Physical Memory Forensics for Files and Cache

James Butler and Justin Murdock

MANDIANT Corporation
james.butler@mandiant.com

Rochester Institute of Technology
jdm8744@rit.edu

ABSTRACT

Physical memory forensics has gained a lot of traction over the past five or six years. While it will never eliminate the need for disk forensics, memory analysis has proven its efficacy during incident response and more traditional forensic investigations. Since attackers typically live in userland memory, reconstructing the processes in that space is essential to an investigation. While previous reconstruction techniques in this area have shown promise, they have left room for improvement. This paper will focus on file reconstruction as a method of reducing unknown data in memory and accelerating forensic analysis.

KEYWORDS

memory mapped file, process reconstitution, physical memory analysis, file extraction, MemD5, *ImageSectionObject, DataSectionObject, SharedCacheMap*

I. Introduction

When performing memory analysis, there are two primary components: a) kernel memory and b) userland memory. In a Windows environment, kernel memory is comprised primarily of device drivers, the NT operating system executable, and HAL.DLL. The majority of userland memory is comprised of individual processes, and the process' address space primarily consists of files loaded from disk that contain code or data needed for process execution, typically Portable Executable files (EXEs) and Dynamic Link Libraries (DLLs). Most of the virtual addresses in kernel memory are global regardless of the process context, whereas the other virtual address space is generally specific to each particular process. This has interesting implications when performing memory acquisition, a topic discussed in later sections. This paper focuses on userland memory in the context of an individual process' address space. Normally, attackers target the victims' userland memory because it is easier to write code and get code loaded in userland memory as opposed to kernel memory.

There have been promising advancements in the field of memory forensics in the area of userland or process reconstitution, but they can be improved upon. Previous tools focused mainly on stepping through the Virtual Address Descriptors (VADs) or virtual memory regions that

comprised a process' address space, but ignored other rich sources of information such as those present for memory mapped files. Since these tools use limited information about the VAD structure to extract files from memory, the resulting files are not accurate representations of their complete contents on disk. The operating system's cache for Input/Output (I/O) has also been largely ignored. This paper introduces techniques to gather information and extract files from memory with much higher precision.

II. Current Physical Memory Forensics Techniques

The two most common (and free) memory forensic tools are Volatility [1] and Memoryze [2]. Both of these tools have commands to analyze the contents of a process. Volatility's commands include vaddump, dlldump, procmemdump, procexedump, and memdump. Memoryze uses the ProcessDD command. This section compares and contrasts the commands in both tools to evaluate the state of process reconstitution.

Most memory analysis tools, Volatility and Memoryze included, use a technique known as VAD walking to find data loaded into memory. VADs are data structures linked to each process' EPROCESS block that the memory manager uses to keep track of which virtual addresses have been reserved for each process [3:448]. These structures are available in memory and can be used to determine information about every running process. Stepping through these memory structures is known as VAD walking [4]. ProcessDD in Memoryze and vaddump [1] in Volatility operate by VAD walking. The four minor differences between these commands are as follows:

1. Memoryze does not acquire the first MB of the process's virtual address space (0x00000000-0x000fffff).
2. Vaddump does not name any of the memory regions if it is a named memory region.
3. Memoryze replaces pages not in the process' address space with NOPs (0x90s) while Vaddump uses zeroes (0x0).
4. Vaddump does not report regions with a name and size of zero.

Dlldump in Volatility has a subset of the information contained in ProcessDD; however, both tools use different methods to achieve nearly the same results. Dlldump walks the linked list of loaded modules (DLLs) in the Process Environment Block (PEB) pointed to by the process' EPROCESS block, and ProcessDD walks the VAD applying the DLL names if they are present. The linked list of loaded modules exists in userland memory and could be altered by the attacker so dlldump gives the user the option of specifying the base physical address of the unlinked module in order to acquire it. A similar attack could be performed against a process' VAD table which would hide the memory region from Memoryze; however, this requires the attacker to be fairly skilled and executing his/her code in the kernel address space where VADs are stored. Dlldump also parses the Portable Executable (PE) format to determine the size of the DLL, whereas Memoryze's ProcessDD utilizes the size contained in the VAD structure, which is page aligned. Since dlldump utilizes the information in the PE header to determine the size of each

section, it will more accurately represent how the file would appear on disk. ProcessDD encounters the data as the Windows PE loader placed it in memory. However, both tools return exactly the same physical contents ignoring the alignment issues.

