linux, there are usually three memory zones, which are the result of the operating system partitioning the memory up in order to allow the kernel 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.

This information about the detailed layout of Windows and Linux memory and how each operating system handles objects within memory will be useful in understanding sections 5.7 and 5.8, which discuss recovering processes and memory-mapped files from volatile memory.

5.5 Strings Search

When an analyst acquires an image of the volatile memory from a machine that is being investigated, one of the first things he or she will want to do us run a search across the image for anything recognizable. This can be done with a simple command-line tool called **strings** that is native to most Unix distributions and has been ported to Windows as well. There are also GUI tools for Windows that will perform the same functions.

The command line version of **strings** can be run using a few options that will find both Unicode strings as well as ASCII strings, and the analyst can specify how long a string of characters should be before the program reports it as interesting (this is useful because a two-letter string is likely to occur by chance quite frequently and looking through these strings is very unlikely to benefit the analyst. Figure 2, below, shows an example screen capture from my own system after running the command '**./memdump | strings > strings.txt**'. This command dumps physical memory and then filters it through the **strings** command to produce all of the strings in the dump of memory, then writes that data into a file called strings.txt. [Note: in this example I wrote the output to my system, but in a real incident response effort the output of memdump should instead be piped over the network. An example command using netcat would be '**./memdump | nc host port**' where host is the IP address of a machine that is running a netcat listener configured to run on the port specified by '**port**'.]



```
I have no name!
rbash
BASH_ENV
i486-slackware-linux-gnu
GNU bash, version %s-(%s)
GNU long options:
 --%s
Shell options:
 -%s or -o option
run_wordexp
--wordexp
run_one_command
FUNCNAME
```

Figure 2: A fragment of a capture of physical memory, filtered through the strings command.

There is also a similar tool, **XORSearch** (Stevens, 2007), that is quick to run and useful if the analyst has a specific list of keywords that he or she is looking for. **XORSearch** takes a keyword as input and will perform a search for it within a memory image, and is able to find it even if the keyword has been obfuscated using the "exclusive or," known in shorthand as

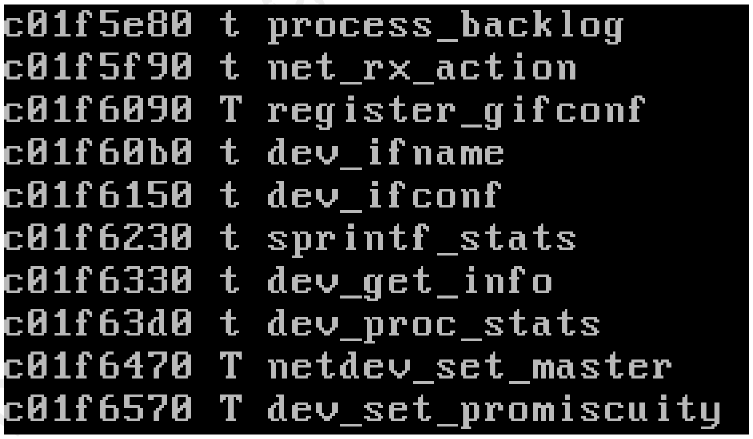
XOR, function, or the "rotate left," or ROL function. If a keyword is found, **XORSearch** will print out the value that was used as the "key," or the value that was XORed with the keyword in order to obfuscated it. If the keyword has been encoded more than once using different keys, the program will report both of the keys.

Once an obfuscated keyword is found and the analyst knows the key that was used to hide it, the analyst can use **XORSearch** to perform a search across the image for any other strings that were encoded using the same key. This can lead to recovery of significantly more obfuscated data that may aid in the investigation. In some cases, several different keys will be used by the attacker; once some data has been recovered using this method, it is a good idea to perform additional searches to make sure that all the available data is recovered.

Unfortunately, advanced attackers will obfuscate their data, making the **strings** function unlikely to turn up useful information. Advanced adversaries would also be very unlikely to use something as elementary as an XOR or ROL function to hide the important data they do not want analysts to recover; more likely they would use more advanced encryption techniques. Still, there are some attackers who do use these techniques, and both are quick and easy to perform. Even if nothing useful is found, the analyst will at least have confirmed that if there is truly valuable information on the machine in question, the suspect has attempted to hide it and he or she is likely to be using more advanced techniques than substitution.

5.6 How Memory is Organized

The structure of volatile memory is important to understand if an analyst hopes to extract anything meaningful from a capture of a machines memory. 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. As an example, a snippet of the **System.map** file from my Redhat machine is shown in Figure 3.



```
c01f5e80 t process_backlog
c01f5f90 t net_rx_action
c01f6090 T register_gifconf
c01f60b0 t dev_ifname
c01f6150 t dev_ifconf
c01f6230 t sprintf_stats
c01f6330 t dev_get_info
c01f63d0 t dev_proc_stats
c01f6470 T netdev_set_master
c01f6570 T dev_set_promiscuity
```

Figure 3: Capture from the System.map file on a Redhat 3.2.2-5 machine.

The capture shows the address of a particular symbol, for example **dev_get_info**, which is a function, and the address where it resides in memory, **c01f6330**. This information could be used in combination with the definition of the **dev_get_info** function to find all the information stored there and extract it from a memory capture.

On Windows, there is not an equivalent way of finding objects in memory – the process is far more complex. The analyst must know what he or she is looking for, and have detailed knowledge of Windows internals for the version of Windows that is being analyzed and how memory is organized on that system. In the following section, 5.7, the procedure for locating the list of processes is outlined.

5.7 Enumerating the Running Processes

One of the first things an analyst will want to do with a capture of volatile memory is parse through it looking for the processes that were running when the capture was taken. 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. This section discusses both Linux- and Windows-specific details in order to provide an overview of the techniques used on the more common operating systems.

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 (Russinovich & Solomon, 2005).

On Windows, process structures look somewhat similar to those seen on Linux operating systems – there is a basic structure that holds all of the process-specific information, and a doubly-linked list of these structures is used to keep track of all the currently-running processes in memory. 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 (Schuster, 2006).

Covering every detail of the process structure is beyond the scope of this paper, but 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. Section 6 describes some of the currently available tools that automate the procedure of carving out interesting process-related data, and help the analyst by parsing through memory using the correct parameters and outputting a list of the running processes based on what is found in the image.

5.8 Recovering Memory-Mapped Files

In Windows, the data structures that store file mappings are allocated by the kernel from a memory pool (a dynamic storage area in memory that is allocated by the kernel). There are several different types of structures that are used to describe a file object. Often, these structures are found by looking at a process data structure and finding the files that are mapped in association with that process; a representation of what this looks like is shown in Figure 4 (van Baar, Alink, & van Ballegooij, 2008).