Comparing the output of procmemdump with ProcessDD yields almost identical binary applications. Procmemdump includes the "slack space" of the binary according to the command reference. Since the slack space is included, there are no alignment issues between the outputs of the two commands. Procexedump is almost identical to procmemdump, but it does not include the slack space in the PE; therefore, it has the same alignment issues as dlldump when comparing it to the output of ProcessDD.

Memdump is unique in the fact that there is no corresponding output created by Memoryze. A comparison of the output of memdump, which dumps all addressable memory in a process' address space, to the output of ProcessDD or vaddump reveals a large discrepancy in the size of the acquired data. In an acquisition of CSRSS.EXE from a Windows 7 SP0 32-bit host, the difference was in the hundreds of MBs. The extra data is a result of the technique memdump uses to acquire a process' address space. Memdump brute forces address translation across the virtual memory range of 0 to 4 GB. As mentioned before, kernel addresses are global for the most part across all processes. Since kernel memory typically starts at virtual address 0x80000000 on 32-bit systems, memdump acquires all the global kernel memory pages that translate in the context of the process specified.

All of these techniques can be generalized into one method:

1. Start with a virtual base address and a size.
    a. Parsing the PE header of the first page to determine the virtual addresses of the subsequent pages is a derivation of this technique.
    b. Brute force translation across all possible virtual addresses is also a derivation where the base address is 0 and the size is 4GB.
2. Translate every address to the corresponding physical page.

The following section focuses on the challenges with this method and section IV describes improvements.

III. Issues with Traditional Approaches

While previous approaches to memory forensics and process reconstitution have made tremendous advances and given analysts access to data previously unavailable, there is room for improvement. Translating virtual addresses across a range of memory ignores the structures the Windows operating system uses to represent files, FileObjects. While FileObjects are not present for all memory ranges, they do exist for memory mapped files such as DLLs and EXEs. Since DLLs and EXEs are paramount to most investigations, the traditional approaches are ignoring portions of the process' address space. The second issue deals with attribution. If a tool performs

a brute force translation of every address in the range of a process' memory, it will undoubtedly attribute certain kernel pages as belonging to the process, when in reality they do not.

a. Missing Data

DLLs and EXEs are called memory mapped files because their data in memory is backed by their original data on disk. For memory mapped files, code is not paged to the paging file(s). Memory mapped files are represented by kernel objects, which will be described in the next section, and are backed by themselves on disk when the contents need to be paged to disk. Since the code does not change under normal operation, it would be redundant to page out data to the paging file(s) because it already exists on disk in the original file. Only the private data of a binary specific to an individual process is paged to the paging file(s)

As part of our investigations, we performed an experiment by acquiring the AcroRd32.exe (Acrobat Reader) process in the memory image from the Honeynet Project Challenge 3 [5] using two separate approaches. The traditional approach used Volatility 2.0 and the command dlldump. The second method acquired the file by using the FileObject technique described in section IV. Table 1 shows the advantage of applying knowledge of the structures pointed to by the FileObject instead of translating across the entire virtual address range of the VAD representing the file.

AcroRd32.exe PID 1752	File Size in Bytes	Bytes Acquired with Traditional Approach (Ignoring FILE_OBJECT)	Bytes Acquired using FILE_OBJECT
Ntdll.dll	708,096	295,936	390,656
Wininet.dll	656,384	309,248	411,648
User32.dll	577,024	234,496	409,600
Crypt32.dll	597,504	213,504	367,104
Ace.dll	565,248	394,752	524,288

Table 1: Bytes acquired using different methods

Table 1 demonstrates that using the FileObject resulted in substantially better results in this sample of DLLs. Since the challenge did not include the underlying file system, pages that could not be read because they were not in memory were identified by comparing a 512 byte sector of 0x00 bytes in one file corresponding to the same sector of 0x90 bytes in the other acquired file. We will further explore the FileObject and the technique used for acquisition in later sections of this document.

b. Misattributed Data

During an incident response investigation, attribution is important. Although it is interesting initially to know that the host is compromised, more context is required to properly remediate.