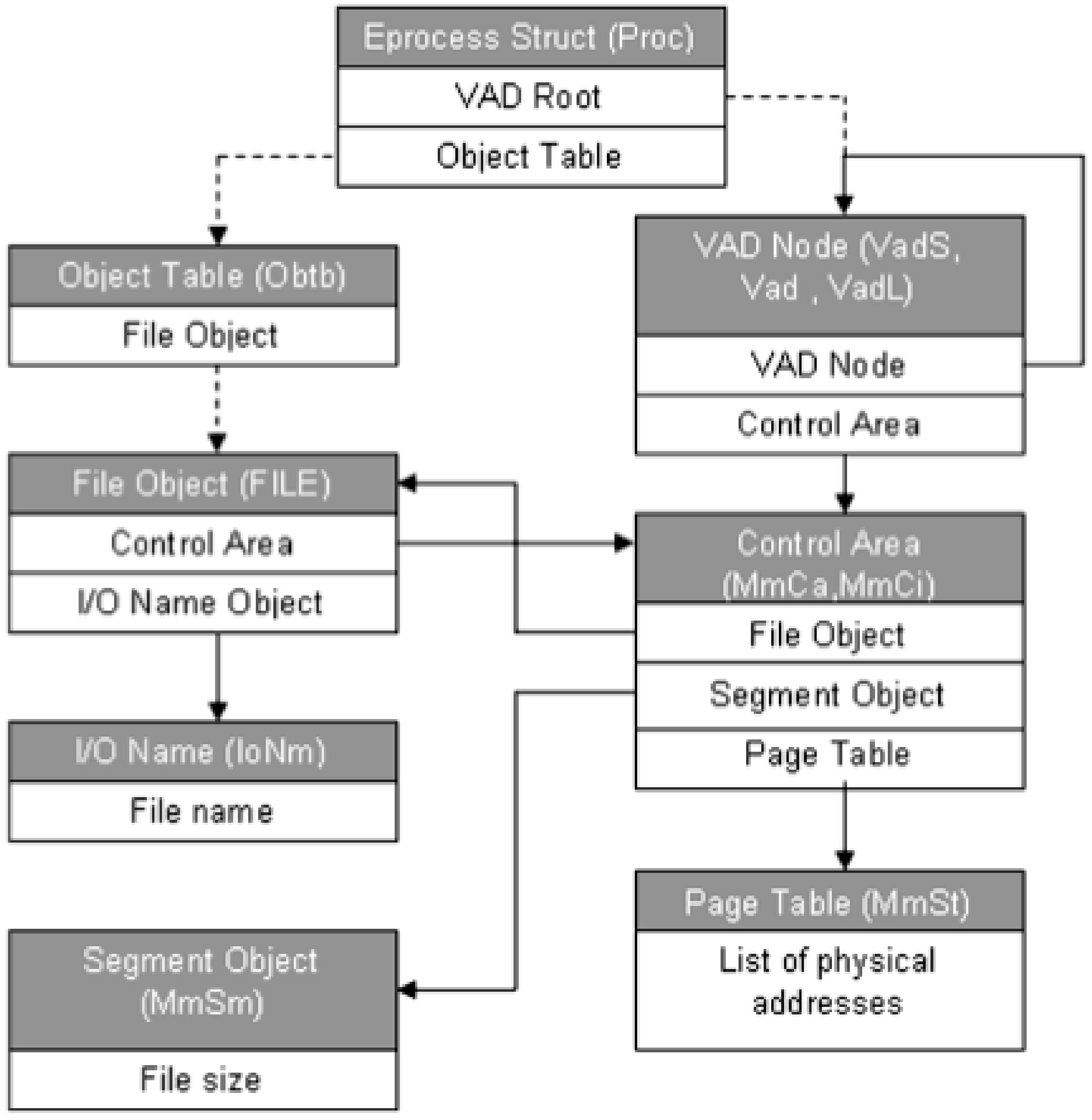


Figure 4: An example of a process structure, and how the links it contains can be followed to find the file objects it has mapped into memory (van Baar, Alink, & van Ballegooij, 2008).

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.

As with processes that have been terminated, files that have been closed often remain in memory, but are no longer linked through the lists maintained by the operating system and must be found using alternative methods. The process of recovering such files is similar to reconstructing files on a hard disk that have been deleted, though the fact that memory is usually far more fragmented than a hard disk makes the process more involved. Often, looking at the page table will allow files to be reconstructed even if they are no longer active in memory. There is also an area of memory called the "Control Area" that maintains links between file names and the file data stored in the pages; if this area is still present the file name can often be recovered as well (van Baar, Alink, & van Ballegooij, 2008).

In Linux, memory-mapped files are described by **inode** structures, which are the same structure used to describe files stored on the hard disk. Using this object, it is possible to obtain information about the directory where the file was executed from, discover the MAC times on the file, and do much more. There are many resources that describe the detailed structure of inodes and enumerate all of the information that can be gleaned from them. To find memory-mapped files, an analyst can first enumerate the processes (as described in the previous section), and for each process look at the file structures associated with that process. All of the memory-mapped files associated with a process can be found in this manner.

A useful tool for recovering memory-mapped files from a memory dump is VADtools (Dolan-Gavitt, 2007), described in more detail in section 6.8. Other tools can also perform this function; see section 6 for more detailed discussions of forensic tools that can be used to analyze volatile memory. 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.

5.9 File Signature Search

In section.5.8, a complex but fairly reliable technique for recovering memory mapped files from a dump of volatile memory was discussed. There is another older and less reliable technique for recovering files from both hard disks and memory that is commonly used in tools like Encase (see section 6.9). This alternate technique is often referred to as "file carving."

Different types of files (for example word documents or zip-compressed files) have different signatures. A signature, in this case, refers to specific patterns of values that are unique to the particular type of file in question. File carving can be done linearly, which means that contiguous files can be recovered, but fragmented files cannot. There are also more complex file carving algorithms that are able to recover fragmented files by looking more deeply inside the data structures that describe a them.

File carving is very useful when analyzing a disk because most operating systems try not to fragment files which makes them easy to recover using straightforward linear techniques. Unfortunately, it is often very difficult to use file carving to recover data from a memory dump, since files are far less likely to be contiguous in memory than they are on disk. Even if a smarter algorithm that can handle fragmentation is used, it is common for only pieces of files to be loaded into memory, which can throw off the tool and prevent it from finding the correct parts of the file and recovering them.

Figure 5 shows the average number of fragments per file versus the number of blocks in the file in a memory dump. This graph, derived in research by R.B. van Baar, W. Alink, and A.R. van Ballegooij (van Baar, Alink, & van Ballegooij, 2008), demonstrates visually that it is nearly impossible to construct files from memory using a linear method, and even one that can handle fragmented files will fail because often the page has not yet been loaded into memory.