Since this research focuses on processes' address spaces, knowing which process is compromised is important.

In the Honeynet Project Challenge 3 solution, Smith, Et al. [6] acquired PID 1752 (AcroRd32.exe) using Volatility's memdump command. Foremost [7] was run against the output to carve out PDFs. Foremost looks for a specified beginning and ending delimiter to carve files. In this example, it created seven PDFs. Five were just over four hundred bytes and determined uninteresting. The final two files were:

1. 00599696.pdf which was 60,124 bytes in size
2. 00600328.pdf which was 607,083 bytes in size

The authors found the suspicious content in 00600328.pdf, which they attributed to the AcroRd32.exe process since it is the process they acquired. However, using Volatility's memdump command and Foremost in exactly the same series of steps on the same image on PID 1108 (VMwareTray.exe) produced the same two file sizes with exactly the same contents. Although their conclusions were correct, if they tried to acquire a process other than AcroRd32.exe, the results would have been incorrect and misleading due to a lack of correct attribution.

The reason both of these files "exist" in each process is because of the technique used to acquire the processes. Memdump brute forces the 4 GB virtual address space writing out successfully translated pages when using the target process' Directory Table Base (DTB). The contents of both of these files must have translated in each process' address space. By placing a print statement in the Volatility script, it was determined that these files are in the operating system's cache at 0xd2dc0000. The operating system uses the cache to keep portions of files and other I/O in memory to avoid the performance hit of constantly reading from and writing to the disk. Since the cache's virtual address range is in kernel memory and global, it translates correctly in both processes.

Because memdump does not write to disk pages that do not translate, addresses that translated appear contiguous in the acquisition when there may be a several hundred virtual page gap. Foremost just looks through the data for beginning and ending delimiters. Using MoonSols bin2dmp.exe [8] tool, you can transform the memory image from the challenge (Bob.vmem) to Windbg format. By loading the dump into Windbg and using the search and !filecache commands, we found that the FILE_OBJECT is at 0x81dfadf0 and present in the cache. However, !filecache reveals that the file 00600328.pdf is actually called PDF.php. Instead of being 607,083 bytes in size, it is actually only 19,731 bytes in size. This discrepancy exists because memdump wrote all the pages of the process that translated as a contiguous file and ignored the pages that failed to translate and the cache structures. Also, since Foremost looks for a beginning delimiter and a corresponding ending delimiter, it treats the content in between as

one contiguous file. Windbg's !filecache command states that only 20K is reserved in the cache for this file. The next section describes how to utilize the FileObject and cache structures.

IV. Utilizing File Objects

Since DLLs and EXEs are memory mapped files represented in kernel memory by FileObjects, understanding their structure is very important to memory forensics. The FILE_OBJECT structure contains several important members. It contains a pointer to the Unicode name of the file and a pointer to the file's DeviceObject, which contains the drive name such as HarddiskVolume1. One of the most important pointers in the FileObject is to a structure called SECTION_OBJECT_POINTERS, as shown in Figure 1.

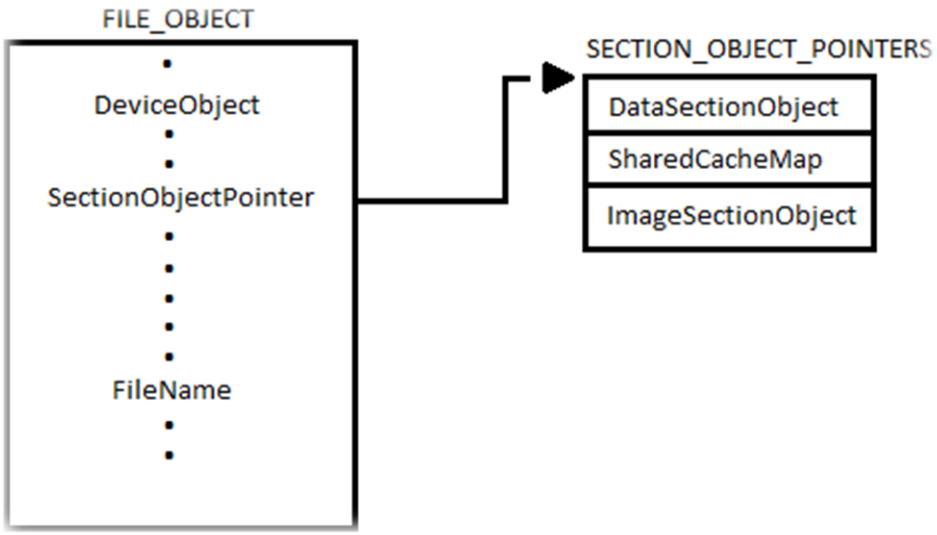


Figure 1: FILE_OBJECT structure with relevant pointers

The SECTION_OBJECT_POINTERS structure only contains three pointers as seen in Figure 1. The first is called the *DataSectionObject*. The next is called the *SharedCacheMap*, and the final pointer is called the *ImageSectionObject*. *DataSectionObject*s and *ImageSectionObject*s are related and are actually pointers to CONTROL_AREAs. *ImageSectionObject*s represent binaries also called images. *DataSectionObject*s can point to structures used to maintain data files such as those used by Microsoft Word. The *SharedCacheMap* is a pointer to the SHARED_CACHE_MAP structure, which is used by the operating system to maintain the cache. The following sections will investigate each of these structures.

a. Image Section Objects

The *ImageSectionObject* is a pointer to a CONTROL_AREA structure and used to represent memory mapped binaries. As shown in Figure 2, the ControlArea contains a pointer to a SEGMENT structure. The Segment contains a pointer back to the ControlArea. This relationship

can be used as a validation method when parsing memory. The Segment also contains the size of the Segment (SizeOfSegment) and the total number of Prototype Page Table Entries (PTEs). The total number of PTEs (TotalNumberOfPtes) can be multiplied by 0x1000 and compared to the Segment size for an additional sanity check. When parsing ControlAreas, the SUBSECTION structures are very important. The easiest way to find the first Subsection related to the FileObject is to add the size of the CONTROL_AREA structure to its address in memory. This varies between different versions of the operating system. Work has been done to automate the detection of the operating system's version and apply the appropriate offsets, which could lead to more portable code [9]. Our research has shown that the first Subsection is immediately following the ControlArea pointed to by the *ImageSectionObject*, and each Subsection contains a pointer to the next Subsection. Figure 2 illustrates the relationship between these objects according to the Windbg command *!ca*.