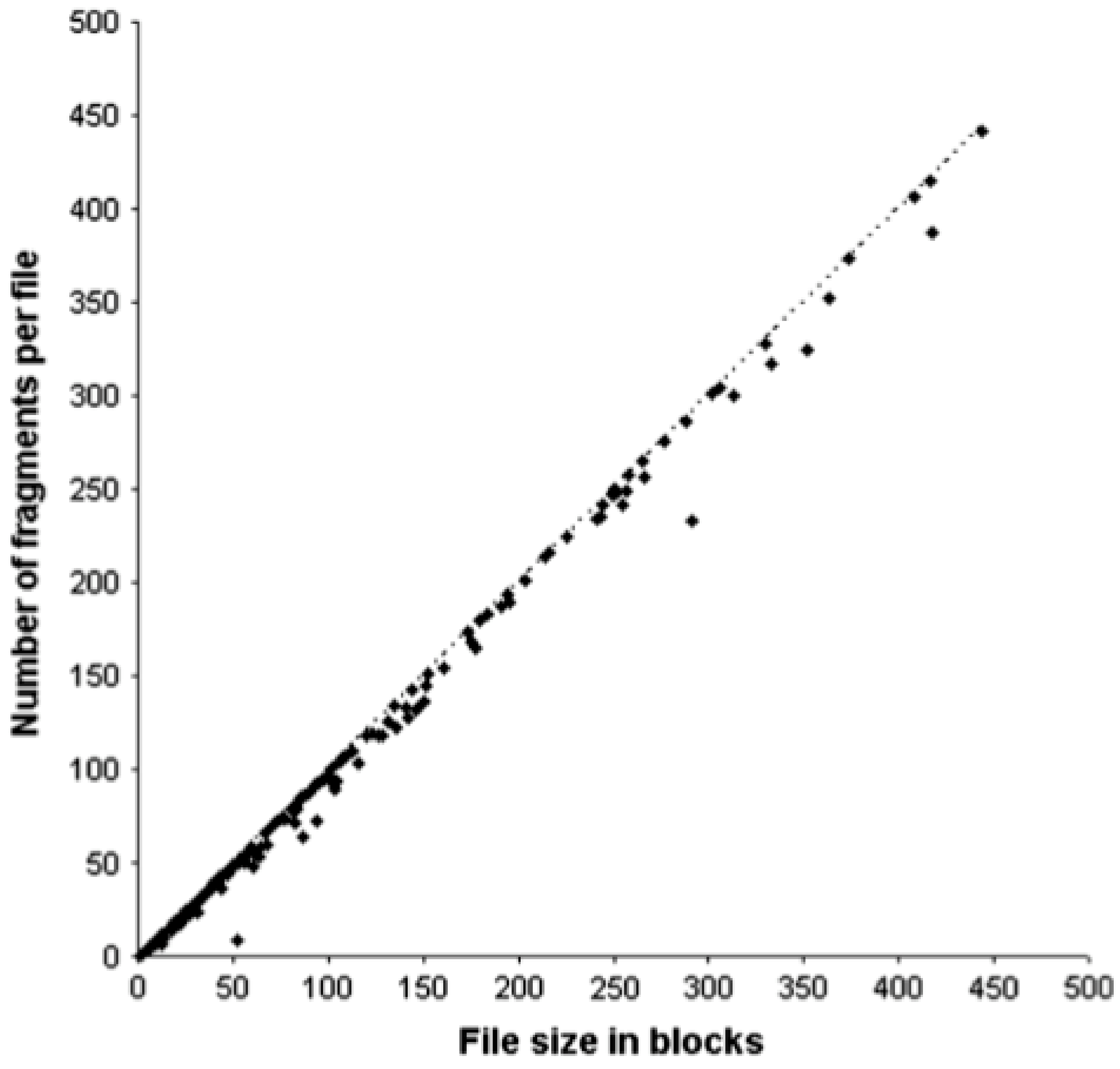


Figure 5: A craph of the number of blocks in a file versus the average number of fragments per file in a

memory dump. The dotted line indicates a fully fragmented file (van Baar, Alink, & van Ballegooij, 2008).

While file carving is a good technique and a very useful one to have in the toolkit of every forensic analyst, it is not optimal and for the best results, the methods described in the previous section should be used when possible. There are many tools available that can perform file carving, such as **PTFinder** (Schuster, 2007).

5.10 Detecting and Recovering Hidden Data

In sections 5.7 and 5.8, we talked about recovering processes, threads, and memory-mapped files by working from the lists that the operating system maintains of these objects. The procedures discussed in those sections are useful and important to understand, but analysts must also understand that these techniques will not find *all* processes or threads, for example. This is because once a process is terminated, or if the operating system is instructed to hide a process, the data structure that defines the process will no longer be a member of the linked-list data structure that the operating system maintains to keep track of what is currently running.

It is good to establish a way of recovering these types of objects because often they are also related to the investigation – an attacker or a suspect may have closed files or terminated processes prior to the incident responder arriving on the scene, but there is a good chance that data will still be in memory. Even more concerning, an attacker may use direct kernel object manipulation (DKOM) to remove a suspect process or other type of object from the lists or tables that the kernel uses to keep track of these resources. Doing so will effectively hide the target object from the Windows API, as well as any other technique that finds objects by walking down linked lists or enumerating tables. This section describes some methods of recovering such data using techniques that don't rely on the data structures maintained by the operating system.

The basic premise for searching for objects in memory that are not accessible through currently active lists of objects used by the kernel is quite simple. 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.

This is, of course, a very tedious process to go through by hand. Luckily, there are many tools that have been written to automate the search for common objects like processes and files. While it may be necessary for a motivated analyst to write a custom script to search for something out of out the ordinary that no one has created an open-source or commercial program to find, in general, especially for common operating systems like Linux and Windows, it is possible to find programs that will parse out the items that are of interest.

We will now detail some example characteristics that are used by the available automated tools to find hidden process objects. Processes are represented in Windows by an **EPROCESS** structure, which contains other values and structures that describe important information necessary to the running process. The pointer **DirectoryTableBase** points to the beginning of the relevant structures, and can be used to traverse the information in order to find the relevant pieces to check against the general "signature" of a process that has been mapped out to help reveal hidden processes. One of the first checks is the value of **PageDirectoryTable**. This should not be equal to zero, and if taken modulo 4096 a value of zero should be the result. Additionally, every process requires at least one thread; threads are stored in a structure that is basically another doubly-linked list. Two pointers (called **ThreadListHead.Flink** and **ThreadListHead.Blink**) are both checked to make sure they point to an address greater than 0x7fffffff. This means that both the pointers must point into kernel space (the top of the memory address space) where the structure is stored. There are other important checks that must be done that, if found to match, increase the probability of the object in memory actually being a hidden process. In this way, by following these rules to parse memory, it is possible to find hidden processes in memory (Schuster, 2006).

6. Current Tools

In this section some of the current tools available to forensic analysts interested in performing memory forensics are described and analyzed. This analysis focuses primarily on freely available tools that any analyst may obtain and use. There are also commercial tools available, briefly addressed in section 6.9. This is just a subset of the currently available tools and should not be considered an exhaustive list. In addition, addressing the full scope of the capabilities of each tool is beyond the scope of this paper.

We also note here that when trying out new tools, or trying to determine the accuracy of a tool, it is vital to test against a trusted baseline. As an example, if an analyst is trying out a tool that parses hidden processes out of memory and there is little actual field information available on the tool, he or she should test the tool on known images of memory and make sure it finds everything it is expected to find. Another way to test new tools is to use a similar tool that has been proven to work correctly as a basis for comparison so that the analyst is able to tell if the tool is missing critical information or providing back false positives. This footwork is essential, because if a tool cannot be proven to work correctly in every situation, evidence and analysis performed with that tool will not hold up in court and may cause an investigation to fail to produce the correct verdict.

It is important to note here that many of the tools discussed in this section have very little information about them documented and available. The following sections attempt to compile the information that is available, and in some cases test the tool to determine additional functionality and ease of use.

6.1 Basic tools

There are some general-purpose tools that can be useful in analyzing virtual memory. An example is the **strings** tool that was covered in section 5.5. Other examples include **netstat**, which lists a variety of information about active network connections, listening ports, and other network state information, **lsof**, which lists the files that are currently open on the machine in question, **ps**, which lists currently running processes and related information, and **ifconfig**, which lists network interface configuration details.

Other tools that are not commonly found on systems but may be installed by a system administrator or brought to the scene by an incident responder include the Sysinternals suite maintained by Microsoft, the Foundstone tools, and the resource kits for Windows. These tools can offer the means to pull specific information about processes, open files, etc., from memory, depending on what the analyst is looking for. None of these tools has been specifically designed with forensics in mind, but they can still offer some valuable capabilities.

Both the native system administration tools and the supplementary suites of tools offered by various groups are useful and often utilized by system administrators to check on the state of the system. They are also used by some incident responders to collect information from volatile memory for the analyst to use in the investigation. There are, however, a few problems with using these tools for a forensic investigation.

An analyst should never use the tools on the system itself, because if an attacker has compromised the system the executables could have been modified to return false information or hide anything related to the attacker's activity. Most analysts bring a copy of their own tools on CD or USB thumb drive in order to mitigate this risk. Even when this approach is taken however, it is important to note that running the tools could overwrite information related to the investigation. This risk is difficult to avoid unless the incident responder has a hardware-based acquisition device, as discussed in section 5.1.

6.2 Memdump, KnTTools

Memdump (Farmer & Venema, 1999) is a free tool that runs on many different systems, including Windows, Linux, and Solaris. It is easy to download, compile, and use, and is very straightforward in its functionality; it simply creates a bit-by-bit copy of the volatile memory on a system. Ideally it should be written off of the machine being examined; one way to do this is to use **netcat** as follows: '**memdump | nc host port**' where host is the IP address of an analysis machine running a netcat listener on the port specified as the last argument.

KnTTools (GMG Systems, Inc., 2007) is a memory acquisition and analysis tool that was created for use with Windows systems. The acquisition component, **KnTDD** can capture the physical memory and store it to a removable drive or send it over the network for archival on a separate machine. Captures can be compressed to several different formats using the tool. It also has the ability to convert a binary capture into the Microsoft format for crash dumps, which could be useful for analysts who prefer the Microsoft crash dump format. The analysis component of the **KnTTools** suite is called **KnTList** and will extract evidence from the captured memory by reconstructing the relevant Windows operating system data structures. It can output report information in XML format to make analysis of the resulting data easier.

6.3 FATKit

FATKit, developed by Petroni, Walters, Fraser, and Arbaugh (Petroni, Walters, Fraser, & Arbaugh, 2006), is a popular memory forensics tool that automates the process of extracting interesting data from volatile memory. Once the data has been extracted, **FATKit** also has the ability to visualize the objects it finds to help the analyst understand the data that the tool was able to find. The tool is able to analyze structures that are specific to both Linux and Windows kernels. Figure 6 shows a graphical overview of the different modules that compose **FATKit** (Volatile Systems, 2008). Because the tool is modular, it is easily extended by an analyst who wants additional support of different operating systems or file systems; it is also scriptable to allow analysts to develop their own custom extraction techniques.

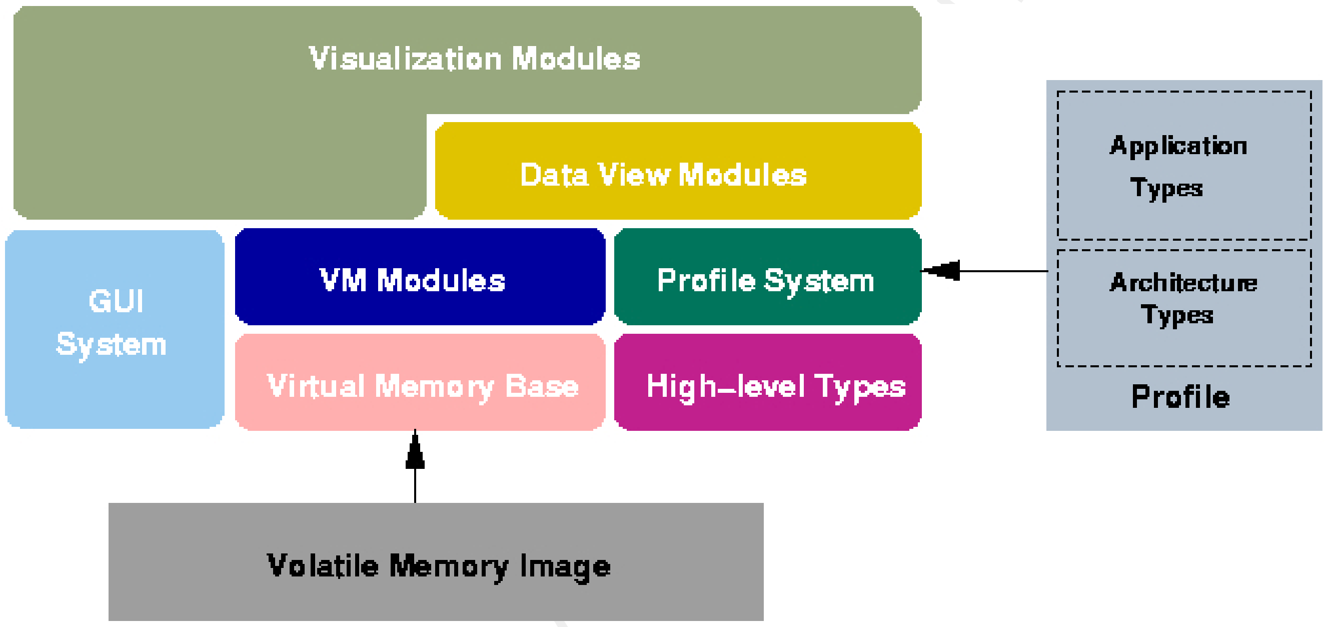


Figure 6: A graphical representation of the FATKit software architecture (Volatile Systems, 2008).