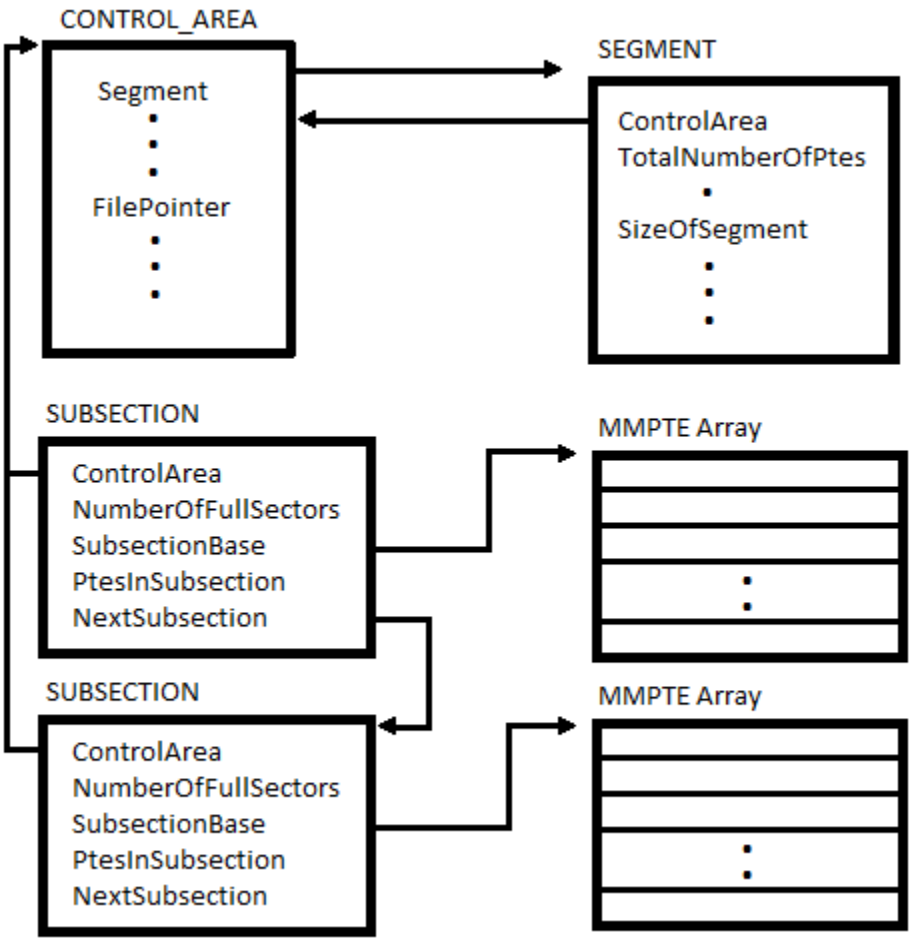


Figure 2: The path from a CONTROL_AREA to physical memory.

The Subsection contains a pointer back to the ControlArea, which also can be used for validation, but the most interesting pointer is the one to the array of Prototype PTEs (MMPTE Array). The Prototype PTE contains the physical address of the page in memory. If instead of a physical address, the Prototype PTE has a pointer back to the Subsection that page is paged out to the original file on disk.

The NumberOfFullSectors and PtesInSubsection in the Subsection are vital when iterating through the array of Prototype PTEs. Every Prototype PTE represents a 4096 byte page; however, each full sector represents a 512 byte sector. This requires certain considerations. Since every Prototype PTE represents 4096 bytes, there can be up to 8 sectors in each Prototype PTE. Getting the alignment correct is crucial in order to match what is in memory with what is on disk. By walking through the Prototype PTEs and incrementing a variable for each sector parsed, it is easy to access the correct sector in the file on the disk if the Prototype PTE is actually a pointer back to the Subsection.

b. Data Section Objects

*DataSectionObject*s are similar to *ImageSectionObject*s. They both point to ControlAreas. Some of the Segment's values will not be initialized correctly, but the algorithm used in section IV.a will work. However, in our research the NumberOfFullSectors and the number of PtesInSubsection were always the same. Therefore, iterating through the Prototype PTE array by the number of PTEs in the Subsection is sufficient.