On a lower level, FATKit uses several different techniques to provide useful output to the analyst. It is able to reconstruct virtual address spaces used by processes, and translate between virtual and physical addresses to allow for an accurate picture of where data actually resided in memory when the machine was running.

Another useful feature of FATKit is its ability to detect malicious code that is residing in volatile memory.

6.4 WMFT

The Windows Memory Forensic Toolkit (**WMFT**) (Burdach, 2006) supports the analysis of memory images from machines running Windows 2000, Windows 2003, and Windows XP. There is also a Linux version available, but its functionality is currently somewhat limited in comparison with the Windows version.

In order to use the **WMFT**, the analyst must first locate symbols that point to important objects and structures in the memory. Locating the symbols is the hard part, and is outlined in the freely available paper "Introduction to Windows Memory Forensics" by Mariusz Burdach (Burdach, 2005). Once the symbols have been located, the analyst can plug the values (corresponding to the locations in memory) into **WMFT** and parse through the data structures to recover processes and other objects stored in memory.

Because **WMFT** uses the structures themselves to traverse memory and pull out the relevant information, it is vulnerable to advanced attacks where an attacker hides processes and other objects by leaving them out of the data structures such as linked lists which are used by the kernel to keep track of such data.

6.5 Procenum

Enumerating user-mode processes is one of the basic capabilities necessary to conduct any forensic analysis of volatile memory. **Procenum** (Burdach, 2006) is a utility that will perform this action by looking at all of the page descriptors that are allocated by the operating system.

The technique used by **procenum** is very generic, which makes it broadly effective against processes that an attacker has hidden by using code patching, by modifying function pointers, or by using direct kernel object manipulation. These techniques for hiding processes are described in more detail in section 4.6.

6.6 Idetect

Idetect (Burdach, 2006) is a linux-only tool that looks at an image of memory and attempts to extract detailed information about active processes. It also may allow an investigator to find out the contents of a file if that file was mapped into memory in the past. Any structure that relates to the process that the investigator is interested in can be inspected using **idetect**.

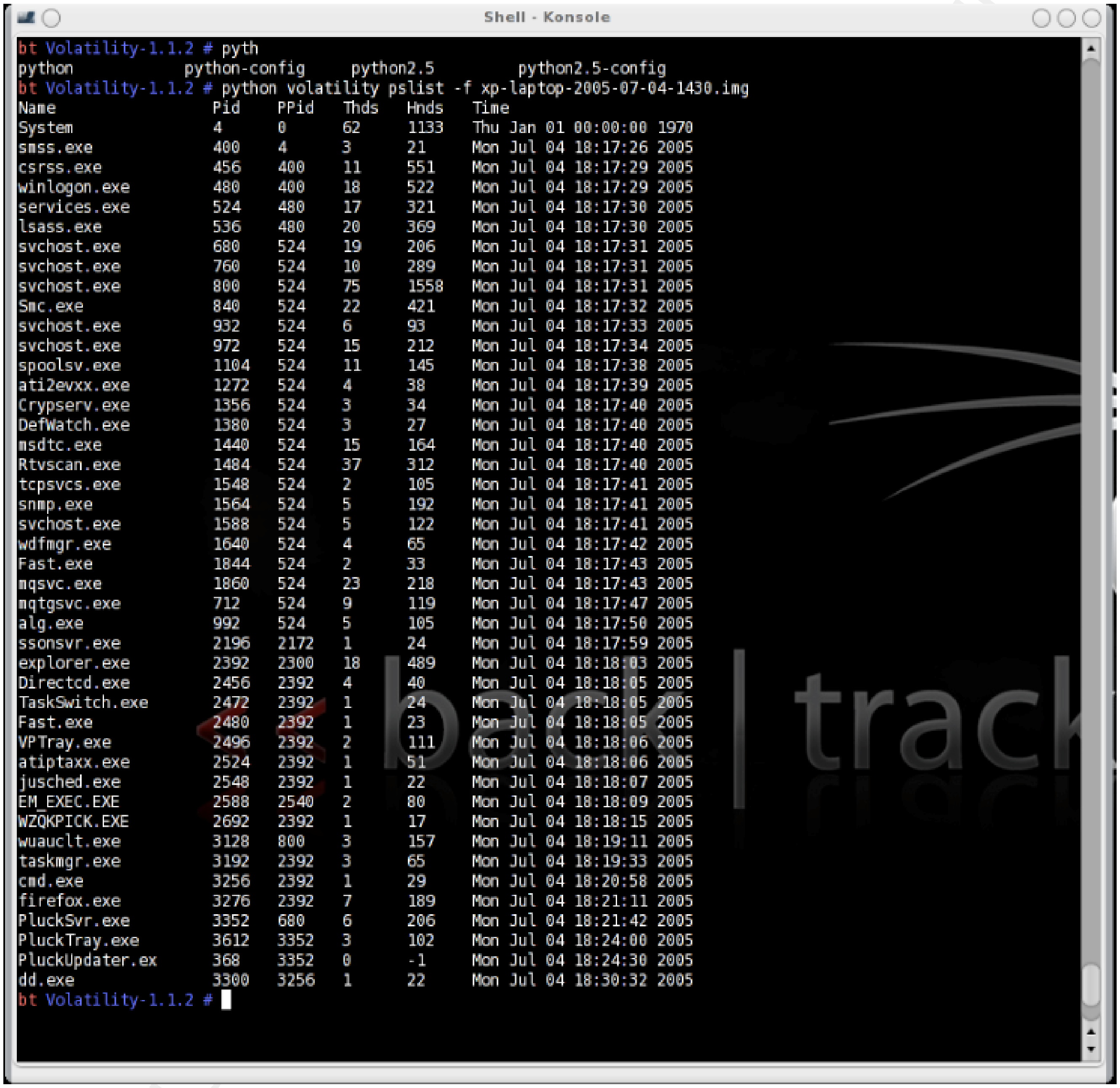
This tool can be used against live systems as well as images of memory created during an incident response effort, which, while not ideal for most forensic efforts, provides some flexibility in the event that some circumstance prevents taking the system in question offline.

6.7 The Volatility Framework

Volatility (Volatile Systems, 2008) is a collection of tools designed to be used as part of incident response and forensic analysis efforts where analyzing volatile memory is necessary or desired. It is free, open source, and written in the Python scripting language. It provides a platform to analyze and extract objects from memory dumps, and supports several operating systems including Linux, OSX 10.5, and Windows.