c. Shared Cache Map

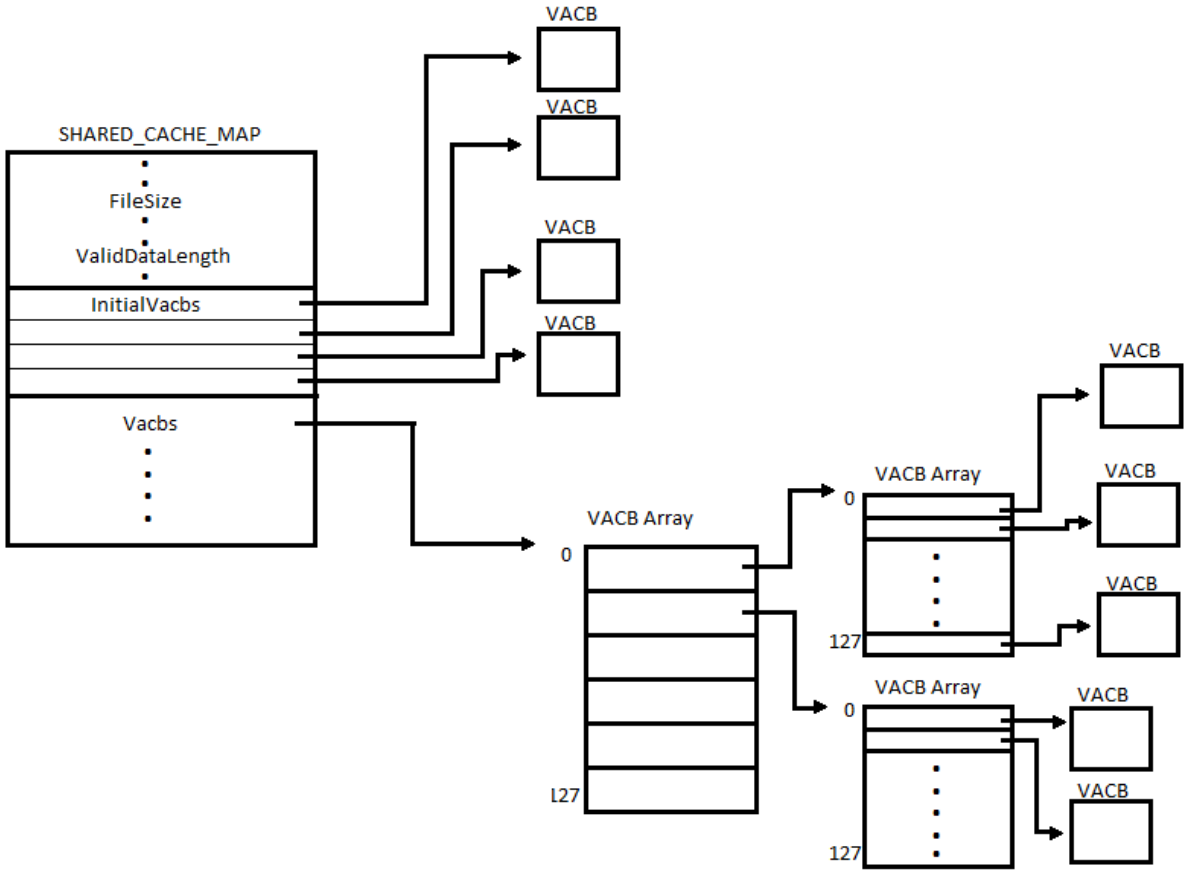
The *SharedCacheMap* points to the SHARED_CACHE_MAP structure, represented by Figure 3. It contains the file size (FileSize) and the length of valid data in the cache (ValidDataLength). If the ValidDataLength is ever greater than the FileSize, the SHARED_CACHE_MAP is invalid. This structure also contains an element called InitialVacbs, which is actually an array of four pointers to VACBs (Virtual Address Control Blocks). Each VACB represents 256 KB of cache. If the file size is less than or equal to 1 MB, then the FileObject can be represented by four VACBs called the InitialVacbs. Files larger than 1 MB must use the Vacbs pointer. The Vacbs points to an array of VACB structures. If the file size is 32 MB or less, it can be represented by a single array; otherwise, a VACB Array can point to arrays of VACB Arrays. Previous work by Hejazi Et al. [10] uses information from the SharedCacheMap, and our technique extends their work by taking into account the multiple layers found in files larger than 32 MB. The depth of this tree can be calculated as

$$(\lceil log_2(n) \rceil - 18) / 7$$

where n is the file size [3:672]. The result must be rounded up to the next whole number. The individual VACB structures are rather simple. They have a pointer to a 256 KB block of the cache and pointer back to the SHARED_CACHE_MAP structure. The other elements are not

relevant to this discussion.

Figure 3: File cache structures in memory.