The Volatility framework supports a wide variety of commands, including commands that list open network connections, print a list of open DLL files, print out the memory map associated with the memory dump being analyzed, print a list of the open files associated with a process, and much more. One of its exciting features (particularly for malware analysts) is its ability to reconstruct and write out an executable sample from its associated process.

The Volatility framework is very easy to install and run; simply unpack it onto a system that has Python installed on it (version 2.5 or later) and run the command '**python volatility**'. The following image (Figure 7) is a screenshot of **volatility** running.


Shell - Konsole

```
bt Volatility-1.1.2 # pyth
python              python-config       python2.5           python2.5-config
bt Volatility-1.1.2 # python volatility pslist -f xp-laptop-2005-07-04-1430.img
Name                 Pid     PPid    Thds    Hnds    Time
System               4       0       62      1133    Thu Jan 01 00:00:00 1970
smss.exe             400     4       3       21      Mon Jul 04 18:17:26 2005
csrss.exe            456     400     11      551     Mon Jul 04 18:17:29 2005
winlogon.exe         480     400     18      522     Mon Jul 04 18:17:29 2005
services.exe         524     480     17      321     Mon Jul 04 18:17:30 2005
lsass.exe            536     480     20      369     Mon Jul 04 18:17:30 2005
svchost.exe          680     524     19      206     Mon Jul 04 18:17:31 2005
svchost.exe          760     524     10      289     Mon Jul 04 18:17:31 2005
svchost.exe          800     524     75      1558    Mon Jul 04 18:17:31 2005
Smc.exe              840     524     22      421     Mon Jul 04 18:17:32 2005
svchost.exe          932     524     6       93      Mon Jul 04 18:17:33 2005
svchost.exe          972     524     15      212     Mon Jul 04 18:17:34 2005
spoolsv.exe          1104    524     11      145     Mon Jul 04 18:17:38 2005
ati2evxx.exe         1272    524     4       38      Mon Jul 04 18:17:39 2005
Crypserv.exe         1356    524     3       34      Mon Jul 04 18:17:40 2005
DefWatch.exe         1380    524     3       27      Mon Jul 04 18:17:40 2005
msdtc.exe            1440    524     15      164     Mon Jul 04 18:17:40 2005
Rtvscan.exe          1484    524     37      312     Mon Jul 04 18:17:40 2005
tcpsvcs.exe          1548    524     2       105     Mon Jul 04 18:17:41 2005
snmp.exe             1564    524     5       192     Mon Jul 04 18:17:41 2005
svchost.exe          1588    524     5       122     Mon Jul 04 18:17:41 2005
wdfmgr.exe           1640    524     4       65      Mon Jul 04 18:17:42 2005
Fast.exe             1844    524     2       33      Mon Jul 04 18:17:43 2005
mqsvc.exe            1860    524     23      218     Mon Jul 04 18:17:43 2005
mqtgsvc.exe          712     524     9       119     Mon Jul 04 18:17:47 2005
alg.exe              992     524     5       105     Mon Jul 04 18:17:50 2005
ssonsvr.exe          2196    2172    1       24      Mon Jul 04 18:17:59 2005
explorer.exe         2392    2300    18      489     Mon Jul 04 18:18:03 2005
Directcd.exe         2456    2392    4       40      Mon Jul 04 18:18:05 2005
TaskSwitch.exe       2472    2392    1       24      Mon Jul 04 18:18:05 2005
Fast.exe             2480    2392    1       23      Mon Jul 04 18:18:05 2005
VPTray.exe           2496    2392    2       111     Mon Jul 04 18:18:06 2005
atiptaxx.exe         2524    2392    1       51      Mon Jul 04 18:18:06 2005
jusched.exe          2548    2392    1       22      Mon Jul 04 18:18:07 2005
EM_EXEC.EXE          2588    2540    2       80      Mon Jul 04 18:18:09 2005
WZQKPICK.EXE         2692    2392    1       17      Mon Jul 04 18:18:15 2005
wuauclt.exe          3128    800     3       157     Mon Jul 04 18:19:11 2005
taskmgr.exe          3192    2392    3       65      Mon Jul 04 18:19:33 2005
cmd.exe              3256    2392    1       29      Mon Jul 04 18:20:58 2005
firefox.exe          3276    2392    7       189     Mon Jul 04 18:21:11 2005
PluckSvr.exe         3352    680     6       206     Mon Jul 04 18:21:42 2005
PluckTray.exe        3612    3352    3       102     Mon Jul 04 18:24:00 2005
PluckUpdater.ex      368     3352    0       -1      Mon Jul 04 18:24:30 2005
dd.exe               3300    3256    1       22      Mon Jul 04 18:30:32 2005
bt Volatility-1.1.2 #
```

Figure 7: A screen capture of running the volatility framework on a Windows XP memory capture.

All that is needed to run **volatility** is a memory dump image, which can be obtained using many different tools, both open source and commercial. In the above screenshot, the memory dump was obtained using **memdump** (described in section 6.2) (Volatile Systems, 2008).

6.8 VAD Tools

As discussed in section 5.8, Virtual Address Descriptor structures contain much valuable information about processes and the structures they have allocated (such as mapped files). VAD Tools (Dolan-Gavitt, 2007) are a set of scripts written in python that are able to parse through this information for things that are of interest to a forensic investigator.

There are five scripts available: **vadwalk.py**, **vadinfo.py**, **vaddump.py**, **procdump.py**, and **listdll.py**. The **vadwalk.py** script traverses the VAD tree and prints it out as a table or an ASCII tree; it can also create a GraphViz file that can be loaded into a visualizer to display the tree. The **vadinfo.py** script, as the name implies, spells out detailed information from the VAD, including files mapped into the process's address space such as DLLs. The **vaddump.py** and **procdump.py** scripts both extract memory regions from the VAD tree – **procdump.py** is specific to executables such as **.dll** files and **.exe** files that are stored in the memory image. Finally, **listdll.py** prints out a list of all of the modules (DLLs) that are loaded for a specific process.

Figure 8 shows an example of the output from **vadwalk.py** generated by Andreas Schuster (Schuster, 2007).

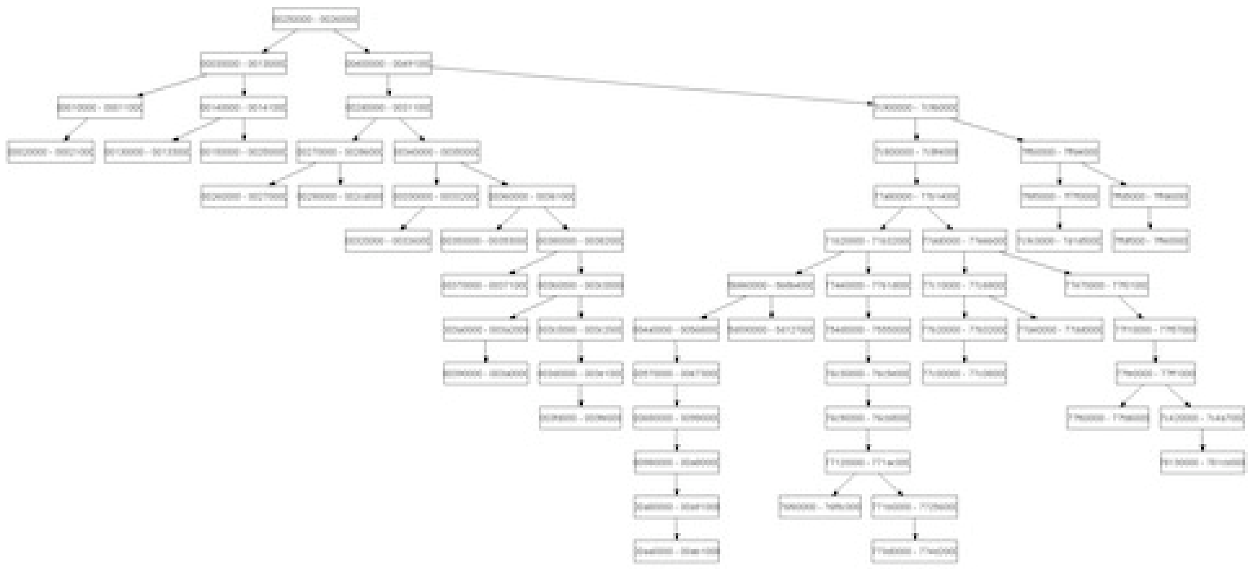


Figure 8: A reconstruction of the VAD tree using the script vadwalk.py (Schuster, 2007).

The VAD Tools are advanced and low-level, but extremely useful for parsing through data structures in memory and extracting information. In particular, the ability to reconstruct an **.exe** file or a **.dll** file is a useful capability.

6.9 Commercially Available Tools

There are many commercially available tools that perform much of the functionality described in section 5, and can be used instead of the free, open-source tools described earlier in this section. In this section a few products that are particularly well-known are briefly described.

Encase Enterprise (Guidance Software, 2008), one of the most widely-used forensic tools, has a “Snapshot” utility that will capture the volatile data that exists in RAM on certain types of systems, and will parse the data so that it is organized and more meaningful to the analyst. Currently Encase does not provide the analyst with a raw capture of the data, so analysts must rely on the software to correctly interpret the memory and find everything of interest.

F-Response is another tool that allows remote, read-only access to the physical memory of a machine of interest to the analyst. It does not actually capture the data (an analyst would need another program to perform the capture).

HBGary Responder (HBGary, 2009) is a very powerful tool that performs memory analysis, and will reverse-engineer malware that is pulled out of memory. The malware analysis capability is of particular interest – it allows a suspicious executable found in memory to be extracted, disassembled, and scanned for suspect code and functionality. HBGary also has a software utility called FastDump that is available for free and can be used to capture physical memory.

These commercial tools are useful, and in some cases their functionality is easier to leverage than open source tools, or they have capabilities that open source tools lack. Still, it is important to note that it is not necessary to have expensive commercial tools to perform a thorough and effective analysis of the volatile memory.

7. Cautions and Considerations

One concern with performing memory analysis is that the act of acquiring memory can cause changes to the system being analyzed. In particular, running a program to collect information from volatile memory often starts up a process on the machine in question, which of course adds new data to the volatile memory. This data could potentially overwrite critical data related to the investigation.

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.

It is also important to understand that at any given moment the state of a system's volatile memory is not reproducible. It essentially would be an impossible task to get the machine's memory to look exactly the same, even if all the same actions were performed after a system was booted up, due to the fact that the machine may load programs into different areas of memory.

Another concern is whether you can trust the operating system to tell the truth about what is actually in memory. An attacker could easily trojanize the commands that capture volatile memory and instead of recording an accurate representation of the data stored in RAM, the command might instead omit important details that the attacker does not want an analyst to see, or even provide a completely false set of data. Analysts must be careful to use their own tools to capture memory, rather than relying on those that the attacker has had access to.

Even more worrisome is that an advanced attacker might alter the way the operating system itself works to hide data from the analyst – as an example, the operating system stores the processes that are currently running in a linked list. In order to hide a process that the attacker has started and wants to keep an analyst from seeing, the attacker could change the pointers in the linked list so that that particular process is omitted each time the operating system traverses the project list.

All of these issues need to be taken into consideration when using volatile memory in a forensic analysis. None of them make the analysis less valuable; they are simply facts that need to be in the analysts arsenal so that he or she will know what to do when faced with such a situation.

8. Conclusion

Forensic analysis on volatile memory is by no means perfected. Researchers and analysts are coming up with new and improved techniques for recovering data from memory on a regular basis, and there is still much work that needs to be done in this field. As technology continues to develop – and as malicious users come up with new ways to attack and exploit that technology – the need for forensic analysis capabilities that encompass the physical memory will only increase.

In this paper, we discussed why the ability to perform forensic analysis on volatile memory is a valuable asset to a forensic analyst. In particular, this paper outlined what types of information can potentially be recovered from volatile memory, and how that information might relate to an investigation. Current techniques for recovering this data from memory were also discussed – when possible, techniques applicable to both Windows and Unix systems were described. Finally, a subset of the current tools available to analysts who wish to perform analysis on volatile memory was described. A brief discussion of what to be careful of and what to keep in mind when analyzing volatile memory was also provided.

Many researchers believe that volatile memory is becoming the location of choice for attackers and other malicious users to store information they do not want found, or execute harmful code that they do not want reverse-engineered. There is ample evidence to support this theory; it is convincing simply because the majority of incident response teams do not collect volatile memory, and the majority of analysts are not equipped to decipher relevant information from it. In addition, several recent pieces of malware have been written to exist solely in memory and never touch the hard drive, effectively evading any analysis that only looks at the physical disk. It is logical to assume that malicious users will continue to move toward using physical memory instead of the hard disks, and analysts will need to adapt to that movement.

The aim of this paper was to make an argument for adding analysis of physical memory to the toolkit of the forensic analyst, and to make that goal accessible by describing some of the techniques and tools used by analysts who are already looking at volatile memory. Armed with this knowledge, analysts who are looking to expand their skill set and become more effective and proficient at their work can begin to explore this exciting branch of digital forensics.

9. References

Bovet, D., & Cesati, M. (2006). *Understanding the Linux Kernel* (3 ed.). Sebastopol, CA: O'Reilly Media, Inc.

Burdach, M. (2005, July 9). *An Introduction to Windows memory forensic.* Retrieved October 25, 2008, from http://forensic.seccure.net/

Burdach, M. (2005, March). *Digital Forensics of the Physical Memory.* Retrieved January 4, 2009, from Forensic Focus: http://www.forensicfocus.com/

Burdach, M. (2006). *DigitalInvestigation*. Retrieved from http://forensic.seccure.net

Carrier, B. (2005). *File System Forensic Analysis.* New York, New York: Addison-Wesley.