V. Applications

FileObjects and their associated structures are pertinent for full process reconstitution. In order to reach the rich data available in memory mapped files, data files, and files in cache finding all the FileObjects associated with the process is imperative. When acquiring or analyzing a process' address space, FileObjects are found in two separate places: a) the process' handle table and b) the VAD tree. Previous work has focused mainly on the VAD tree; however, some important data to an investigation is found in the handle table. This section describes the usefulness of both.

a. Windows Registry Files

In the Windows operating system, the System process contains handles to the Registry hives. Large portions of the hives are present in the cache. By parsing the handle table and acquiring the System process, it is possible to use tools such as RegRipper [11] to analyze the Registry

hives. The approach used by Dolan-Gavitt [12] is to scan memory for evidence of a Registry hive. This has the advantage that it finds the volatile hives that are not present on disk. These are known as the HARDWARE and the REGISTRY hives. The advantage of using the handle table is that the entire memory space does not need to be scanned and non-legitimate hives are never found. Both techniques effectively are parsing the cache when analyzing the contents of the Registry.

b. PDFs and Other Files

Some files such as PDFs are also contained in the cache or portions of them may be. The cache is a less reliable source as opposed to *DataSectionObject*s and *ImageSectionObject*s because the operating system manages what pages are present in the cache and which ones are not. More research is needed in this area to determine what other useful files may exist in cache. Tangentially, our research has observed GIFs from a Web browsing session in the cache. Microsoft Word files are not contained in the cache [13], but can be acquired using the *DataSectionObject*.

c. Hashes

Although hashes fail when the adversary is constantly evolving, there is a long history of its use in computer security. In memory analysis, hashing can be useful even if the attacker is changing rapidly because known hashes can be eliminated from the address space and used as a filter. However, the problem has always been that the entire file could not be analyzed so the hash lists built up historically were useless. Other research has used fuzzy hashing or subsets of the file to hash against [14], [15], but again the historical data is too limited to make a comparison optimal. On the other hand, by parsing the *ImageSectionObject* of a FileObject, it is possible to calculate the same hash from memory as the one on disk. We have implemented this approach to produce the MemD5 since it is identical to the MD5 on disk. This requires a live analysis of memory because of the necessity to read paged out sectors of the file from disk. If the hash from memory matches a known good hash, the memory region can be ignored, therefore, greatly reducing the search space.

d. Byte Signatures

Classification tools such as VxClass, formerly from Zynamics, and other automated malware analysis tools have the ability to generate a series of bytes present in a cluster of related malware samples. Although these byte signatures could be labeled as traditional anti-virus signatures, they have been specifically engineered to be based on code sections that are common across all the related samples. Therefore, the byte signatures have better longevity. Although using byte signatures against an attacker that does not reuse any code will not be successful, it has proven

effective in our research against several related families of malware in our collection. Operational use of byte signatures must leverage the *ImageSectionObject* of the FileObject for binaries in memory as illustrated by Table 1. If the analysis is run on a live system, the ability to recover the entire file is possible as mentioned in section V.c; hence, all the content can be searched and false negatives will not be reported.

VI. Further Work

Although our research has made progress and its implementation in Memoryze 2.0 looks promising, there is still more work to be done. While we were evaluating the implementation, several issues were identified.

a. Caching Across Different Platforms

On Windows 7 and Windows 2008, the caching behavior appeared differently during our research, but the issues were outside the scope of this paper. From our limited observations, the Registry hives did not appear to be cached. In fact, Windbg reported that there were no active system cache mappings for our virtual instance of Windows 7 32-bit and 64-bit, but we could find evidence of the Registry in physical memory.

b. Address Space Layout Randomization

Platforms with address space layout randomization (ASLR) enabled will influence the hash calculation and the byte signature matching to a lesser extent because the Windows loader modified the binary in memory so that it no longer matches its contents on disk. These changes are a result of rebasing and should be reversible by comparing the load address of the binary in memory with the preferred load address on disk and utilizing the relocation section to reverse the steps the Windows loader took.

c. The Security Directory

In the PE format, the Security Directory of a binary contains the digital certificate information for the file. Although this data is not necessary after the file is loaded, many versions of the Windows loader create artifacts in the SEGMENT and SUBSECTION structures to represent this area of the file, but our research has shown that this is not the case on Windows 7. Because of its similarity to Windows 7, Windows 2008 is likely to exhibit the same behavior. Going forward, we will have to detect whether or not a Security Directory exists in the PE header in memory. If one exists, it will have to be parsed from disk and fed into the MemD5 hashing function.

VII. Conclusion

In this paper we have introduced new techniques to accurately reconstruct files and processes from memory. Building on previous work, we have shown the importance of the SECTION_OBJECT_POINTERS structure in FileObjects. Specifically, when extracting a process from memory, the *ImageSectionObject* acts as an invaluable resource of information about the location of the process contents in memory. *DataSectionObjects* show promise because they can be used to extract data files such as Microsoft Word documents from memory. Also, we implemented several improvements over previous attempts to acquire the cache that make the result attributable and more accurate.

Previous tools that have been used to extract files from memory have overlooked these details in their implementation. The techniques we have introduced can give analysts further insight into the state of a system from the contents of memory and greatly reduce the search space using hashing principles.

VIII. References

[1] *Volatility. Volatile Systems* <http://code.google.com/p/volatility/>.

[2] *Memoryze (Memory Analyzer).* <http://www.mandiant.com/products/free_software/memoryze/>.

[3] Mark E. Russinovich, and David A. Solomon. *Windows Internals*. 4th ed. Redmond: Microsoft, 2005. Print.

[4] Brendan Dolan-Gavitt. "The VAD Tree: A Process-eye View of Physical Memory." *Digital Investigation* 4 (2007): 62-64. Print.

[5] The Honeynet Project. "Challenge 3 of the Forensic Challenge 2010 - Banking Troubles The Honeynet Project." *Challenge 3 of the Forensic Challenge 2010 - Banking Troubles*. The Honeynet Project. Web. <http://www.honeynet.org/challenges/2010_3_banking_troubles>.

[6] Josh Smith, Matt Cote, Angelo Dell'Aera, and Nicolas Collery. "Forensic Challenge 3: Banking Troubles Solution." *Honeynet*. Honeynet, 12 May 2010. Web. <http://www.honeynet.org/files/Forensic_Challenge_3_-_Banking_Troubles_Solution.pdf>.

[7] Jesse Kornblum, and Kris Kendall. "Foremost." *Foremost*. Air Force OSI, 1 Mar. 2010. Web. <http://foremost.sourceforge.net/>.

[8] Moonsols. "Moonsols Bin2dmp.exe." *MoonSols Products*. Moonsols. http://www.moonsols.com/products/>.

[9] James Okolica, and Gilbert L. Peterson. "Windows Operating Systems Agnostic Memory Analysis." *Digital Investigation* (2010): S48-56. Print.

[10] Seyed M. Hejazi, Mourad Debbabi, and Chamseddine Talhi. "Automated Windows Memory File Extraction for Cyber Forensics Investigation." *Journal of Digital Forensic Practice* (2008): 117-31. Print.

[11] Harlan Carvey. "Regripper." *RegRipper*. Web. <http://regripper.wordpress.com/regripper/>.

[12] Brendan Dolan-Gavitt. "Forensic Analysis of the Windows Registry in Memory." *Digital Investigation* (2008): 26-32. Print.

[13] "Finding File Contents in Memory." *The NT Insider* 11.1 (2004). Print.

[14] Jesse Kornblum. "Identifying Almost Identical Files Using Context Triggered Piecewise Hashing." *Digital Investigation* 3 (2006): 91-97. Print.

[15] R. B. Van Baar, W. Alink, and A. R. Van Ballegooij. "Forensic Memory Analysis: Files Mapped in Memory." *Digital Investigation* (2008): S52-57. Print.

IX. Acknowledgements

Special thanks to Peter Silberman and Jen Andre for their participation in supporting this research.