Carrier, B., & Grand, J. (2004, March). A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Journal of Digital Investigations* .

Dolan-Gavitt, B. (2007). VADTools. SourceForge.

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering.* Indianapolis, Indiana: Wiley Publishing, Inc.

Farmer, D., & Venema, W. (1999, August). Memdump.

Garfinkel, T., Pfaff, B., Chow, J., & Rosenblum, M. (2004). Data lifetime is a systems problem. *ACM SIGOPS European Workshop* (p. 10). New York: ACM.

GMG Systems, Inc. (2007). KnTTools.

Guidance Software. (2008). *Computer Forensics*. Retrieved March 3, 2009, from http://www.guidancesoftware.com/

HBGary. (2009). *Responder Professional*. Retrieved January 15, 2009, from HBGary: http://www.hbgary.com/responder_pro.html

Microsoft Corporation. (2005, August). Windows Feature allows a Memory.dmp file to be generated with the keyboard. Redmond, Washington, USA: Microsoft Corporation.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., & Weaver, N. (2003). Inside the Slammer Worm. *IEEE Security and Privacy. 1*, pp. 33-39. Piscataway: IEEE Educational Activities Department.

Petroni, N. L., Walters, A., Fraser, T., & Arbaugh, W. A. (2006, December). FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory. *Digital Investigation , 3* (4), pp. 197-210.

Russinovich, M. E., & Solomon, D. A. (2005). *Windows Internals* (4th ed.). Redmond, Washington: Microsoft Press.

Schuster, A. (2007, May 19). *Memory Analysis: Walking the VAD Tree*. Retrieved December 2008, from Computer Forensic Blog:

http://computer.forensikblog.de/en/2007/05/walking_the_vad_tree.html

Schuster, A. (2007, November 27). *PTFinder Version 0.3.05*. Retrieved January 2009, from Computer Forensic Blog: http://computer.forensikblog.de/en/2007/11/ptfinder_0_3_05.html

Schuster, A. (2006). Searching for processes and threads in Microsoft Windows memory dumps. *The Proceedings of the 6th Annual Digital Forensic Research Workshop. 3*, pp. 10-16. Digital Investigation.

Sparks, S., & Butler, J. (2005). *Shadow Walker: Raising The Bar For Windows Rootkit Detection*. Retrieved December 2008, from Phrack: http://www.phrack.org/issues.html?issue=63&id=8#article

Stevens, D. (2007, January 30). XORSearch.

Szor, P. (2005). *Virus Research and Defense.* New York, New York: Addison-Wesley.

van Baar, R., Alink, W., & van Ballegooij, A. (2008). Forensic Memory Analysis: Files Mapped in Memory. *Digital Forensic Research Workshop. 5*, pp. 52-57. Elsevier Ltd.

Volatile Systems. (2008). *FATKit: The Forensic Analysis ToolKit*. Retrieved December 2008, from http://4tphi.net/fatkit/

Volatile Systems. (2008). *The Volatility Framework*. Retrieved November 2008, from

Volatile Systems: https://www.volatilesystems.com/

Last Updated: November 23rd, 2019

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Bangalore 2019	Bangalore, IN	Nov 25, 2019 - Nov 30, 2019	Live Event
SANS Cyber Threat Summit 2019	London, GB	Nov 25, 2019 - Nov 26, 2019	Live Event
SANS Nashville 2019	Nashville, TNUS	Dec 02, 2019 - Dec 07, 2019	Live Event
SANS San Francisco Winter 2019	San Francisco, CAUS	Dec 02, 2019 - Dec 07, 2019	Live Event
SANS Security Operations London 2019	London, GB	Dec 02, 2019 - Dec 07, 2019	Live Event
SANS Paris December 2019	Paris, FR	Dec 02, 2019 - Dec 07, 2019	Live Event
SANS Frankfurt December 2019	Frankfurt, DE	Dec 09, 2019 - Dec 14, 2019	Live Event
SANS Cyber Defense Initiative 2019	Washington, DCUS	Dec 10, 2019 - Dec 17, 2019	Live Event
SANS Austin Winter 2020	Austin, TXUS	Jan 06, 2020 - Jan 11, 2020	Live Event
SANS Threat Hunting & IR Europe Summit & Training 2020	London, GB	Jan 13, 2020 - Jan 19, 2020	Live Event
SANS Miami 2020	Miami, FLUS	Jan 13, 2020 - Jan 18, 2020	Live Event
Cyber Threat Intelligence Summit & Training 2020	Arlington, VAUS	Jan 20, 2020 - Jan 27, 2020	Live Event
SANS Tokyo January 2020	Tokyo, JP	Jan 20, 2020 - Jan 25, 2020	Live Event
SANS Amsterdam January 2020	Amsterdam, NL	Jan 20, 2020 - Jan 25, 2020	Live Event
SANS Anaheim 2020	Anaheim, CAUS	Jan 20, 2020 - Jan 25, 2020	Live Event
MGT521 Beta Two 2020	San Diego, CAUS	Jan 22, 2020 - Jan 23, 2020	Live Event
SANS Las Vegas 2020	Las Vegas, NVUS	Jan 27, 2020 - Feb 01, 2020	Live Event
SANS Vienna January 2020	Vienna, AT	Jan 27, 2020 - Feb 01, 2020	Live Event
SANS San Francisco East Bay 2020	Emeryville, CAUS	Jan 27, 2020 - Feb 01, 2020	Live Event
SANS Security East 2020	New Orleans, LAUS	Feb 01, 2020 - Feb 08, 2020	Live Event
SANS London February 2020	London, GB	Feb 10, 2020 - Feb 15, 2020	Live Event
SANS Northern VA - Fairfax 2020	Fairfax, VAUS	Feb 10, 2020 - Feb 15, 2020	Live Event
SANS New York City Winter 2020	New York City, NYUS	Feb 10, 2020 - Feb 15, 2020	Live Event
SANS Dubai February 2020	Dubai, AE	Feb 15, 2020 - Feb 20, 2020	Live Event
SANS Brussels February 2020	Brussels, BE	Feb 17, 2020 - Feb 22, 2020	Live Event
SANS Scottsdale 2020	Scottsdale, AZUS	Feb 17, 2020 - Feb 22, 2020	Live Event
SANS San Diego 2020	San Diego, CAUS	Feb 17, 2020 - Feb 22, 2020	Live Event
Open-Source Intelligence Summit & Training 2020	Alexandria, VAUS	Feb 18, 2020 - Feb 24, 2020	Live Event
SANS Tokyo November 2019	OnlineJP	Nov 25, 2019 - Nov 30, 2019	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced