NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

OFFLINE FORENSIC ANALYSIS OF MICROSOFT® WINDOWS® XP PHYSICAL MEMORY

by

John S. Schultz

September 2006

Thesis Advisor:	Chris Eagle
Second Reader:	George Dinolt

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE *Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE:
Offline Forensic Analysis Of Microsoft® Windows® XP Physical Memory

5. FUNDING NUMBERS

6. AUTHOR(S) John Schultz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A.

13. ABSTRACT (maximum 200 words)

The rise of cyber crimes combined with the recent use of computer viruses and malicious programs that reside only in volatile main memory demand further development of appropriate forensic tools. Existing forensic tools that analyze non-volatile memory are not capable of analyzing volatile memory and the few tools that are capable of detailed analysis of volatile memory are not openly available to the public. In this thesis, an open source tool is developed to analyze images of physical memory originating from the Windows XP and Windows 2003 Server operating systems. The tool, named Windows Physical Memory Offline Analyzer (WPMOA), scans the memory image and, utilizing input from the user, extracts relevant data from the various structures maintained by the Windows operating system. The WPMOA program automatically generates reports about the image and provides key information necessary for a user to perform additional manual investigation of the image beyond what is done automatically. This thesis details instructions on the preparation and use of the program, initial testing results of the program with actual physical memory images, and C language code for the program itself.

14. SUBJECT TERMS Forensics, Physical Memory, RAM, Cyber Crime, Computers

15. NUMBER OF PAGES 91

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

OFFLINE FORENSIC ANALYSIS OF MICROSOFT® WINDOWS® XP PHYSICAL MEMORY

John S. Schultz
Ensign, United States Navy
B.S., United States Naval Academy, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: John S. Schultz

Approved by: Chris Eagle
Thesis Advisor

George Dinolt
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The rise of cyber crimes combined with the recent use of computer viruses and malicious programs that reside only in volatile main memory demand further development of appropriate forensic tools. Existing forensic tools that analyze non-volatile memory are not capable of analyzing volatile memory and the few tools that are capable of detailed analysis of volatile memory are not openly available to the public. In this thesis, an open source tool is developed to analyze images of physical memory originating from the Windows XP and Windows 2003 Server operating systems. The tool, named Windows Physical Memory Offline Analyzer (WPMOA), scans the memory image and, utilizing input from the user, extracts relevant data from the various structures maintained by the Windows operating system. The WPMOA program automatically generates reports about the image and provides key information necessary for a user to perform additional manual investigation of the image beyond what is done automatically. This thesis details instructions on the preparation and use of the program, initial testing results of the program with actual physical memory images, and C language code for the program itself.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank LCDR Chris Eagle, USN (Ret.) and Harlan Carvey for providing the idea for this thesis and for their help and advice throughout.

I would also like to thank Dr. George Dinolt for his input and constructive criticism on the written report.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CYBER CRIME TODAY

No one can deny the continuing rise in the rate of computer crime with each passing year. Reading about a new computer virus or worm in the newspaper has become a common occurrence in recent history. Computer attacks such as the *I Love You* worm in 2000 [1], *Code Red* worm in 2001 [2], and *Blaster* worm in 2003 [3] proved the reality of this type of threat. As the technology of computer security advances, so too does the sophistication of the attacks on computers. A recent article in “Core Security Technologies” discusses a computer exploit that remains entirely in volatile memory, leaving no trace on a computer system's hard drive [4]. An example of malicious software that resides in volatile memory only is the *W32.Witty.Worm* worm [5]. When a malicious program is capable of residing solely in volatile memory, it will leave no evidence of itself on the compromised system [4]. Without evidence of its activity and origin, an attack can continue to wreak havoc upon businesses, government agencies, and private citizens unchecked by law enforcement agencies.

B. COMPUTER FORENSICS

The impact forensic science has had on countless criminal investigations and trials make it a crucial part of law enforcement [6]. Therefore, it is necessary to continue advancing the forensic science to meet the increasing demand of law enforcement against cyber crime. In cyber crime investigations, the crime scene can consist of one or more computers perhaps spanning one or more computer networks. A cyber criminal may affect a system locally or remotely. A local attacker may leave physical evidence at the scene such as witnesses or fingerprints in addition to electronic evidence. Via the Internet, a remote attacker can penetrate other systems connected to the Internet from anywhere in the world, leaving only electronic evidence. In either case, digital forensic evidence can be gathered from the criminal's computer, the victim's computer, or both. This digital evidence can be broadly categorized in two ways, non-volatile and volatile.

Non-volatile electronic evidence can be recovered after a system is powered down and is found on hard drives, USB flash drives, and floppy disks. It is in non-volatile memory where most of the electronic evidence originates. System logs, network logs, malicious code, corrupted files, emails, internet browser cached files and history, and deleted files are all forensic evidence stored in non-volatile memory. Network logs may contain TCP session logs indicating the source IP address from where the attack originated. The malicious code may be analyzed to determine exactly what the attacker did to the system. Emails may contain incriminating records of criminal activity and possibly reveal accomplices. Analysis of the disk drive's file system can lead to the recovery of deleted files, which may contain further evidence. Electronic evidence gathered from non-volatile memory can be used to determine how and when a system was infiltrated, what files were corrupted and how, and how much damage, if any, was done to the system. For the criminal's computer, email and browser history and cache can prove the criminal's intent, expose any accomplices, and even give further evidence of how an attack, if any, occurred [7]. However, a careful cyber criminal may have permanently erased any incriminating evidence from non-volatile memory, thereby making its recovery impossible. In the case of an infiltrated computer running malicious code, there is other evidence that can be useful.

The other type of electronic evidence is in volatile memory. Unlike data stored on hard drives, electronic evidence found in main memory disappears once power is removed from the system. Information about each running process, such as create times, exit times, open files, executing code, and child process are stored in main memory. This type of evidence is useful if a malicious program is running or another program has been corrupted on a live system. Unlike the non-volatile memory, this evidence cannot be erased from memory as long as malicious code is running. Additionally, trusted programs may be used to gather data from a live system such as open network ports, established network connections, logged on users, and list of running processes.

As with other forms of forensic evidence, special tools are necessary to analyze computer crime scenes. At present, there exist few computer forensic tools capable of performing anything other than a rudimentary analysis of non-volatile evidence gathered from running computer systems. Jorge Urrea researched and offered a method for forensically analyzing physical memory images for Linux systems. Urrea's method combined the use of a hexadecimal editor, some Perl scripts developed by Urrea, and knowledge of the Linux kernel memory manager provided by his research. Other than the work done by Urrea, the computer forensics field lacks the tools necessary to analyze RAM contents for forensic evidence.

The Digital Forensics Research Workshop (DFRWS) issued a challenge prior to their 2005 conference in hopes of addressing the lack of tools for analyzing physical memory for Windows 2000 systems. One of the questions posed by the challenge required finding hidden processes given a physical memory dump of a Windows 2000 system. Additionally respondents were asked to consider what other types of forensic evidence one could gather from physical memory dumps. The two winners of the challenge created their own tools to analyze the physical memory dumps, but neither of the tools has been made available to the public [8].

Inspired by the DFRWS 2005 challenge, this thesis will discus the details involved in creating an open source C program to analyze Windows XP physical memory dumps for digital forensic evidence. To begin, the methods, tools, and challenges associated with obtaining a snapshot of physical memory will be discussed in chapter II. Once a physical memory image is obtained, its contents must be understood in order to obtain meaningful forensic information from the image. The structures maintained in physical memory will be described in chapter III. Once the contents of memory are understood, the software used to automate the analysis of the image will be covered in chapter IV and chapter V will describe how to use the Windows Physical Memory Analyzer (WPMOA) program developed for this thesis. The results from using the program to analyze actual physical memory images will be discussed in chapter VI. The final chapter will provide areas for further research on the topic of automating forensic analysis of physical memory images.

THIS PAGE INTENTIONALLY LEFT BLANK

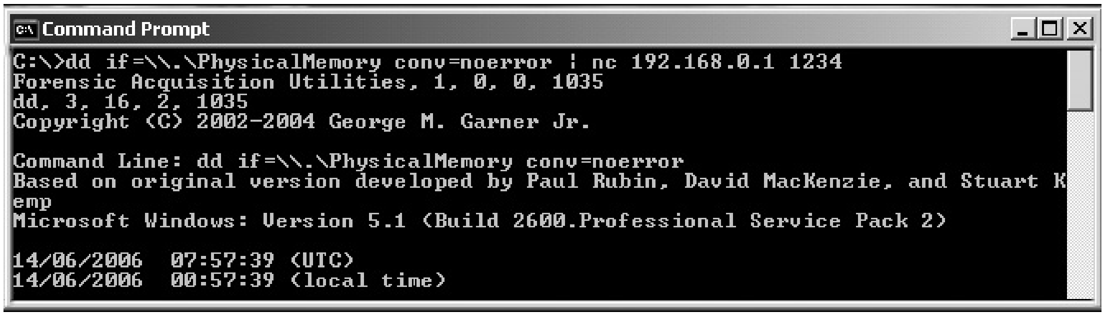
II. THE PHYSICAL MEMORY IMAGE

To analyze forensic evidence from physical memory, one must be able to extract the contents of RAM. Ideally, when a process reads from or writes to a location in its virtual address space, that address is available in physical memory for immediate use. But no system is capable of providing enough physical addresses in main memory to map to every address of each process' virtual address space. So, when virtual memory space exceeds physical memory space, non-essential data mapped to physical memory is transferred to the system hard disk, to free physical addresses for remapping. This process is known as paging out. The data is paged to a file on the system hard disk called a swap file. On Windows systems, the swap file is named pagefile.sys and is located in the root directory of the C drive (C:\). The swap file is conceptually just an extension of physical memory to meet the capacity of virtual memory. For ease of management, main memory is divided into fixed sized chunks called pages. While a system is powered on, pages in memory are swapped to and from the swap file as new processes are created and as the existing processes run, as needed to meet the demand of the running processes. Because of the constant swapping of pages in physical memory, the total address space of virtual memory is rarely contained entirely within physical memory, nor is it ever constant.

When a forensics imaging program is used to copy the contents of physical memory to a file, that same memory will be altered by the operating system which must create data structures to manage the newly created imaging process. Furthermore, the imaging program itself will alter physical memory as it copies the data in memory to a file which typically requires a large number of data transfers between various memory resident input/output buffers. Besides affecting memory, it is possible that the program could be receiving false data from a corrupted operating system since a memory imaging program can only access physical memory through the operating system. Thus, with software tools alone, it is not possible to obtain a snapshot of physical memory unaffected by the tool used to measure it. Simply stated, the act of capturing the state of memory causes changes to the state of memory. However, it is possible to obtain a snapshot of physical memory without altering its state by using a bus mastering hardware device connected to the system’s memory bus. The device must be connected to the system prior to intrusion, when the system is powered down, but because it communicates with physical memory through the host controller and not the operating system, the problems discussed with software acquisition of physical memory are eliminated. For those who do not have a bus mastering device to capture memory or did not have the foresight to install said device before an intrusion, snapshots of the unaltered physical memory image are not possible. An undisturbed snapshot is not necessary for forensic analysis because, as will be shown in this document, useful forensic evidence can be extracted from a physical memory image captured with the software tool dd [9].

Since there are no native tools on Windows XP to dump the contents of system memory to a file, a third-party program must be used. One such tool is George Garner’s variant of the GNU dd utility [10]. George Garner’s dd is used in a manner similar to the way in which the standard version of dd is used on Linux systems [11]. The dd program copies data in blocks from one file to another. In the case of Garner’s version, the input file is \\.\PhysicalMemory, which represents the device name for physical memory in Windows. It is important to use the *noerror* option to ensure all readable blocks of data are copied from physical memory. Without the *noerror* option, the dd program will stop copying after a read error. As for the output file, it is recommended that no output file is specified in favor of piping the output of dd to netcat via the command line (see Figure 1). The netcat program sends and receives data via the host’s network connection, thus allowing the copy of physical memory to be sent directly to an external, network connected computer. This method will produce the best results for forensic analysis because it will not alter the target system’s hard disk.

Figure 1. Using dd to obtain physical memory image



```
C:\>dd if=\\.\PhysicalMemory conv=noerror ¦ nc 192.168.0.1 1234
Forensic Acquisition Utilities, 1, 0, 0, 1035
dd, 3, 16, 2, 1035
Copyright (C) 2002-2004 George M. Garner Jr.

Command Line: dd if=\\.\PhysicalMemory conv=noerror
Based on original version developed by Paul Rubin, David MacKenzie, and Stuart K
emp
Microsoft Windows: Version 5.1 (Build 2600.Professional Service Pack 2)

14/06/2006  07:57:39 (UTC)
14/06/2006  00:57:39 (local time)
```

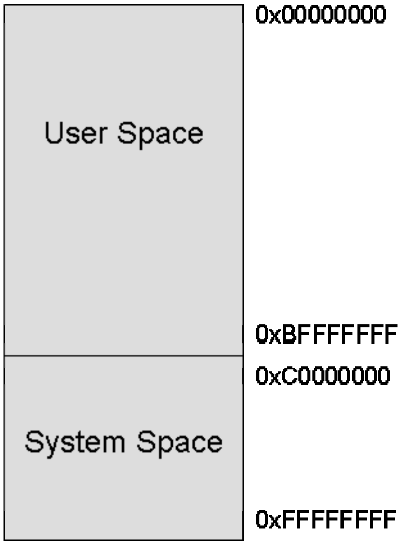
After obtaining an image of Windows physical memory using George Garner’s dd program, forensic analysis can be performed on the image just as traditional forensic analysis is performed on hard disk images. The fundamental problem with analyzing physical memory structures however is that unlike disk partition images which contain well structured file systems, the layout of data in RAM must be carefully reconstructed by locating and properly parsing operating system data structures that are also resident in the RAM image. Therefore, in order to analyze physical memory and its relationship with virtual memory, the internal structures scattered throughout the RAM image must be understood first.

THIS PAGE INTENTIONALLY LEFT BLANK

III. WINDOWS VIRTUAL MEMORY

The Windows operating system provides every running process with a virtual private memory space. The virtual space is an abstract representation of each process' memory space. In reality, there is only physical memory (RAM) and it is shared among all processes on a system. All virtual address references must be translated to physical address addresses before data can actually be read or written.

Figure 2. Layout of virtual address space



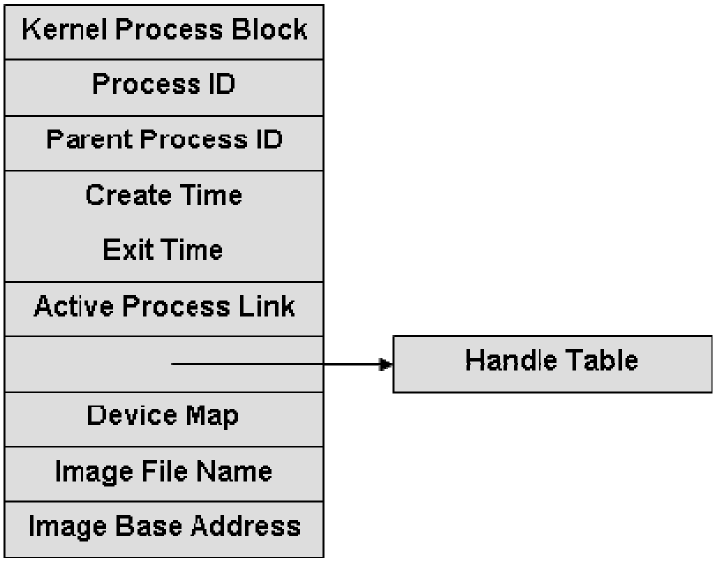
A. VIRTUAL MEMORY LAYOUT

Virtual memory for Microsoft Windows XP and Windows 2003 Server on the Intel x86 architecture is the subject of this entire document and is assumed constant unless otherwise noted. Each active process on a system has its own virtual memory address space, separate from all other processes' virtual memory address space. Within each virtual address space, there is a portion dedicated for the user application and a portion for the operating system as shown in Figure 2. The user virtual address space is for use by the application itself (e.g. Internet Explorer) and contains the application code, global variables, thread stacks, and dynamically linked library code. It spans from virtual addresses (in hexadecimal) 0x00000000 to 0xBFFFFFFF. The system address space is for use by the operating system and is accessible by all processes. The system address space ranges from virtual addresses 0xC0000000 to 0xFFFFFFFF. It contains the necessary information for system management of the virtual memory space, including the process' page directory and the page table entries (PTEs) used for translating virtual addresses to physical addresses in memory.

B. VIRTUAL MEMORY STRUCTURES

There are numerous structures within the process' virtual address space and not all of them relate to the work described in this document. Therefore, the following discussion of memory structures is limited to the structures of interest in our forensic analysis of Windows physical memory. Each structure discussed is defined as a C structure in the win32structs.h file in Appendix A.

Figure 3. Structure of an EPROCESS block



1. Executive Process

The executive process (EPROCESS) block is the most important structure found in memory for forensic analysis because it is the starting point of any further investigation of that process. Figure 3 shows an abbreviated version of its structure, listing the fields of interest for forensic analysis. All EPROCESS blocks are part of a doubly linked list, which includes blocks for active processes, although it is not uncommon to encounter exited processes in the list. This phenomenon occurs when an exited process is still opened as a handle by another active process, as an exited process'

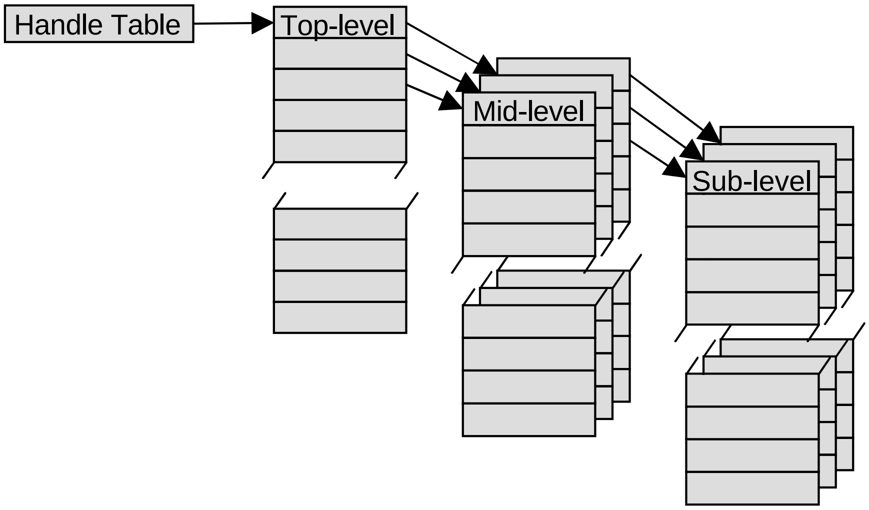
EPROCESS block is deleted only when the last handle to the process is closed. See Table 1 for a description of key EPROCESS block fields [12][13].

Table 1. Description of EPROCESS block fields [13]

Field Name	Description
Kernel Process Block or Process Control Block (PCB)	Contains information about thread user and kernel times, process state, the pointer to the process page directory, and the pointer to the list of active threads for that process.
Process ID	A unique number identifying the process.
Parent Process ID	The process ID of the process' parent process. Where the parent is the process that spawned the process in question.
Create and Exit times	Values representing the date and time the process was created and, if applicable, exited. Note: The System process does not have a valid Create Time and this field is null.
Active Process Links	This field is a structure containing a forward and backward link that facilitates the doubly linked list of all processes.
Device Map	This field contains the address of the process' object directory, which is used to resolve object names for the process.
Image File Name	This field contains the process' name limited to 16 characters (e.g. explorer.exe).
Image Base Address	The preferred base address of the process in the user address space.

Field Name	Description
Virtual Memory Information	This field is a pointer to the process' Memory Manager Support structure

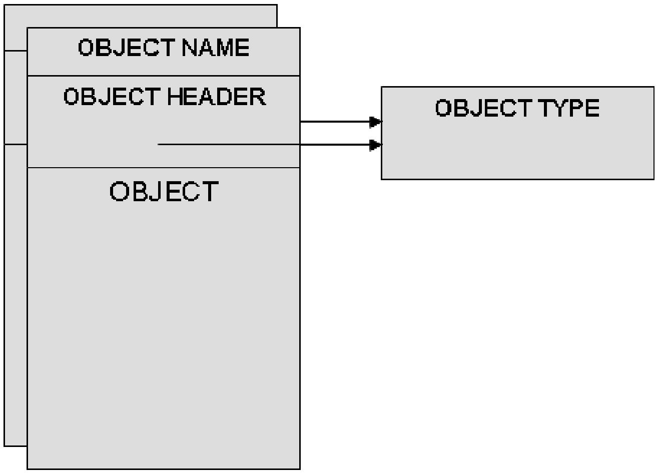
Figure 4. Handle table hierarchy



2. Handle Table

Each process accounts for every object it currently has open and it does so via the handle table. The process' EPROCESS block contains a pointer to its handle table structure (see Figure 4). Within the handle table structure, there is a table code field, which contains the address of the actual handle table(s). The address in the handle code field is that of a sub handle table by default. The handle number is an index of the handle table entry for an object. Each entry in a sub handle table is represents a pointer to an opened object and a 32-bit value used to determine access control to that object. There can be up to 511 entries in one sub handle table. When more than 511 objects are opened in a process, the object manager creates a middle-level handle table, which contains pointers to a maximum of 1024 sub handle tables (see Figure 4). Additionally, one top-level handle table can be created, which contains pointers to a maximum of 1024 middle-level handle tables. A handle table's type is indicated in the least significant 3 bits of its table code field. The method is as follows: 0 indicates address is of a sub handle table, 1 indicates the address is of a middle-level handle table, and 2 indicates the address is of a top-level handle table [13].

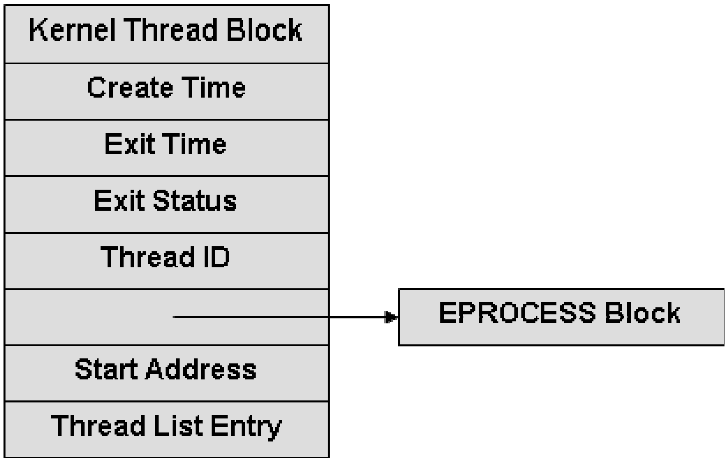
Figure 5. Generic object structure



3. Objects

Figure 5 shows the generic structure of all objects found in memory. Every object has an object header structure. The object header stores the address of that object's object type structure. The object type structure is common among all objects of the same type; some common examples of object types are file, process, and thread. Most objects have a body and a name, which precedes the object header, but where an object's name is stored is ultimately dependant on its type. The object names facilitate finding objects by name and the sharing of objects between processes. If the object is not to be shared, the process' object manager will not assign a name to it and it will be referred to by object handle only [13].

Figure 6. Executive thread block structure



4. Executive Thread

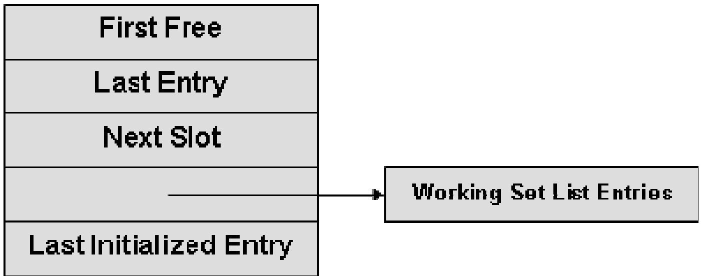
Threads are entities within a process that represent executable code of the process. Every process created begins with a single thread, which can create other threads. Multiple threads allow a process to perform several tasks in parallel, with each thread executing a separate task. Windows represents a thread by the executive thread (ETHREAD) block stored in the physical memory. Threads are important to forensic analysis because threads are scheduled for execution by the operating system, not processes. A thread’s creation and exit times are stored in its ETHREAD block. The initial execution address for a thread is also stored in the ETHREAD block. Figure 6 shows an abbreviated ETHREAD block structure [13].

5. Process Working Set

A process’ working set is the set of virtual pages currently residing in main memory. When a thread references a virtual page that is not in the working set, a page fault occurs. The page fault triggers the memory manager to make room for the desired page by moving a resident page from main memory to the system swap file. Once space is made available, the desired virtual page is loaded into memory and the working set is updated to include the newly loaded virtual page [14].

The working set list structure contains fields for the memory manager to update the working set list easily. The addresses of the first free slot, the last entry, and last initialized entry are kept in the structure (see Figure 7). These are used by the WPMOA program to determine when to stop searching for working set addresses. The working set list entry field contains the virtual address of the working set list entry table. In this table are the virtual addresses of every resident page of the process.

Figure 7. Working Set List structure



Each structure maintained in physical memory stores a small amount of data about the overall system. The aggregation of that data is what leads to profound information about a system. Therefore, it is essential to understand where to find and how to interpret the many structures scattered throughout physical memory in order to recover useful information. Once these structures are understood, a program can be implemented to automate the recovery of relevant forensic data from a physical memory image.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SOFTWARE IMPLEMENTATION

The source code for the Windows Physical Memory Offline Analyzer (WPMOA) software is included in appendix A. The WPMOA program was developed using ANSI standard C for UNIX-based operating systems and has been successfully compiled and executed on SuSE Linux 10.0 and Mac OS X Tiger (Intel) Darwin 8.0.

A. PROGRAM OVERVIEW

The WPMOA program consists of two C source files named wpmoa.c and dissector.c and two C header files named wpmoa.h and win32structs.h. The wpmoa.h file contains all of the function prototypes, global variables, and include statements, while the win32structs.h file contains struct definitions based on those defined in the Ntddk.h file included in the Windows Driver Development Kit (DDK). Functions scanning for and manipulating the data found on the memory image are located in the dissector.c source file, while all other functions are in the wpmoa.c source file.

Figure 8. WPMOA program flow chart

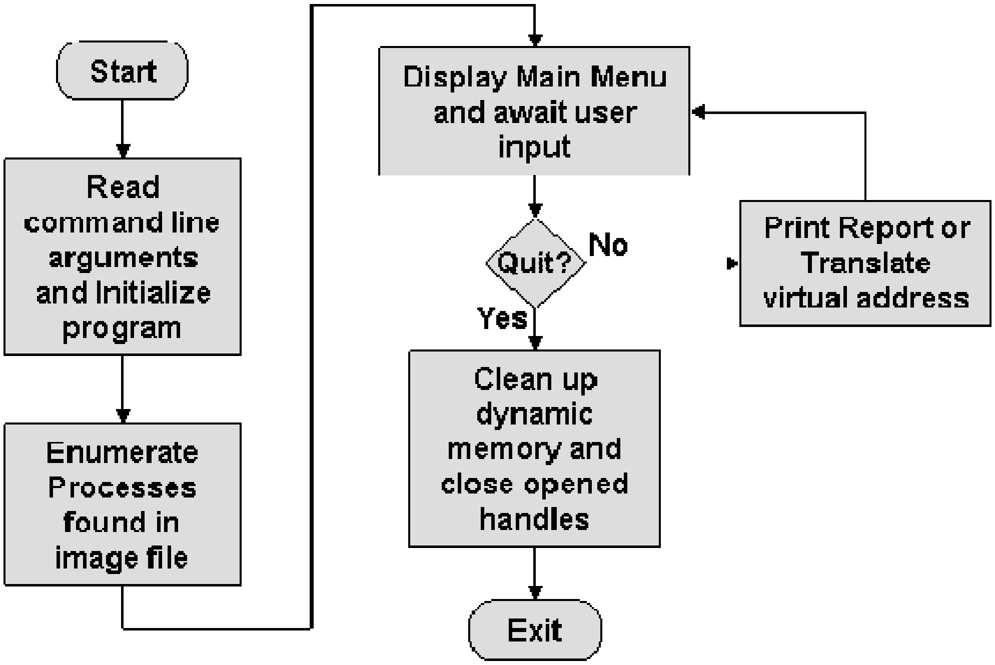
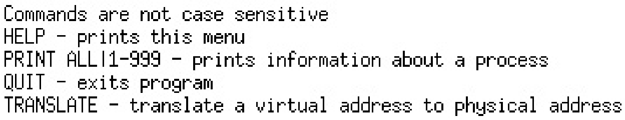

Start
Read command line arguments and Initialize program
Enumerate Processes found in image file
Display Main Menu and await user input
Quit?
No
Yes
Print Report or Translate virtual address
Clean up dynamic memory and close opened handles
Exit

Figure 8 shows the general flow of the program, which is described in words below. The program uses the user command line inputs to initialize the program global variables setting the start address of the active process list and the page size for the physical memory image. The program then enumerates each process in the active process list by copying each EPROCESS block into a dynamically allocated array, along with the open handles and working set of each process. Once completed, the program prints the main menu and awaits user input for the next action. The user may choose to translate a virtual address to a physical address, print a report of one or all processes, or quit the program. After the chosen action is completed, the program will return to the command prompt awaiting further action until the user chooses the quit option. When quit is chosen, the program cleans up all dynamically allocated memory and closes all opened handles before exiting.

B. INITIALIZING AND MAIN MENU

There is no constant value for the address of the first EPROCESS block in the active process list; therefore, this value must be set manually via the command line arguments. The names of the physical memory image file and the output file are passed to the program via the command line arguments as well. On startup, the program scans a physical memory image, and allocates a dynamic array to hold all EPROCESS blocks found in the image. The size of the array is determined by the process_count function, which traverses the active process list, found using the address provided via the command line argument, in the physical memory image, keeping count of the number of processes traversed. The program then utilizes the enumerate_processes function, described later, which fills the allocated array with data read from the EPROCESS blocks. If the enumerate_processes function executes successfully, the interactive menu will display a command prompt preceded by a table of all processes found. There is also a help menu, showing all available options to the user (see Figure 9).

Figure 9. WPMOA help menu



```
Commands are not case sensitive
HELP - prints this menu
PRINT ALL|1-999 - prints information about a process
QUIT - exits program
TRANSLATE - translate a virtual address to physical address
```

C. ENUMERATING PROCESSES

The enumerate_processes function, found in dissector.c, copies the contents of each EPROCESS block found in the active process list into an array allocated in main. Traversing the list is relatively straight forward because the virtual addresses of each

EPROCESS block did not require process specific address translation. The virtual addresses of the EPROCESS blocks differ from the physical addresses by the kernel offset of 0x80000000 and are determined by Equation 1, where VA is the virtual address and PA is the translated physical address. To determine the end of the list, the program checks the forward link of each EPROCESS block against the address of the head of the list. If the two are equal, the EPROCESS block is the last process in the list.

Equation 1

$$PA = VA - 0x80000000$$

Enumerating a process' open file handles requires process specific virtual to physical address translation. The translation function, described later, enables further enumeration of a process. A process' open file handles are found by the enumerate_handle_table function, also located in dissector.c. As mentioned in chapter II, the open file handles are kept in a hierarchy of handle tables. The enumerate_handle_table function traverses any mid-level and top-level handle tables, eventually reaching all sub handle tables. Each object pointed to by the file handles is visited and its name and type is recorded. Since each object of the same type points to a common object type structure in memory, a cache is kept of the object type names to save time wasted by repetitive disk reads visiting the same object type structure. Once the object type is known, the object's name, if one exists, is determined based on its type. A process object's name resides in the process object's body, as does a file object's name. The thread object does not have a text name; therefore, it is referred to by its unique identification number and by the process name that spawned it. All other object types, except key objects (used to access registry data), maintain their object names in the object name block, preceding the object header block. All object names are stored as Unicode, which are converted to ASCII strings and stored for later printing.

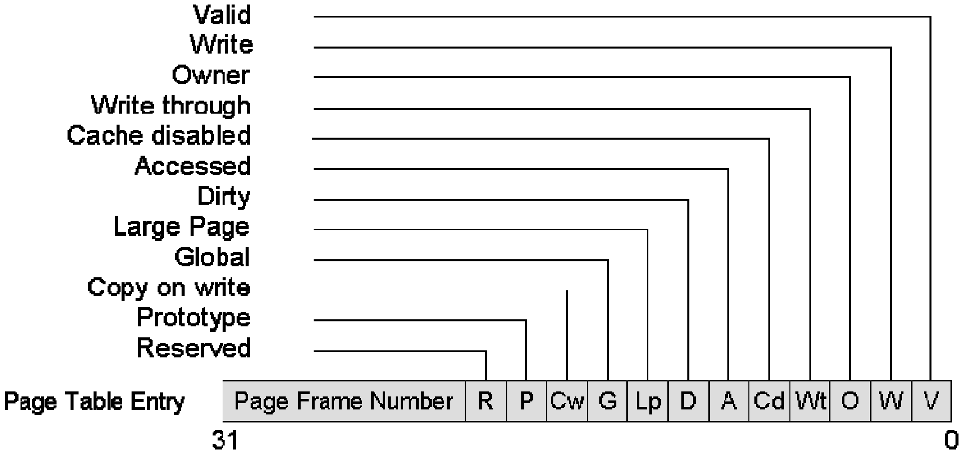
Enumerating file objects does not mean the contents of that file can be discovered as well. File objects are references to files stored on a hard drive or any other block device. If the file name is known and the system hard drive is possessed, then the file's entire contents could be inspected for forensic evidence. However, if the system hard drive is not in possession or the process' executable file was wiped from the hard drive, then the process' executable code cannot be examined. Once mapped into memory by the operating system loader, a process' executable file is generally not opened for I/O and has no assigned file descriptor; therefore, in order to view the process' executable code, the process' working set must be enumerated. A process' working set is cached data referenced by its threads. This data includes the executable code each thread is currently referencing. The working set is arranged in a list of virtual addresses. Each address refers to a location in memory containing at most one page of data or code. The WPMOA program copies the working set list into an array containing both virtual addresses and their translated physical addresses, which is then sorted based on the virtual addresses from lowest address to highest. The list is then traversed, comparing each address to the address before it to determine contiguous addresses. If addresses are found to differ by the size of a page in memory (typically 4K bytes), then they are considered contiguous, but contiguous addresses that differ in the twelve most significant bits are put into separate files. All contiguous address ranges are then extracted to files named after the address ranges contained in the file (e.g. 00400000h-00402000h). The files created are placed in a directory named according to the owning process name and unique ID number. Including the process identification number is necessary because a program can be executed several times and be referred to by the same image name, but each process will have a unique identification number.

Each working set item from the working set list is the size of one page. Depending on the system settings, the page size could vary from the typical 4096-byte value. If the physical memory image page size is not 4096 bytes, then the actual size, in bytes, must be set via the command line arguments. Additionally, each working set is a referenced section of data from the virtual address space of the process. Therefore, when viewing the files it is normal to see large spaces of zeroes even though there is a much smaller space of zeroes in the file containing the original data. This is due to the difference in the layout of an executable image on disk as defined by the Portable Executable (PE) file specification, and the way in which that image is mapped into memory by the operating system loader. Program file images consist of various sections and the headers of a PE file detail the difference in size of each program section as it exists on disk with the size of that section as it exists one mapped into memory.

D. VIRTUAL ADDRESS TRANSLATION

All pages in a process' private address space are located by a virtual address. For the program to read locations in the physical memory image referred to within a process, that virtual address had to be translated to a physical one. The address translation begins with the virtual address to be translated. The first (most significant) 10-bits of the virtual address is an index in the process' page directory. The page directory is found in the Kernel Process block and is referred to by its physical location. The entry in the page directory contains the valid address of a page table, if the desired page table resides in physical memory, where page tables contain addresses of page frames. The next 10-bit value of the virtual address is an index to the page table entry containing the desired page frame address. If the page is not actually in physical memory, because it has been swapped to the page file, then the page table entry will indicate this by setting the least significant bit of the page table entry and the contents of that page will not be recovered by the WPMOA program. There are a total of twelve bits in the page table entry that are used by the memory manager and are not part of the physical address (see Figure 10). The final 12-bits of the virtual address take the place of the 12-bits lost to the memory manager to form a translated 32-bit physical address [13].

Figure 10. Page Table Entry



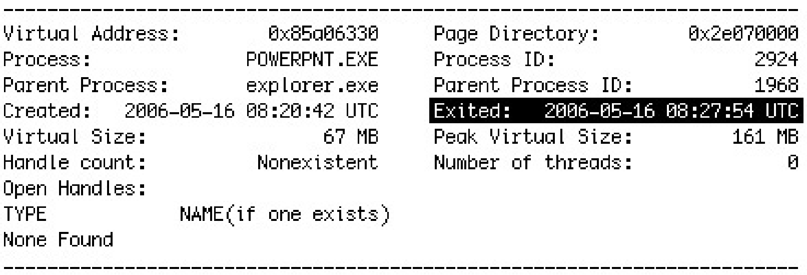
E. THE REPORT

The print command of the program prints a report of data found in the physical memory image and calls the function that enumerates the process' working set. Although there is much data found on the image, the program was designed to display only the most relevant data found. The program generates a report, which displays the virtual address, image name, process ID, the parent process' name, the parent process ID, time of creation, time of exit, virtual size, peak virtual size, page directory address, handle count, number of active threads, and the types and names of every open handles of every process found in the image file. The information displayed was chosen to give as much relevant forensic data to the user as possible and provide flexibility for further analysis.

The process ID numbers can be used to trace how a process was executed. For example, if a process' parent process ID was that of cmd.exe, then that process was executed from the Windows command line. Therefore, each process in the open handle table has a number in parenthesis, which is the unique process ID of the opened process. For threads in the open handle table, the parent process of the thread is shown with the unique ID number of the thread following a semicolon.

The process creation and exit times are useful for recreating a timeline of system events. All times are shown as Coordinated Universal Time (UTC). Since the target system originated the times in the physical memory image, their accuracy depends on the accuracy of the target system's clock. If the reported process was exited, then a time will occupy the space next to the Exited Time field, otherwise that space will indicate that no time was reported (see Figure 11).

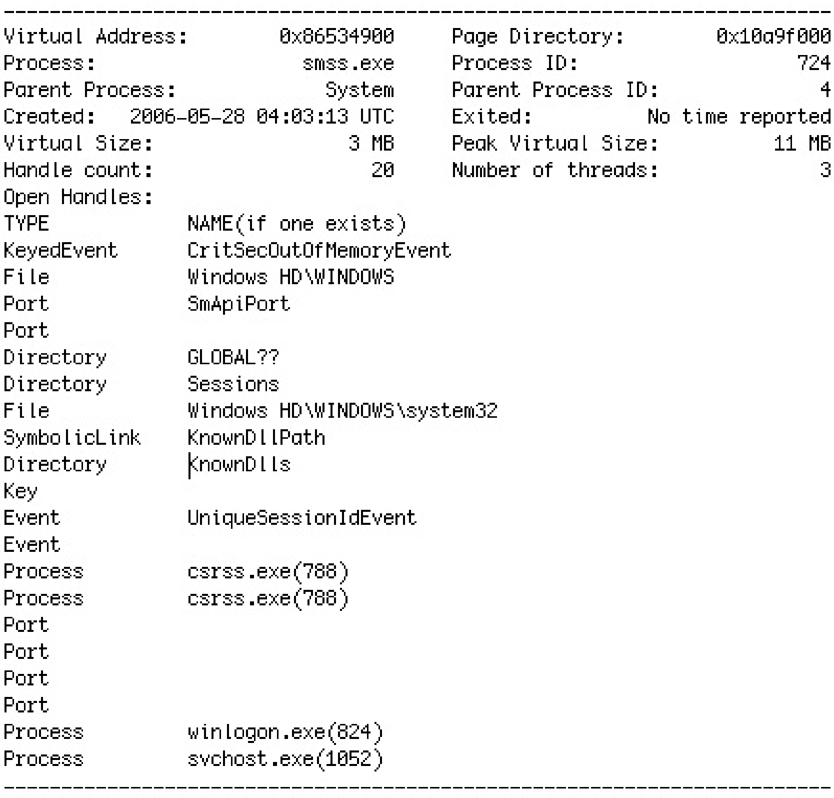
Figure 11. WPMOA report of exited process



```
------------------------------------------------------------------------
Virtual Address:        0x85a06330     Page Directory:        0x2e070000
Process:              POWERPNT.EXE     Process ID:                  2924
Parent Process:       explorer.exe     Parent Process ID:           1968
Created:   2006-05-16 08:20:42 UTC     Exited:   2006-05-16 08:27:54 UTC
Virtual Size:                67 MB     Peak Virtual Size:         161 MB
Handle count:          Nonexistent     Number of threads:              0
Open Handles:
TYPE            NAME(if one exists)
None Found
------------------------------------------------------------------------
```

For users desiring more information than the WPMOA program provides, the report gives the necessary information to do so. Specifically, the page directory address is intended to be used with the virtual address translation feature of the WPMOA program. The virtual address translation feature provides necessary functionality to explore the physical memory image beyond the basic analysis of the WPMOA program. With it, the virtual addresses referencing every structure in RAM can be translated to physical locations in the image file. To analyze a process further, its virtual address is provided so that its location can be found in the image file. To navigate through the physical memory image, use the virtual address translator feature of the WPMOA program in conjunction with the Windows system header files. Some portions of the header files are included in the win32structs.h file (see Appendix A) used by the WPMOA program.

Figure 12. Example report of smss.exe



```
-----------------------------------------------------------------------
Virtual Address:        0x86534900     Page Directory:        0x10a9f000
Process:                  smss.exe     Process ID:                   724
Parent Process:             System     Parent Process ID:              4
Created:   2006-05-28 04:03:13 UTC     Exited:          No time reported
Virtual Size:                 3 MB     Peak Virtual Size:          11 MB
Handle count:                   20     Number of threads:              3
Open Handles:
TYPE            NAME(if one exists)
KeyedEvent      CritSecOutOfMemoryEvent
File            Windows HD\WINDOWS
Port            SmApiPort
Port
Directory       GLOBAL??
Directory       Sessions
File            Windows HD\WINDOWS\system32
SymbolicLink    KnownDllPath
Directory       KnownDlls
Key
Event           UniqueSessionIdEvent
Event
Process         csrss.exe(788)
Process         csrss.exe(788)
Port
Port
Port
Port
Process         winlogon.exe(824)
Process         svchost.exe(1052)
-----------------------------------------------------------------------
```

A sample report of the smss.exe process is shown in Figure 12. The working set information is not printed to the screen, but is printed to files labeled by the virtual address ranges each file contains. Although a small amount of information is shown in the example, more can be displayed in future versions of the software.

E. QUITING THE PROGRAM

When the user chooses the quit option from the command prompt, the program de-allocates all dynamically allocated memory and closes all opened file handles before returning with a value of zero to indicate successful execution.

V. SOFTWARE USE

A. PRE-EXECUTION

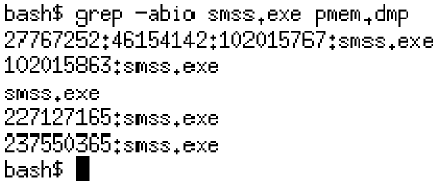
Before using the Windows Physical Memory Offline Analyzer (wpmoa) program, the head of the process list must be determined manually. This is accomplished by using the grep and hexdump tools found on UNIX or UNIX-like operating systems. The necessary steps follow.

1. Find the smss.exe EPROCESS block

The active process list maintained in physical memory begins with the System process, but because the word “System” is found in countless locations throughout memory, it is not practical to search directly for it. The next process executed at system startup is smss.exe, which is a word not commonly used throughout memory and it is a process guaranteed to be running since it is responsible for starting user sessions. Thus, to find the System EPROCESS block, the smss.exe EPROCESS block must first be found. Once the smss.exe EPROCESS block is found, the System EPROCESS block can be found by traversing the active process list backwards one link.

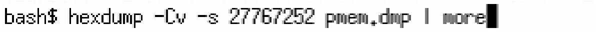
To find the smss.exe EPROCESS block, use the grep command to search the image file of the physical memory for the string “smss.exe” treating the file as ASCII text, ignoring case, and reporting only the byte offset of the line containing the desired string (see Figure 13). The grep search will most likely return several addresses, so each one must be evaluated until the smss.exe executive process block is found. Examine each potential address of the physical memory image using hexdump with the display hex+ASCII and offset options (see Figure 14). Set the offset of hexdump to one of the byte offsets returned by grep. Find the string “smss.exe” and note the address where it begins. Subtract the hex offset of the process name field (0x174) from the hex address of “smss.exe.” The new address marks the beginning of the EPROCESS block and the location of the block’s type field. Verify that the first byte starting at the new address is 0x03, which indicates that the following type is of EPROCESS block type (see Figure 15). If the byte matches, then the address at which it is found is the beginning address of the smss.exe EPROCESS block.

Figure 13. Use of grep to find smss.exe



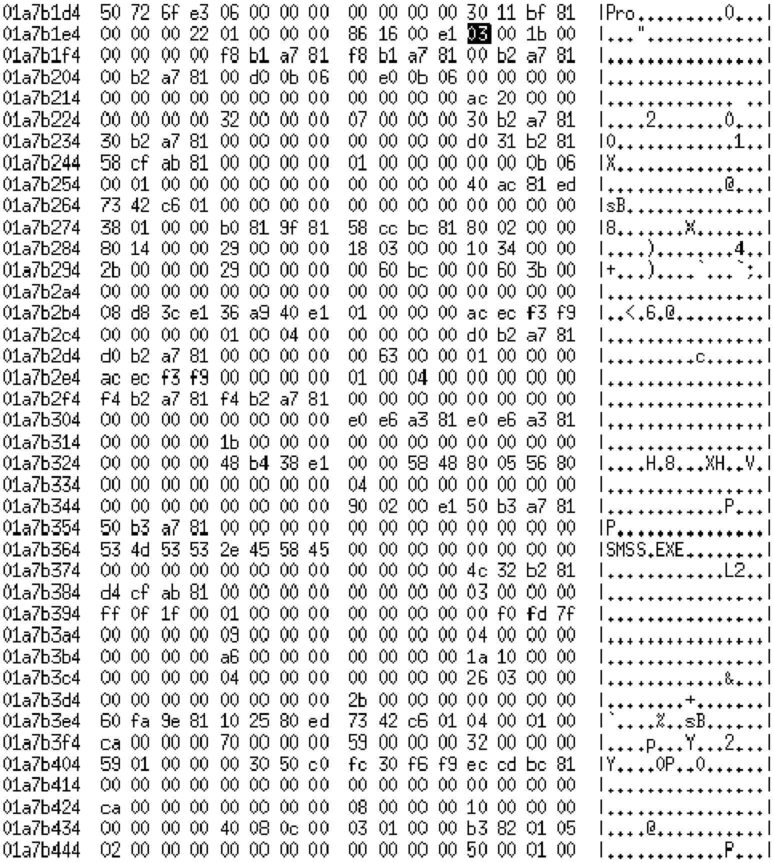
```
bash$ grep -abio smss.exe pmem.dmp
27767252:46154142:102015767:smss.exe
102015863:smss.exe
smss.exe
227127165:smss.exe
237550365:smss.exe
bash$ █
```

Figure 14. Use of hexdump command to view smss.exe EPROCESS block



```
bash$ hexdump -Cv -s 27767252 pmem.dmp | more█
```

Figure 15. hexdump view of smss.exe EPROCESS block

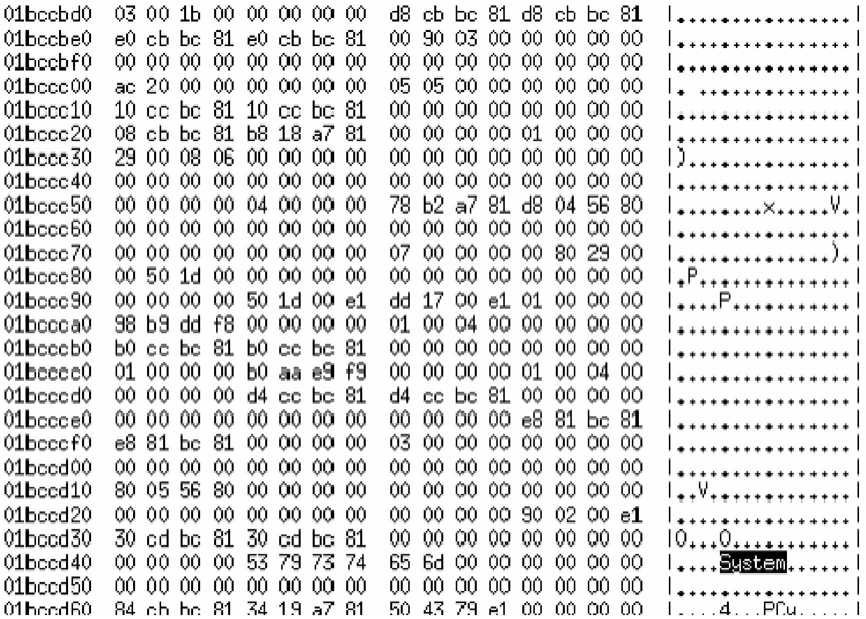


```
01a7b1d4  50 72 6f e3 06 00 00 00  00 00 00 00 30 11 bf 81  |Pro.........0...|
01a7b1e4  00 00 00 22 01 00 00 00  86 16 00 e1 03 00 1b 00  |..."............|
01a7b1f4  00 00 00 00 f8 b1 a7 81  f8 b1 a7 81 00 b2 a7 81  |................|
01a7b204  00 b2 a7 81 00 d0 0b 06  00 e0 0b 06 00 00 00 00  |................|
01a7b214  00 00 00 00 00 00 00 00  00 00 00 00 ac 20 00 00  |............. ..|
01a7b224  00 00 00 00 32 00 00 00  07 00 00 00 30 b2 a7 81  |....2.......0...|
01a7b234  30 b2 a7 81 00 00 00 00  00 00 00 00 d0 31 b2 81  |0............1..|
01a7b244  58 cf ab 81 00 00 00 00  01 00 00 00 00 00 0b 06  |X...............|
01a7b254  00 01 00 00 00 00 00 00  00 00 00 00 40 ac 81 ed  |............@...|
01a7b264  73 42 c6 01 00 00 00 00  00 00 00 00 00 00 00 00  |sB..............|
01a7b274  38 01 00 00 b0 81 9f 81  58 cc bc 81 80 02 00 00  |8.......X.......|
01a7b284  80 14 00 00 29 00 00 00  18 03 00 00 10 34 00 00  |....)........4..|
01a7b294  2b 00 00 00 29 00 00 00  00 60 bc 00 00 60 3b 00  |+...)....`...`;.|
01a7b2a4  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01a7b2b4  08 d8 3c e1 36 a9 40 e1  01 00 00 00 ac ec f3 f9  |..<.6.@.........|
01a7b2c4  00 00 00 00 01 00 04 00  00 00 00 00 d0 b2 a7 81  |................|
01a7b2d4  d0 b2 a7 81 00 00 00 00  00 63 00 00 01 00 00 00  |.........c......|
01a7b2e4  ac ec f3 f9 00 00 00 00  01 00 04 00 00 00 00 00  |................|
01a7b2f4  f4 b2 a7 81 f4 b2 a7 81  00 00 00 00 00 00 00 00  |................|
01a7b304  00 00 00 00 00 00 00 00  e0 e6 a3 81 e0 e6 a3 81  |................|
01a7b314  00 00 00 00 1b 00 00 00  00 00 00 00 00 00 00 00  |................|
01a7b324  00 00 00 00 48 b4 38 e1  00 00 58 48 80 05 56 80  |....H.8...XH..V.|
01a7b334  00 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  |................|
01a7b344  00 00 00 00 00 00 00 00  90 02 00 e1 50 b3 a7 81  |............P...|
01a7b354  50 b3 a7 81 00 00 00 00  00 00 00 00 00 00 00 00  |P...............|
01a7b364  53 4d 53 53 2e 45 58 45  00 00 00 00 00 00 00 00  |SMSS.EXE........|
01a7b374  00 00 00 00 00 00 00 00  00 00 00 00 4c 32 b2 81  |............L2..|
01a7b384  d4 cf ab 81 00 00 00 00  00 00 00 00 03 00 00 00  |................|
01a7b394  ff 0f 1f 00 01 00 00 00  00 00 00 00 00 f0 fd 7f  |................|
01a7b3a4  00 00 00 00 09 00 00 00  00 00 00 00 04 00 00 00  |................|
01a7b3b4  00 00 00 00 a6 00 00 00  00 00 00 00 1a 10 00 00  |................|
01a7b3c4  00 00 00 00 04 00 00 00  00 00 00 00 26 03 00 00  |............&...|
01a7b3d4  00 00 00 00 00 00 00 00  2b 00 00 00 00 00 00 00  |........+.......|
01a7b3e4  60 fa 9e 81 10 25 80 ed  73 42 c6 01 04 00 01 00  |`....%..sB......|
01a7b3f4  ca 00 00 00 70 00 00 00  59 00 00 00 32 00 00 00  |....p...Y...2...|
01a7b404  59 01 00 00 00 30 50 c0  fc 30 f6 f9 ec cd bc 81  |Y....0P..0......|
01a7b414  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01a7b424  ca 00 00 00 00 00 00 00  08 00 00 00 10 00 00 00  |................|
01a7b434  00 00 00 00 40 08 0c 00  03 01 00 00 b3 82 01 05  |....@...........|
01a7b444  02 00 00 00 00 00 00 00  00 00 00 00 50 00 01 00  |............P...|
```

2. Find System EPROCESS Block

Note the address located +0x08c relative to the start of the smss.exe EPROCESS block. This address marks the location of the EPROCESS block's back link, pointing to the previous EPROCESS block in the active process list. Since the smss.exe process is always the second process executed in Windows XP, the back link pointer points to the System process: the first process in the active process list. Since the back link actually points to the back link of the previous EPROCESS block, the address contained in the back link field must be adjusted to the starting address for the System EPROCESS block. Subtract 0x088 from the back link address to compute the virtual address of the head of the process list. The new address should be verified to be that of the System EPROCESS block to ensure that the program will function properly. Repeat the steps used to view the smss.exe EPROCESS block and look for the "System" string in the hexdump view (see Figure 16). Be sure to compute the physical address by subtracting the kernel offset of 0x80000000 from the virtual address when using hexdump.

Figure 16. hexdump view of the System EPROCESS block



```
01bccbd0  03 00 1b 00 00 00 00 00  d8 cb bc 81 d8 cb bc 81  |................|
01bccbe0  e0 cb bc 81 e0 cb bc 81  00 90 03 00 00 00 00 00  |................|
01bccbf0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01bccc00  ac 20 00 00 00 00 00 00  05 05 00 00 00 00 00 00  |. ..............|
01bccc10  10 cc bc 81 10 cc bc 81  00 00 00 00 00 00 00 00  |................|
01bccc20  08 cb bc 81 b8 18 a7 81  00 00 00 00 01 00 00 00  |................|
01bccc30  29 00 08 06 00 00 00 00  00 00 00 00 00 00 00 00  |)...............|
01bccc40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01bccc50  00 00 00 00 04 00 00 00  78 b2 a7 81 d8 04 56 80  |........x.....V.|
01bccc60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01bccc70  00 00 00 00 00 00 00 00  07 00 00 00 00 80 29 00  |..............).|
01bccc80  00 50 1d 00 00 00 00 00  00 00 00 00 00 00 00 00  |.P..............|
01bccc90  00 00 00 00 50 1d 00 e1  dd 17 00 e1 01 00 00 00  |....P...........|
01bccca0  98 b9 dd f8 00 00 00 00  01 00 04 00 00 00 00 00  |................|
01bcccb0  b0 cc bc 81 b0 cc bc 81  00 00 00 00 00 00 00 00  |................|
01bcccc0  01 00 00 00 b0 aa e9 f9  00 00 00 00 01 00 04 00  |................|
01bcccd0  00 00 00 00 d4 cc bc 81  d4 cc bc 81 00 00 00 00  |................|
01bccce0  00 00 00 00 00 00 00 00  00 00 00 00 e8 81 bc 81  |................|
01bcccf0  e8 81 bc 81 00 00 00 00  03 00 00 00 00 00 00 00  |................|
01bccd00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01bccd10  80 05 56 80 00 00 00 00  00 00 00 00 00 00 00 00  |..V.............|
01bccd20  00 00 00 00 00 00 00 00  00 00 00 00 90 02 00 e1  |................|
01bccd30  30 cd bc 81 30 cd bc 81  00 00 00 00 00 00 00 00  |0...0...........|
01bccd40  00 00 00 00 53 79 73 74  65 6d 00 00 00 00 00 00  |....System......|
01bccd50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01bccd60  84 cb bc 81 34 19 a7 81  50 43 79 e1 00 00 00 00  |....4...PCy.....|
```

3. Compile

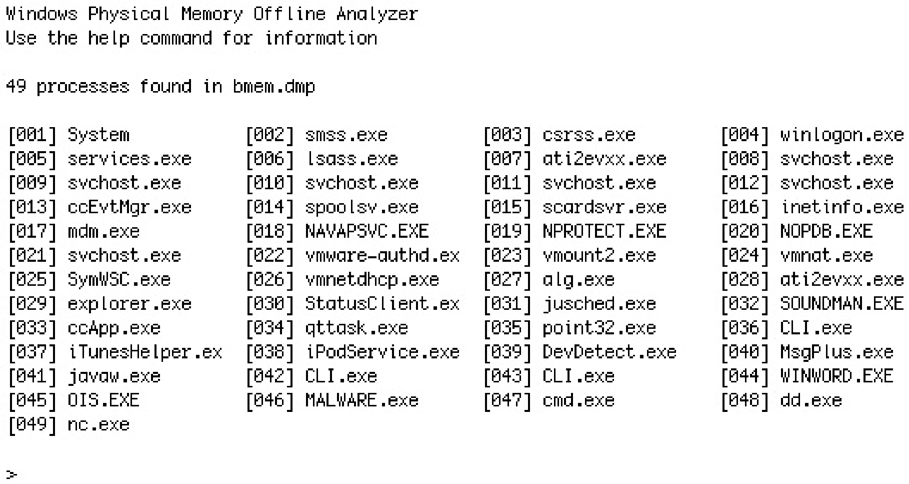
Compile the program using the make command, if not already compiled, and the program will be ready for use.

B. EXECUTION

After successful compilation, run wpmoa to begin the program. If the file name of the physical memory image is valid and the address of the PS_ACTIVE_PROCESS_HEAD correct, the program prints a greeting to the screen and displays the number of processes found in the image file. If the

PS_ACTIVE_PROCESS_HEAD was incorrect for the image being analyzed, then the program will appear to hang and must be interrupted or killed by the user to stop. After printing the number of process blocks found, the program displays a table associating a number with each process found followed by a command prompt (see Figure 17).

Figure 17. Welcome screen and process table



```
Windows Physical Memory Offline Analyzer
Use the help command for information

49 processes found in bmem.dmp

[001] System           [002] smss.exe         [003] csrss.exe        [004] winlogon.exe
[005] services.exe     [006] lsass.exe        [007] ati2evxx.exe     [008] svchost.exe
[009] svchost.exe      [010] svchost.exe      [011] svchost.exe      [012] svchost.exe
[013] ccEvtMgr.exe     [014] spoolsv.exe      [015] scardsvr.exe     [016] inetinfo.exe
[017] mdm.exe          [018] NAVAPSVC.EXE     [019] NPROTECT.EXE     [020] NOPDB.EXE
[021] svchost.exe      [022] vmware-authd.ex  [023] vmount2.exe      [024] vmnat.exe
[025] SymWSC.exe       [026] vmnetdhcp.exe    [027] alg.exe          [028] ati2evxx.exe
[029] explorer.exe     [030] StatusClient.ex  [031] jusched.exe      [032] SOUNDMAN.EXE
[033] ccApp.exe        [034] qttask.exe       [035] point32.exe      [036] CLI.exe
[037] iTunesHelper.ex  [038] iPodService.exe  [039] DevDetect.exe    [040] MsgPlus.exe
[041] javaw.exe        [042] CLI.exe          [043] CLI.exe          [044] WINWORD.EXE
[045] OIS.EXE          [046] MALWARE.exe      [047] cmd.exe          [048] dd.exe
[049] nc.exe

>
```

At this stage, the available commands are help, quit, print, and translate. Commands are not case sensitive and only the first letter is required in order to execute a command. The help command prints the help menu. The quit command ends the program. The print command, which is followed by a number 1-999 or the word *all* prints information about one of the processes or all of the processes found. This number can be found on the main menu process list table. The number corresponds to the order in which the processes occurred in the EPROCESS list (e.g. the System process always corresponds to number 1 because it is always at the head of the process list). The print command prints to a file specified by the command line argument, or standard out if no argument was given. There is no limit to the use of the print command. Any one of the processes can be printed to the same place multiples times. The output, STDOUT or a file, is updated once the print command is executed, so that the process information can be viewed with the program still running. Once the print command is finished executing, the command prompt appears again for the next user command.

The translate command provides virtual address translation to the user. Upon execution of the translate command, prompts for necessary information appear for the user to follow. The required information is the virtual address and page directory address, where the page directory address is provided in the printed report of the process of interest. If the virtual address maps to a valid physical address, then that address is printed to the screen and the user is returned to the command prompt.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. FORENSIC ANALYSIS OF PHYSICAL MEMORY DUMP

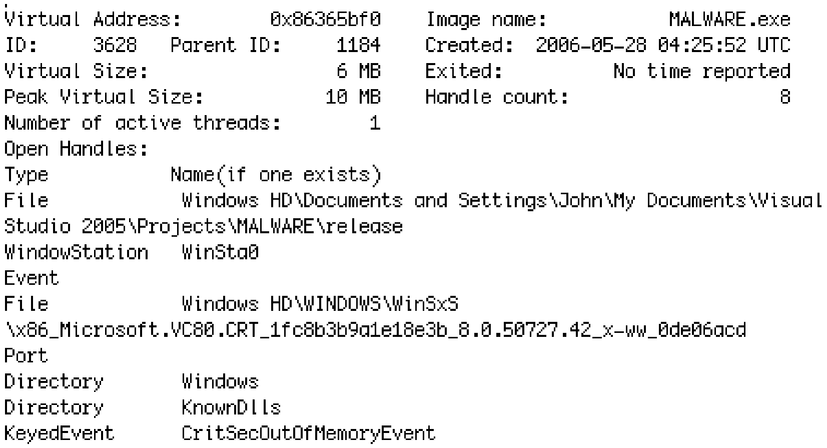
To verify the operation of the Windows Physical Memory Offline Analyzer, a test C program was run on a system running Windows XP during a live capture of the physical memory and the image was analyzed with the WPMOA program. In addition, the WPMOA program was used to analyze the physical memory images provided in the DFRWS challenge.

A. THE TEST PROGRAM

The test program was named MALWARE.exe and the source code for it is included in Appendix A. This program was intended to be very simple and not demand a lot of CPU time, but it was also desired that the program execute continually during the main memory capture. Reading input from STDIN achieved all of that with only a few lines of code. The program reads a string from STDIN and prints it to STDOUT. If the input string is "quit" then the program prints the string followed by "Hello World!" to STDOUT and then exits normally. For the test, the program was executed and left running while physical memory was captured from the target system.

B. THE FORENSIC RESULTS

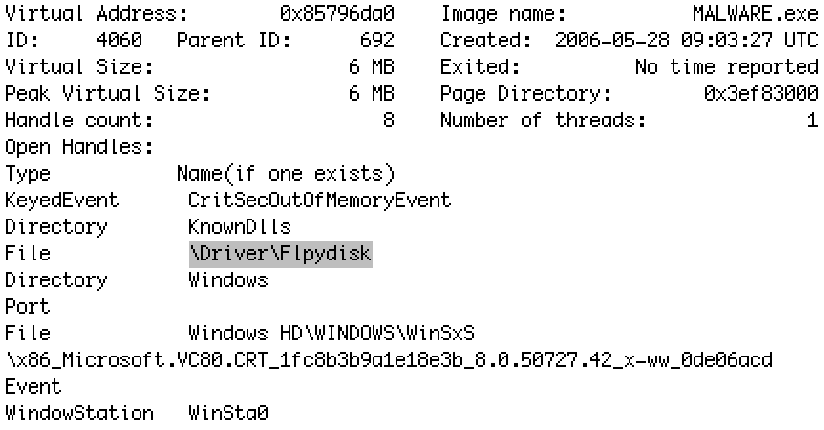
Figure 18. MALWARE.exe WPMOA report



```
Virtual Address:        0x86365bf0    Image name:            MALWARE.exe
ID:     3628   Parent ID:     1184    Created:  2006-05-28 04:25:52 UTC
Virtual Size:                 6 MB    Exited:          No time reported
Peak Virtual Size:           10 MB    Handle count:                   8
Number of active threads:        1
Open Handles:
Type           Name(if one exists)
File            Windows HD\Documents and Settings\John\My Documents\Visual
Studio 2005\Projects\MALWARE\release
WindowStation   WinSta0
Event
File            Windows HD\WINDOWS\WinSxS
\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.42_x-ww_0de06acd
Port
Directory       Windows
Directory       KnownDlls
KeyedEvent      CritSecOutOfMemoryEvent
```

The PS_ACTIVE_PROCESS_HEAD address was found in the physical memory image in accordance with the instructions from Chapter IV and the WPMOA program was compiled successfully. The forensic report of MALWARE.exe was found and printed to a file (see Figure 18). The created time correlated to the time that is was executed and the exit time does not exist because the program had not been exited. The MALWARE process reported a file handle to the directory ‘Windows HD\Documents and Settings\John\My Documents\Visual Studio 2005\Projects\MALWARE\release’, which was the base directory of MALWARE.exe. From the report, it is known that the MALWARE program was executed at 04:25 GMT on May 28, 2006 and was still running at the time of the capture. In this example, the MALWARE.exe file was executed from the system hard drive, but if it were run from a removable drive, the report would indicate that. In a test where MALWARE.exe was executed from a floppy disk, the WPMOA report reflected the new base directory of the program (see Figure 19). This information would indicate where the malicious code was executed from, which may reveal how the malicious code was transferred to the attacked system. In the report shown in Figure 19, an investigator would have learned that the source of the attack was a floppy disk and that the intruder had physical access to the floppy drive.

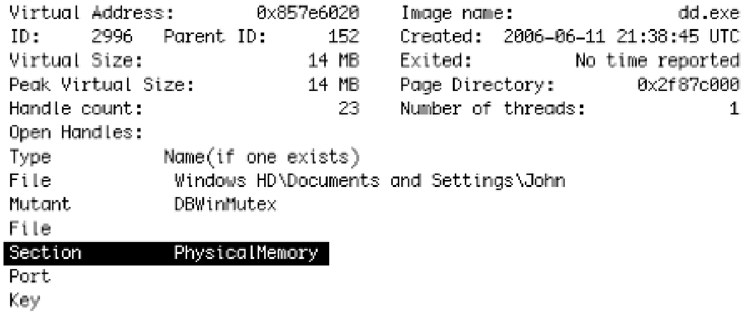
Figure 19. MALWARE.exe WPMOA report showing removable drive



```
Virtual Address:       0x85796da0    Image name:           MALWARE.exe
ID:     4060   Parent ID:      692    Created:  2006-05-28 09:03:27 UTC
Virtual Size:                 6 MB    Exited:          No time reported
Peak Virtual Size:            6 MB    Page Directory:        0x3ef83000
Handle count:                    8    Number of threads:              1
Open Handles:
Type           Name(if one exists)
KeyedEvent      CritSecOutOfMemoryEvent
Directory       KnownDlls
File            \Driver\Flpydisk
Directory       Windows
Port
File            Windows HD\WINDOWS\WinSxS
\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.42_x-ww_0de06acd
Event
WindowStation   WinSta0
```

Beyond knowing where and when the malicious code was executed, the report does not help investigators understand how the code worked or even what the actual code was. For that information, the WPMOA program creates files from the working set found in the malicious program’s EPROCESS block. In the MALWARE.exe test, the WPMOA program created thirteen files with various virtual address ranges. The virtual address range of interest was the range 0x00400000 to 0x7FFFFFFF because within this range is kept application and dynamically linked library machine code, thread stacks, and global variables. The file expected to contain the application machine code was the one named 00400000h-00405000h. The 00400000h-00405000h was compared with the MALWARE.exe file. This comparison revealed that the first 1024 bytes were identical between the two files. Because the working set pages represent sections of data referenced by threads, there are large spaces of zeroes in the 00400000h-00405000h file, where there are none in the original MALWARE.exe file. At the 4096-byte offset, the size of a page in the tested system, the 00400000h-00405000h file is identical to the next section in the MALWARE.exe for 2048 bytes. The final section of the executable file is loaded into the working set of the MALWARE.exe process similarly to the manners of the first two sections.

Figure 20. WPMOA dd.exe report



```
Virtual Address:       0x857e6020    Image name:               dd.exe
ID:     2996   Parent ID:      152    Created:  2006-06-11 21:38:45 UTC
Virtual Size:                14 MB    Exited:         No time reported
Peak Virtual Size:           14 MB    Page Directory:       0x2f87c000
Handle count:                   23    Number of threads:             1
Open Handles:
Type           Name(if one exists)
File            Windows HD\Documents and Settings\John
Mutant          DBWinMutex
File
Section         PhysicalMemory
Port
Key
```

The results from the MALWARE.exe test were ideal because all of the executable code was recovered from the process' working set. A larger program will not likely yield in such ideal results as obtained from the MALWARE test. Using the WPMOA program to generate a report on George Garner's dd.exe program demonstrates this issue. The printed report does indicate data that correlates with facts pertaining to its use for copying physical memory (see Figure 20).

In addition to the report generated by the WPMOA program, the working set files were created. It is in the working set files where the discrepancies exist. Comparing the dd.exe executable file with the 00411000h-004f1000h file generated by the WPMOA program does not give the same results as the MALWARE test. Not all of the sections of dd.exe were part of the working set at the time of the physical memory capture and must have been swapped to the system swap file.

C. RESULTS FROM DFRWS CHALLENGE

The WPMOA program was unable to analyze the physical memory images provided by the DFRWS 2005 challenge because the images were obtained from a Windows 2000 system. Windows 2000 differs from Windows XP and Windows 2003 Server enough to prevent forensic analysis by the WPMOA program. When executed, the program appears to hang and does not reach the interactive menu.

VII. FUTURE WORK

Future work for this topic includes expanding the forensic analysis of Windows virtual memory to the system swap file. Non-resident virtual pages are stored to the system swap file located on the system hard disk to create free space in main memory. With the program developed for this report, only resident pages in main memory are analyzed for forensic evidence. An executable file being executed by a process is loaded into memory by sections. If the file is very large or the working set is very small, some sections of the file may be copied to the swap file and removed from main memory. The WPMOA program would only be able to extract the resident pages containing the file content and the complete original file may not be recovered. This problem would be eliminated by combining both the physical memory image with the system swap file. Their combination yields a complete view of the virtual address space for all processes running on the system.

The current manual process of locating the head of the active process list in the memory image may intimidate users inexperienced with hexdump and grep commands. Future work could be done to automate this process to relieve the burden from the user. The instructions for manually locating the address could be translated to machine language for the computer to carry out. As 1 GB RAM capacities is not uncommon in many modern computers, the automation should be efficient to minimize the time required to locate the address in question.

Unfortunately, the results of this work could not verify the findings of the DFRWS challenge because the tool developed does not support RAM images for Windows 2000 systems. Future work could expand the compatibility of the WPMOA program to be able to analyze physical memory images obtained from Windows 2000 systems. The work could then be verified against the findings posted on the DRFWS 2005 challenge website in Ref.8.

Another area of research would be integrating the WPMOA program functionality into another forensics tool, such as Autopsy. It would be convenient to have the functionality of Autopsy include to the ability to analyze physical memory images in addition to analysis of file systems already supported.

APPENDIX A. WPMOAH

```
/**********************************************************************
Name: Windows Physical Memory Offline Analyzer
File: wpmoa.h
Version: 1.0
Author: John Schultz

Description: Console Program which enumerates the processes running on
a Windows PC. The program reads an offline copy of the physical memory
for all the information gathered about the running processes.
**********************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<strings.h>
#include<ctype.h>
#include<unistd.h>
#include<time.h>
#include"win32structs.h"

/*kernel offset of 0x80000000 set for WinXP & Win2K3 Server*/
#define KERNEL_OFFSET 0x80000000
/*value set on case by case basis see chapter IV for instructions*/
#define PS_ACTIVE_PROCESS_HEAD 0x867c6660 /* 0x81bccbd0 */
/*PAGESIZE set to the pagesize of system RAM image was taken from
  4096 bytes is typical*/
#define PAGESIZE 4096

void print_report(struct EPROCESS *[] , int , FILE * restrict );

void win_time(long long , char [] );

void unicode_to_ascii(char *, unsigned short );

int process_count(FILE * restrict );

int enumerate_processes(FILE * restrict, struct EPROCESS ** );

long virtual_to_physical(long , long , FILE * restrict );

void *enumerate_handle_table(FILE * restrict , struct EPROCESS *);

int enumerate_workingsetlist(FILE * restrict , struct EPROCESS * );

int longcmp(const void * , const void * );
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. WIN32STRUCTS.H

```
/**********************************************************************
Name Windows Physical Memory Offline Analyzer
File win32structs.h
Version 1.0
Author John Schultz

Description Console Program which enumerates the processes running on a
             Windows PC. The program reads an offline copy of the
             physical memory for all the information gathered about the
             running processes.

**********************************************************************/

#include<sys/cdefs.h>

#ifndef __WIN32STRUCTS_H
#define __WIN32STRUCTS_H

__BEGIN_DECLS

/*Each struct is defined based on that given by the Windows(R) Kernel
Debugger with few exceptions, which are noted*/
struct _unnamed {
    /* +0x000 */ long double var01;
    /* +0x00a */ long double var02;
    /* +0x014 */ long double var03;
    /* +0x01e */ long double var04;
};

struct LIST_ENTRY {
    /* +0x000 */ void *Flink;
    /* +0x004 */ void *Blink;
};

struct UNICODE_STRING {
    /* 0x000 */ unsigned short  Length;
    /* 0x002 */ unsigned short  MaximumLength;
    /* 0x004 */ unsigned short *Buffer;
};

struct STRING {
    /* 0x000 */ unsigned short Length;
    /* 0x002 */ unsigned short MaximumLength;
    /* 0x004 */ char          *Buffer;
};

struct DISPATCHER_HEADER {
    /* +0x000 */ unsigned char     Type;
    /* +0x001 */ unsigned char     Absolute;
    /* +0x002 */ unsigned char     Size;
    /* +0x003 */ unsigned char     Inserted;
    /* +0x004 */ unsigned long     SignalState;
    /* +0x008 */ struct LIST_ENTRY WaitList;
```

```
};

struct KEVENT {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
};

struct KDEVICE_QUEUE {
    /* +0x000 */ short              Type;
    /* +0x002 */ short              Size;
    /* +0x004 */ struct LIST_ENTRY  DeviceListHead;
    /* +0x00c */ unsigned long      Lock;
    /* +0x010 */ unsigned char      Busy;
};

struct KDPC {
    /* +0x000 */ short             Type;
    /* +0x002 */ unsigned char     Number;
    /* +0x003 */ unsigned char     Importance;
    /* +0x004 */ struct LIST_ENTRY DpcListEntry;
    /* +0x00c */ void             *DeferredRoutine;
    /* +0x010 */ void             *DeferredContext;
    /* +0x014 */ void             *SystemArgument1;
    /* +0x018 */ void             *SystemArgument2;
    /* +0x01c */ unsigned long    *Lock;
};

struct MMWSLE_HASH {
  /* +0x000 */ void         *Key;
  /* +0x004 */ unsigned long Index;
};

struct MMSUPPORT_FLAGS {
  /* +0x000 */ int Sessionspace : 1;
  /* +0x000 */ int BeingTrimmed : 1;
  /* +0x000 */ int SessionLoader : 1;
  /* +0x000 */ int TrimHard : 1;
  /* +0x000 */ int WorkingSetHard : 1;
  /* +0x000 */ int AddressSpaceBeingDeleted : 1;
  /* +0x000 */ int Available : 10;
  /* +0x000 */ int AllowWorkingSetAdjustment : 8;
  /* +0x000 */ int MemoryPriority : 8;
};

struct MMWSL {
  /* +0x000 */ unsigned long       Quota;
  /* +0x004 */ unsigned long       FirstFree;
  /* +0x008 */ unsigned long       FirstDynamic;
  /* +0x00c */ unsigned long       LastEntry;
  /* +0x010 */ unsigned long       NextSlot;
  /* +0x014 */ void               *Wsle; /*points to MMWSLE*/
  /* +0x018 */ unsigned long       LastInitializedWsle;
  /* +0x01c */ unsigned long       NonDirectCount;
  /* +0x020 */ struct MMWSLE_HASH *HashTable;
  /* +0x024 */ unsigned long       HashTableSize;
  /* +0x028 */ unsigned long       NumberOfCommittedPageTables;
  /* +0x02c */ void               *HashTableStart;
  /* +0x030 */ void               *HighestPermittedHashAddress;
  /* +0x034 */ unsigned long       NumberOfImageWaiters;
  /* +0x038 */ unsigned long       VadBitMapHint;
  /* +0x03c */ unsigned short      UsedPageTableEntries[768];
  /* +0x63c */ unsigned long       CommittedPageTables[24];
};

struct MMSUPPORT {
  /* +0x000 */ long long              LastTrimTime;
  /* +0x008 */ struct MMSUPPORT_FLAGS Flags;
  /* +0x00c */ unsigned long          PageFaultCount;
  /* +0x010 */ unsigned long          PeakWorkingSetSize;
  /* +0x014 */ unsigned long          WorkingSetSize;
  /* +0x018 */ unsigned long          MinimumWorkingSetSize;
  /* +0x01c */ unsigned long          MaximumWorkingSetSize;
  /* +0x020 */ struct MMWSL          *VmWorkingSetList;
  /* +0x024 */ struct LIST_ENTRY      WorkingSetExpansionLinks;
  /* +0x02c */ unsigned long          Claim;
  /* +0x030 */ unsigned long          NextEstimationSlot;
  /* +0x034 */ unsigned long          NextAgingSlot;
  /* +0x038 */ unsigned long          EstimatedAvailable;
  /* +0x03c */ unsigned long          GrowthSinceLastEstimate;
};

struct FAST_MUTEX {
    /* +0x000 */ long                     Count;
    /* +0x004 */ void                    *Owner;
    /* +0x008 */ unsigned long            Contention;
    /* +0x00c */ struct DISPATCHER_HEADER Event;
    /* +0x01c */ unsigned long            OldIrql;
};

struct CURDIR {
    /* 0x000 */ struct UNICODE_STRING DosPath;
    /* 0x008 */ void                 *Handle;
};

struct PEB_LDR_DATA {
    /* 0x000 */ unsigned long     Length;
    /* 0x004 */ unsigned char     Initialized;
    /* 0x008 */ void             *SsHandle;
    /* 0x00c */ struct LIST_ENTRY InLoadOrderModuleList;
    /* 0x014 */ struct LIST_ENTRY InMemoryOrderModuleList;
    /* 0x01c */ struct LIST_ENTRY InInitializationOrderModuleList;
    /* 0x024 */ void             *EntryInProgress;
};

struct KSEMAPHORE {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ long   Limit;
};

struct KGDTENTRY {
    /* +0x000 */ unsigned short    LimitLow;
    /* +0x002 */ unsigned short    BaseLow;
    /* +0x004 */ unsigned long     HighWord;
};
```

```
struct KIDTENTRY {
    /* +0x000 */ unsigned short    Offset;
    /* +0x002 */ unsigned short    Selector;
    /* +0x004 */ unsigned short    Access;
    /* +0x006 */ unsigned short    ExtendedOffset;
};

struct HANDLE_TABLE {
    /* +0x000 */ unsigned long     TableCode;
    /* +0x004 */ struct EPROCESS  *QuotaProcess;
    /* +0x008 */ void             *UniqueProcessID;
    /* +0x00c */ unsigned long     HandleTableLock[4];
    /* +0x01c */ struct LIST_ENTRY HandleTableList;
    /* +0x02c */ unsigned long     HandleContentionEvent;
    /* +0x030 */ void             *DebugInfo;
    /* +0x034 */ long              ExtraInfoPages;
    /* +0x038 */ unsigned long     FirstFree;
    /* +0x03c */ unsigned long     LastFree;
    /* +0x040 */ unsigned long     NextHandleNeedingPool;
    /* +0x044 */ long              HandleCount;
    /* +0x048 */ unsigned long     Flags;
};

struct OWNER_ENTRY {
    /* +0x000 */ unsigned long OwnerThread;
    /* +0x004    long          OwnerCount; */
    /* +0x004 */ unsigned long TableSize;
};

struct OBJECT_TYPE_INITIALIZER {
    /* +0x000 */ unsigned short  Length;
    /* +0x002 */ unsigned char   UseDefaultObject;
    /* +0x003 */ unsigned char   CaseInsensitive;
    /* +0x004 */ unsigned long   InvalidAttributes;
    /* +0x008 */ void           *GenericMapping;
    /* +0x018 */ unsigned long   ValidAccessMask;
    /* +0x01c */ unsigned char   SecurityRequired;
    /* +0x01d */ unsigned char   MaintainHandleCount;
    /* +0x01e */ unsigned char   MaintainTypeList;
    /* +0x020 */ void           *PoolType;
    /* +0x024 */ unsigned long   DefaultPagedPoolCharge;
    /* +0x028 */ unsigned long   DefaultNonPagedPoolCharge;
    /* +0x02c */ void           *DumpProcedure;
    /* +0x030 */ void           *OpenProcedure;
    /* +0x034 */ void           *CloseProcedure;
    /* +0x038 */ void           *DeleteProcedure;
    /* +0x03c */ void           *ParseProcedure;
    /* +0x040 */ void           *SecurityProcedure;
    /* +0x044 */ void           *QueryNameProcedure;
    /* +0x048 */ void           *OkayToCloseProcedure;
};

struct ERESOURCE {
    /* +0x000 */ struct LIST_ENTRY         SystemResourcesList;
    /* +0x008 */ struct OWNER_ENTRY       *OwnerTable;
    /* +0x00c */ short                     ActiveCount;
    /* +0x00e */ unsigned short            Flag;
    /* +0x010 */ struct KSEMAPHORE        *SharedWaiters;
    /* +0x014 */ struct DISPATCHER_HEADER *ExclusiveWaiters;
    /* +0x018 */ struct OWNER_ENTRY        OwnerThreads[2];
    /* +0x028 */ unsigned long             ContentionCount;
    /* +0x02c */ unsigned short            NumberOfSharedWaiters;
    /* +0x02e */ unsigned short            NumberOfExclusiveWaiters;
    /* +0x030    void                     *Address;*/
    /* +0x030 */ unsigned long             CreatorBackTraceIndex;
    /* +0x034 */ unsigned long             SpinLock;
};

struct SEGMENT {
    /* +0x000 */ struct CONTROL_AREA *ControlArea;
    /* +0x004 */ unsigned long        TotalNumberOfPtes;
    /* +0x008 */ unsigned long        NonExtendedPtes;
    /* +0x00c */ unsigned long        WritableUserReferences;
    /* +0x010 */ unsigned long long   SizeOfSegment;
    /* +0x018 */ long                 SegmentPteTemplate;
    /* +0x01c */ unsigned long        NumberOfCommittedPages;
    /* +0x020 */ /* struct MMEXTEND_INFO */void *ExtendInfo;
    /* +0x024 */ void                *SystemImageBase;
    /* +0x028 */ void                *BasedAddress;
    /* +0x02c */ long                 u1;
    /* +0x030 */ long                 u2;
    /* +0x034 */ long                *PrototypePte;
    /* +0x038 */ long                 ThePtes[1];
};

struct CONTROL_AREA {
    /* +0x000 */ struct SEGMENT       *Segment;
    /* +0x004 */ struct LIST_ENTRY     DereferenceList;
    /* +0x00c */ unsigned long         NumberOfSectionReferences;
    /* +0x010 */ unsigned long         NumberOfPfnReferences;
    /* +0x014 */ unsigned long         NumberOfMappedViews;
    /* +0x018 */ unsigned short        NumberOfSubsections;
    /* +0x01a */ unsigned short        FlushInProgressCount;
    /* +0x01c */ unsigned long         NumberOfUserReferences;
    /* +0x020 */ long                  unnamed;
    /* +0x024 */ struct FILE_OBJECT   *FilePointer;
    /* +0x028 */ /*struct EVENT_COUNTER*/void *WaitingForDeletion;
    /* +0x02c */ unsigned short        ModifiedWriteCount;
    /* +0x02e */ unsigned short        NumberOfSystemCacheViews;
};

struct SUBSECTION {
    /* +0x000 */ struct CONTROL_AREA  *ControlArea;
    /* +0x004 */ long                  unnamed;
    /* +0x008 */ unsigned long         StartingSector;
    /* +0x00c */ unsigned long         NumberOfFullSectors;
    /* +0x010 */ /*struct MMPTE*/void         *SubsectionBase;
    /* +0x014 */ unsigned long         UnusedPtes;
    /* +0x018 */ unsigned long         PtesInSubsection;
    /* +0x01c */ struct SUBSECTION    *NextSubsection;
};

struct SEGMENT_OBJECT {
    /* +0x000 */ void                     *BaseAddress;
    /* +0x004 */ unsigned long             TotalNumberOfPtes;
    /* +0x008 */ long long                 SizeOfSegment;
    /* +0x010 */ unsigned long             NonExtendedPtes;
    /* +0x014 */ unsigned long             ImageCommitment;
    /* +0x018 */ struct CONTROL_AREA      *ControlArea;
    /* +0x01c */ struct SUBSECTION        *Subsection;
    /* +0x020 */ /*struct LARGE_CONTROL_AREA*/void *LargeControlArea;
    /* +0x024 */ /*struct MMSECTION_FLAGS*/ void     *MmSectionFlags;
    /* +0x028 */ /*struct MMSUBSECTION_FLAGS*/ void
*MmSubSectionFlags;
};

struct SECTION_OBJECT_POINTERS {
    /* +0x000 */   void                  *DataSectionObject;
    /* +0x004 */   void                  *SharedCacheMap;
    /* +0x008 */   void                  *ImageSectionObject;
};

struct SECTION_OBJECT {
    /* +0x000 */   void                  *StartingVa;
    /* +0x004 */   void                  *EndingVa;
    /* +0x008 */   void                  *Parent;
    /* +0x00c */   void                  *LeftChild;
    /* +0x010 */   void                  *RightChild;
    /* +0x014 */   struct SEGMENT_OBJECT *Segment;
};

struct OBJECT_TYPE {
    /* +0x000 */ struct ERESOURCE                  Mutex;
    /* +0x038 */ struct LIST_ENTRY                 TypeList;
    /* +0x040 */ struct UNICODE_STRING             Name;
    /* +0x048 */ void                             *DefaultObject;
    /* +0x04c */ unsigned long                     Index;
    /* +0x050 */ unsigned long                    TotalNumberOfObjects;
    /* +0x054 */ unsigned long                    TotalNumberOfHandles;
    /* +0x058 */ unsigned long                HighWaterNumberOfObjects;
    /* +0x05c */ unsigned long                HighWaterNumberOfHandles;
    /* +0x060 */ struct OBJECT_TYPE_INITIALIZER    TypeInfo;
    /* +0x0ac */ unsigned long                     Key;
    /* +0x0b0 */ struct ERESOURCE                  ObjectLocks[4];
};

struct OBJECT_HEADER {
    /* +0x000 */ long                     PointerCount;
    /* +0x004    long                     HandleCount; */
    /* +0x004 */ void                    *NextToFree;
    /* +0x008 */ struct OBJECT_TYPE      *Type;
    /* +0x00c */ unsigned char            NameInfoOffset;
    /* +0x00d */ unsigned char            HandleInfoOffset;
    /* +0x00e */ unsigned char            QuotaInfoOffset;
    /* +0x00f */ unsigned char            Flags;
    /* +0x010 */ /* struct OBJECT_CREATE_INFORMATION *ObjectCreateInfo;
*/
    /* +0x010 */ void                    *QuotaBlockCharged;
    /* +0x014 */ void                    *SecurityDescriptor;
};
```

```
struct VPB {
    /* +0x000 */ short                        Type;
    /* +0x002 */ short                        Size;
    /* +0x004 */ unsigned short               Flags;
    /* +0x006 */ unsigned short               VolumeLabelLength;
    /* +0x008 */ struct DEVICE_OBJECT        *DeviceObject;
    /* +0x00c */ struct DEVICE_OBJECT        *RealDevice;
    /* +0x010 */ unsigned long                SerialNumber;
    /* +0x014 */ unsigned long                ReferenceCount;
    /* +0x018 */ unsigned short               VolumeLabel[32];
/*Unicode name*/
};

struct DEVICE_OBJECT {
    /* +0x000 */ short                    Type;
    /* +0x002 */ unsigned short           Size;
    /* +0x004 */ long                     ReferenceCount;
    /* +0x008 */ struct DRIVER_OBJECT    *DriverObject;
    /* +0x00c */ struct DEVICE_OBJECT    *NextDevice;
    /* +0x010 */ struct DEVICE_OBJECT    *AttachedDevice;
    /* +0x014 */ /* struct IRP */ void             *CurrentIrp;
    /* +0x018 */ /* struct IO_TIMER */ void        *Timer;
    /* +0x01c */ unsigned long            Flags;
    /* +0x020 */ unsigned long            Characteristics;
    /* +0x024 */ struct VPB              *Vpb;
    /* +0x028 */ void                    *DeviceExtension;
    /* +0x02c */ unsigned long            DeviceType;
    /* +0x030 */ char                     StackSize;
    /* +0x034 */ struct _unnamed          Queue;
    /* +0x05c */ unsigned long            AlignmentRequirement;
    /* +0x060 */ struct KDEVICE_QUEUE     DeviceQueue;
    /* +0x074 */ struct KDPC              Dpc;
    /* +0x094 */ unsigned long            ActiveThreadCount;
    /* +0x098 */ void                    *SecurityDescriptor;
    /* +0x09c */ struct KEVENT            DeviceLock;
    /* +0x0ac */ unsigned short           SectorSize;
    /* +0x0ae */ unsigned short           Spare1;
    /* +0x0b0 */ /* struct DEVOBJ_EXTENSION */ void
*DeviceObjectExtension;
    /* +0x0b4 */ void                    *Reserved;
};

struct DRIVER_OBJECT {
    /* +0x000 */ short                    Type;
    /* +0x002 */ short                    Size;
    /* +0x004 */ struct DEVICE_OBJECT    *DeviceObject;
    /* +0x008 */ unsigned long            Flags;
    /* +0x00c */ void                    *DriverStart;
    /* +0x010 */ unsigned long            DriverSize;
    /* +0x014 */ void                    *DriverSection;
    /* +0x018 */ /* struct DRIVER_EXTENSION */ void *DriverExtension;
    /* +0x01c */ struct UNICODE_STRING    DriverName;
    /* +0x024 */ struct UNICODE_STRING   *HardwareDatabase;
    /* +0x028 */ /* struct FAST_IO_DISPATCH */ void *FastIoDispatch;
    /* +0x02c */ void                    *DriverInit;
    /* +0x030 */ void                    *DriverStartIo;
    /* +0x034 */ void                    *DriverUnload;
    /* +0x038 */ void                    *MajorFunction[28];
};

struct FILE_OBJECT {
    /* +0x000 */ short                           Type;
    /* +0x002 */ short                           Size;
    /* +0x004 */ struct DEVICE_OBJECT           *DeviceObject;
    /* +0x008 */ struct VPB                     *Vpb;
    /* +0x00c */ void                           *FsContext;
    /* +0x010 */ void                           *FsContext2;
    /* +0x014 */ struct SECTION_OBJECT_POINTERS *SectionObjectPointer;
    /* +0x018 */ void                           *PrivateCacheMap;
    /* +0x01c */ long                            FinalStatus;
    /* +0x020 */ struct FILE_OBJECT             *RelatedFileObject;
    /* +0x024 */ unsigned char                   LockOperation;
    /* +0x025 */ unsigned char                   DeletePending;
    /* +0x026 */ unsigned char                   ReadAccess;
    /* +0x027 */ unsigned char                   WriteAccess;
    /* +0x028 */ unsigned char                   DeleteAccess;
    /* +0x029 */ unsigned char                   SharedRead;
    /* +0x02a */ unsigned char                   SharedWrite;
    /* +0x02b */ unsigned char                   SharedDelete;
    /* +0x02c */ unsigned long                   Flags;
    /* +0x030 */ struct UNICODE_STRING           FileName;
    /* +0x038 */ long long                       CurrentByteOffset;
    /* +0x040 */ unsigned long                   Waiters;
    /* +0x044 */ unsigned long                   Busy;
    /* +0x048 */ void                           *LastLock;
    /* +0x04c */ struct KEVENT                   Lock;
    /* +0x05c */ struct KEVENT                   Event;
    /* +0x06c */ /* struct IO_COMPLETION_CONTEXT */ void
*CompletionContext;
};

struct EX_PUSH_LOCK {
    /* +0x000 */ int           Waiting : 1; //LSB
    /* +0x000 */ int           Exclusive : 1;
    /* +0x000 */ int           Shared : 30;
    /* +0x000 */ unsigned long Value;
    /* +0x000 */ void         *Ptr;
};

struct OBJECT_DIRECTORY_ENTRY {
    /* +0x000 */ struct OBJECT_DIRECTORY_ENTRY *ChainLink;
    /* +0x004 */ void                          *Object;
};

struct OBJECT_DIRECTORY {
    /* +0x000 */ struct OBJECT_DIRECTORY_ENTRY *HashBuckets[37];
    /* +0x094 */ struct EX_PUSH_LOCK            Lock;
    /* +0x098 */ struct DEVICE_MAP             *DeviceMap;
    /* +0x09c */ unsigned long                  SessionId;
    /* +0x0a0 */ unsigned short                 Reserved;
    /* +0x0a2 */ unsigned short                 SymbolicLinkUsageCount;
};

struct DEVICE_MAP {
    /* +0x000 */ struct OBJECT_DIRECTORY       *DosDevicesDirectory;
    /* +0x004 */ struct OBJECT_DIRECTORY
*GlobalDosDevicesDirectory;
    /* +0x008 */ unsigned long                  ReferenceCount;
    /* +0x00c */ unsigned long                  DriveMap;
    /* +0x010 */ unsigned char                  DriveType[32];
};

struct RTL_DRIVE_LETTER_CURDIR {
    /* +0x000 */ unsigned short Flags;
    /* +0x002 */ unsigned short Length;
    /* +0x004 */ unsigned long  TimeStamp;
    /* +0x008 */ struct STRING  DosPath;
};

struct RTL_USER_PROCESS_PARAMETERS {
    /* +0x000 */ unsigned long                  MaximumLength;
    /* +0x004 */ unsigned long                  Length;
    /* +0x008 */ unsigned long                  Flags;
    /* +0x00c */ unsigned long                  DebugFlags;
    /* +0x010 */ void                          *ConsoleHandle;
    /* +0x014 */ unsigned long                  ConsoleFlags;
    /* +0x018 */ void                          *StandardInput;
    /* +0x01c */ void                          *StandardOutput;
    /* +0x020 */ void                          *StandardError;
    /* +0x024 */ struct CURDIR                  CurrentDirectory;
    /* +0x030 */ struct UNICODE_STRING          DllPath;
    /* +0x038 */ struct UNICODE_STRING          ImagePathName;
    /* +0x040 */ struct UNICODE_STRING          CommandLine;
    /* +0x048 */ void                          *Environment;
    /* +0x04c */ unsigned long                  StartingX;
    /* +0x050 */ unsigned long                  StartingY;
    /* +0x054 */ unsigned long                  CountX;
    /* +0x058 */ unsigned long                  CountY;
    /* +0x05c */ unsigned long                  CountCharsX;
    /* +0x060 */ unsigned long                  CountCharsY;
    /* +0x064 */ unsigned long                  FillAttribute;
    /* +0x068 */ unsigned long                  WindowFlags;
    /* +0x06c */ unsigned long                  ShowWindowFlags;
    /* +0x070 */ struct UNICODE_STRING          WindowTitle;
    /* +0x078 */ struct UNICODE_STRING          DesktopInfo;
    /* +0x080 */ struct UNICODE_STRING          ShellInfo;
    /* +0x088 */ struct UNICODE_STRING          RuntimeData;
    /* +0x090 */ struct RTL_DRIVE_LETTER_CURDIR CurrentDirectores[32];
};

struct PEB {
    /* +0x000 */ unsigned char        InheritedAddressSpace;
    /* +0x001 */ unsigned char        ReadImageFileExecOptions;
    /* +0x002 */ unsigned char        BeingDebugged;
    /* +0x003 */ unsigned char        SpareBool;
    /* +0x004 */ void                *Mutant;
    /* +0x008 */ void                *ImageBaseAddress;
    /* +0x00c */ struct PEB_LDR_DATA *Ldr;
    /* +0x010 */ struct RTL_USER_PROCESS_PARAMETERS *ProcessParameters;
    /* +0x014 */ void                *SubSystemData;
    /* +0x018 */ void                *ProcessHeap;
    /* +0x01c */ struct RTL_CRITICAL_SECTION *FastPebLock;
    /* +0x020 */ void                *FastPebLockRoutine;
    /* +0x024 */ void                *FastPebUnlockRoutine;
    /* +0x028 */ unsigned long        EnvironmentUpdateCount;
    /* +0x02c */ void                *KernelCallbackTable;
    /* +0x030 */ unsigned long        SystemReserved[1];
    /* +0x034 */ unsigned long        AtlThunkSList;
    /* +0x038 */ struct PEB_FREE_BLOCK *FreeList;
    /* +0x03c */ unsigned long        TlsExpansionCounter;
    /* +0x040 */ void                *TlsBitmap;
    /* +0x044 */ unsigned long        TlsBitmapBits[2];
    /* +0x04c */ void                *ReadOnlySharedMemoryBase;
    /* +0x050 */ void                *ReadOnlySharedMemoryHeap;
    /* +0x054 */ void               **ReadOnlyStaticServerData;
    /* +0x058 */ void                *AnsiCodePageData;
    /* +0x05c */ void                *OemCodePageData;
    /* +0x060 */ void                *UnicodeCaseTableData;
    /* +0x064 */ unsigned long        NumberOfProcessors;
    /* +0x068 */ unsigned long        NtGlobalFlag;
    /* +0x070 */ long long            CriticalSectionTimeout;
    /* +0x078 */ unsigned long        HeapSegmentReserve;
    /* +0x07c */ unsigned long        HeapSegmentCommit;
    /* +0x080 */ unsigned long        HeapDeCommitTotalFreeThreshold;
    /* +0x084 */ unsigned long        HeapDeCommitFreeBlockThreshold;
    /* +0x088 */ unsigned long        NumberOfHeaps;
    /* +0x08c */ unsigned long        MaximumNumberOfHeaps;
    /* +0x090 */ void               **ProcessHeaps;
    /* +0x094 */ void                *GdiSharedHandleTable;
    /* +0x098 */ void                *ProcessStarterHelper;
    /* +0x09c */ unsigned long        GdiDCAttributeList;
    /* +0x0a0 */ void                *LoaderLock;
    /* +0x0a4 */ unsigned long        OSMajorVersion;
    /* +0x0a8 */ unsigned long        OSMinorVersion;
    /* +0x0ac */ unsigned short      *OSBuildNumber;
    /* +0x0ae */ unsigned short      *OSCSDVersion;
    /* +0x0b0 */ unsigned long        OSPlatformId;
    /* +0x0b4 */ unsigned long        ImageSubsystem;
    /* +0x0b8 */ unsigned long        ImageSubsystemMajorVersion;
    /* +0x0bc */ unsigned long        ImageSubsystemMinorVersion;
    /* +0x0c0 */ unsigned long        ImageProcessAffinityMask;
    /* +0x0c4 */ unsigned long        GdiHandleBuffer[34];
    /* +0x14c */ void                *PostProcessInitRoutine;
    /* +0x150 */ void                *TlsExpansionBitmap;
    /* +0x154 */ unsigned long        TlsExpansionBitmapBits[32];
    /* +0x1d4 */ unsigned long        SessionId;
    /* +0x1d8 */ unsigned long long   AppCompatFlags;
    /* +0x1e0 */ unsigned long long   AppCompatFlagsUser;
    /* +0x1e8 */ void                *pShimData;
    /* +0x1ec */ void                *AppCompatInfo;
    /* +0x1f0 */ struct UNICODE_STRING CSDVersion;
    /* +0x1f8 */ void                *ActivationContextData;
    /* +0x1fc */ void                *ProcessAssemblyStorageMap;
    /* +0x200 */ void              *SystemDefaultActivationContextData;
    /* +0x204 */ void                *SystemAssemblyStorageMap;
    /* +0x208 */ unsigned long        MinimumStackCommit;
};
```

```
struct HARDWARE_PTE {
    /* +0x000 */ int Valid : 1; //LSB
    /* +0x000 */ int Write : 1;
    /* +0x000 */ int Owner : 1;
    /* +0x000 */ int WriteThrough : 1;
    /* +0x000 */ int CacheDisable : 1;
    /* +0x000 */ int Accessed : 1;
    /* +0x000 */ int Dirty : 1;
    /* +0x000 */ int LargePage : 1;
    /* +0x000 */ int Global : 1;
    /* +0x000 */ int CopyOnWrite : 1;
    /* +0x000 */ int Prototype : 1;
    /* +0x000 */ int Reserved : 1;
    /* +0x000 */ int PageFrameNumber : 20;
};

struct KPROCESS {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ struct LIST_ENTRY        ProfileListHead;
    /* +0x018 */ unsigned long            DirectoryTableBase[2];
    /* +0x020 */ struct KGDTENTRY         LdtDescriptor;
    /* +0x028 */ struct KIDTENTRY         Int21Descriptor;
    /* +0x030 */ unsigned short           IopmOffset;
    /* +0x032 */ unsigned char            Iopl;
    /* +0x033 */ unsigned char            Unused;
    /* +0x034 */ unsigned long            ActiveProcessors;
    /* +0x038 */ unsigned long            KernelTime;
    /* +0x03c */ unsigned long            UserTime;
    /* +0x040 */ struct LIST_ENTRY        ReadyListHead;
    /* +0x048 */ void                    *SwapListEntry;
    /* +0x04c */ void                    *VdmTrapcHandler;
    /* +0x050 */ struct LIST_ENTRY        ThreadListHead;
    /* +0x058 */ unsigned long            ProcessLock;
    /* +0x05c */ unsigned long            Affinity;
    /* +0x060 */ unsigned short           StackCount;
    /* +0x062 */ char                     BasePriority;
    /* +0x063 */ char                     ThreadQuantum;
    /* +0x064 */ unsigned char            AutoAlignment;
    /* +0x065 */ unsigned char            State;
    /* +0x066 */ unsigned char            ThreadSeed;
    /* +0x067 */ unsigned char            DisableBoost;
    /* +0x068 */ unsigned char            PowerState;
    /* +0x069 */ unsigned char            DisableQuantum;
    /* +0x06a */ unsigned char            IdealNode;
    /* +0x06b  KEXECUTE_OPTIONS Flags; */
    /* +0x06b */ unsigned char            ExecuteOptions;
};

struct EPROCESS {
    /* +0x000 */ struct KPROCESS      Pcb;
    /* +0x06c */ unsigned long        ProcessLock;
    /* +0x070 */ unsigned long long   CreateTime;
    /* +0x078 */ unsigned long long   ExitTime;
    /* +0x080 */ unsigned long        RunDownProtect;
    /* +0x084 */ void                *UniqueProcessId;
    /* +0x088 */ struct LIST_ENTRY    ActiveProcessLinks;
    /* +0x090 */ unsigned long        QuotaUsage[3];
    /* +0x09c */ unsigned long        QuotaPeak[3];
    /* +0x0a8 */ unsigned long        CommitCharge;
    /* +0x0ac */ unsigned long        PeakVirtualSize;
    /* +0x0b0 */ unsigned long        VirtualSize;
    /* +0x0b4 */ struct LIST_ENTRY    SessionProcessLinks;
    /* +0x0bc */ void                *DebugPort;
    /* +0x0c0 */ void                *ExceptionPort;
    /* +0x0c4 */ struct HANDLE_TABLE *ObjectTable;
    /* +0x0c8 */ unsigned long        Token;
    /* +0x0cc */ struct FAST_MUTEX    WorkingSetLock;
    /* +0x0ec */ unsigned long        WorkingSetPage;
    /* +0x0f0 */ struct FAST_MUTEX    AddressCreationLock;
    /* +0x110 */ unsigned long        HyperSpaceLock;
    /* +0x114 */ struct ETHREAD      *ForkInProgress;
    /* +0x118 */ unsigned long        HardwareTrigger;
    /* +0x11c */ void                *VadRoot;
    /* +0x120 */ void                *VadHint;
    /* +0x124 */ void                *CloneRoot;
    /* +0x128 */ unsigned long        NumberOfPrivatePages;
    /* +0x12c */ unsigned long        NumberOfLockedPages;
    /* +0x130 */ void                *Win32Process;
    /* +0x134 */ void                *Job; /*Points to EJOB*/
    /* +0x138 */ void                *SectionObject;
    /* +0x13c */ void                *SectionBaseAddress;
    /* +0x140 */ void   *QuotaBlock; /*points to EPROCESS_QUOTA_BLOCK*/
    /* +0x144 */ void  *WorkingSetWatch;/*points to PAGEFAULT_HISTORY*/
    /* +0x148 */ void                *Win32WindowStation;
    /* +0x14c */ void                *InheritedFromUniqueProcessId;
    /* +0x150 */ void                *LdtInformation;
    /* +0x154 */ void                *VadFreeHint;
    /* +0x158 */ void                *VdmObjects;
    /* +0x15c */ struct DEVICE_MAP   *DeviceMap;
    /* +0x160 */ struct LIST_ENTRY    PhysicalVadList;
    /* +0x168 */ struct HARDWARE_PTE  PageDirectoryPte;
    /* +0x16c */ unsigned long        Filler;
    /* +0x170 */ void                *Session;
    /* +0x174 */ char                 ImageFileName[16];
    /* +0x184 */ struct LIST_ENTRY    JobLinks;
    /* +0x18c */ void                *LockedPagesList;
    /* +0x190 */ struct LIST_ENTRY    ThreadListHead;
    /* +0x198 */ void                *SecurityPort;
    /* +0x19c */ void                *PaeTop;
    /* +0x1a0 */ unsigned long        ActiveThreads;
    /* +0x1a4 */ unsigned long        GrantedAccess;
    /* +0x1a8 */ unsigned long        DefaultHardErrorProcessing;
    /* +0x1ac */ long                 LastThreadExitStatus;
    /* +0x1b0 */ struct PEB          *Peb;
    /* +0x1b4 */ unsigned long        PrefetchTrace;
    /* +0x1b8 */ unsigned long long   ReadOperationCount;
    /* +0x1c0 */ unsigned long long   WriteOperationCount;
    /* +0x1c8 */ unsigned long long   OtherOperationCount;
    /* +0x1d0 */ unsigned long long   ReadTransferCount;
    /* +0x1d8 */ unsigned long long   WriteTransferCount;
    /* +0x1e0 */ unsigned long long   OtherTransferCount;
    /* +0x1e8 */ unsigned long        CommitChargeLimit;
    /* +0x1ec */ unsigned long        CommitChargePeak;
    /* +0x1f0 */ void                *AweInfo;
```

```
    /* +0x1f4 */ unsigned long        SeAuditProcessCreationInfo;
    /* +0x1f8 */ struct MMSUPPORT     Vm;
    /* +0x238 */ unsigned long        LastFaultCount;
    /* +0x23c */ unsigned long        ModifiedPageCount;
    /* +0x240 */ unsigned long        NumberOfVads;
    /* +0x244 */ unsigned long        JobStatus;
    /* +0x248 */ unsigned long        Flags;
    /* +0x24c */ long                 ExitStatus;
    /* +0x250 */ unsigned short       NextPageColor;
    /* +0x252 */ unsigned char        SubSystemMinorVersion;
    /* +0x253 */ unsigned char        SubSystemMajorVersion;
    /* +0x254 */ unsigned short       SubSystemVersion;
    /* +0x256 */ unsigned char        PriorityClass;
    /* +0x257 */ unsigned char        WorkingSetAcquiredUnsafe;
    /* +0x258 */ unsigned long        Cookie;
};

struct CLIENT_ID {
    /* +0x000 */ void *UniqueProcess;
    /* +0x004 */ void *UniqueThread;
};

struct KAPC_STATE {
    /* +0x000 */ struct LIST_ENTRY  ApcListHead[2];
    /* +0x010 */ struct KPROCESS   *Process;
    /* +0x014 */ unsigned char      KernelApcInProgress;
    /* +0x015 */ unsigned char      KernelApcPending;
    /* +0x016 */ unsigned char      UserApcPending;
      /* +0x017 */ unsigned char      Trailer;
};

struct KQUEUE {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ struct LIST_ENTRY        EntryListHead;
    /* +0x018 */ unsigned long            CurrentCount;
    /* +0x01c */ unsigned long            MaximumCount;
    /* +0x020 */ struct LIST_ENTRY        ThreadListHead;
};

struct KTIMER {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ unsigned long long       DueTime;
    /* +0x018 */ struct LIST_ENTRY        TimerListEntry;
    /* +0x020 */ struct KDPC             *Dpc;
    /* +0x024 */ long                     Period;
};

struct KAPC {
    /* +0x000 */ short                    Type;
    /* +0x002 */ short                    Size;
    /* +0x004 */ unsigned long            Spare0;
    /* +0x008 */ struct KTHREAD          *Thread;
    /* +0x00c */ struct LIST_ENTRY        ApcListEntry;
    /* +0x014 */ void                    *KernelRoutine;
    /* +0x018 */ void                    *RundownRoutine;
    /* +0x01c */ void                    *NormalRoutine;
    /* +0x020 */ void                    *NormalContext;
    /* +0x024 */ void                    *SystemArgument1;
    /* +0x028 */ void                    *SystemArgument2;
    /* +0x02c */ char                     ApcStateIndex;
    /* +0x02d */ char                     ApcMode;
    /* +0x02e */ unsigned char            Inserted;
      /* +0x02f */ char                     Trailer;
};

struct KWAIT_BLOCK {
    /* +0x000 */ struct LIST_ENTRY   WaitListEntry;
    /* +0x008 */ struct KTHREAD     *Thread;
    /* +0x00c */ void               *Object;
    /* +0x010 */ struct KWAIT_BLOCK *NextWaitBlock;
    /* +0x014 */ unsigned short      WaitKey;
    /* +0x016 */ unsigned short      WaitType;
};

struct KTHREAD {
    /* +0x000 */ struct DISPATCHER_HEADER  Header;
    /* +0x010 */ struct LIST_ENTRY         MutantListHead;
    /* +0x018 */ void                     *InitialStack;
    /* +0x01c */ void                     *StackLimit;
    /* +0x020 */ void                     *Teb;
    /* +0x024 */ void                     *TlsArray;
    /* +0x028 */ void                     *KernelStack;
    /* +0x02c */ unsigned char             DebugActive;
    /* +0x02d */ unsigned char             State;
    /* +0x02e */ unsigned char             Alerted[2];
    /* +0x030 */ unsigned char             Iopl;
    /* +0x031 */ unsigned char             NpxState;
    /* +0x032 */ char                      Saturation;
    /* +0x033 */ char                      Priority;
    /* +0x034 */ struct KAPC_STATE         ApcState;
    /* +0x04c */ unsigned long             ContextSwitches;
    /* +0x050 */ unsigned char             IdleSwapBlock;
    /* +0x051 */ unsigned char             Spare0[3];
    /* +0x054 */ long                      WaitStatus;
    /* +0x058 */ unsigned char             WaitIrql;
    /* +0x059 */ char                      WaitMode;
    /* +0x05a */ unsigned char             WaitNext;
    /* +0x05b */ unsigned char             WaitReason;
    /* +0x05c */ struct KWAIT_BLOCK       *WaitBlockList;
    /* +0x060 */ struct LIST_ENTRY         WaitListEntry;
    /* +0x068 */ unsigned long             WaitTime;
    /* +0x06c */ char                      BasePriority;
    /* +0x06d */ unsigned char             DecrementCount;
    /* +0x06e */ char                      PriorityDecrement;
    /* +0x06f */ char                      Quantum;
    /* +0x070 */ struct KWAIT_BLOCK        WaitBlock[4];
    /* +0x0d0 */ void                     *LegoData;
    /* +0x0d4 */ unsigned long             KernelApcDisable;
    /* +0x0d8 */ unsigned long             UserAffinity;
    /* +0x0dc */ unsigned char             SystemAffinityActive;
    /* +0x0dd */ unsigned char             PowerState;
    /* +0x0de */ unsigned char             NpxIrql;
    /* +0x0df */ unsigned char             InitialNode;
    /* +0x0e0 */ void                     *ServiceTable;
    /* +0x0e4 */ struct KQUEUE            *Queue;
    /* +0x0e8 */ unsigned long             ApcQueueLock;
    /* +0x0f0 */ struct KTIMER             Timer;
    /* +0x118 */ struct LIST_ENTRY         QueueListEntry;
    /* +0x120 */ unsigned long             SoftAffinity;
    /* +0x124 */ unsigned long             Affinity;
    /* +0x128 */ unsigned char             Preempted;
    /* +0x129 */ unsigned char             ProcessReadyQueue;
    /* +0x12a */ unsigned char             KernelStackResident;
    /* +0x12b */ unsigned char             NextProcessor;
    /* +0x12c */ void                     *CallbackStack;
    /* +0x130 */ void                     *Win32Thread;
    /* +0x134 */ void              *TrapFrame;/*points to KTRAP_FRAME*/
    /* +0x138 */ struct KAPC_STATE        *ApcStatePointer[2];
    /* +0x140 */ char                      PreviousMode;
    /* +0x141 */ unsigned char             EnableStackSwap;
    /* +0x142 */ unsigned char             LargeStack;
    /* +0x143 */ unsigned char             ResourceIndex;
    /* +0x144 */ unsigned long             KernelTime;
    /* +0x148 */ unsigned long             UserTime;
    /* +0x14c */ struct KAPC_STATE         SavedApcState;
    /* +0x164 */ unsigned char             Alertable;
    /* +0x165 */ unsigned char             ApcStateIndex;
    /* +0x166 */ unsigned char             ApcQueueable;
    /* +0x167 */ unsigned char             AutoAlignment;
    /* +0x168 */ void                     *StackBase;
    /* +0x16c */ struct KAPC               SuspendApc;
    /* +0x19c */ struct KSEMAPHORE         SuspendSemaphore;
    /* +0x1b0 */ struct LIST_ENTRY         ThreadListEntry;
    /* +0x1b8 */ char                      FreezeCount;
    /* +0x1b9 */ char                      SuspendCount;
    /* +0x1ba */ unsigned char             IdealProcessor;
    /* +0x1bb */ unsigned char             DisableBoost;
    /* +0x1bc */ long long                 Trailer;
};

struct ETHREAD {
    /* +0x000 */ struct KTHREAD       Tcb;
    /* +0x1c0 */ long long            CreateTime;
    /* +0x1c8 */ long long            ExitTime;
    /* +0x1d0 */ long                 ExitStatus;
    /* +0x1d4 */ struct LIST_ENTRY    PostBlockList;
    /* +0x1dc */ void                *TerminationPort;
    /* +0x1e0 */ unsigned long        ActiveTimerListLock;
    /* +0x1e4 */ struct LIST_ENTRY    ActiveTimerListHead;
    /* +0x1ec */ struct CLIENT_ID     Cid;
    /* +0x1f4 */ struct KSEMAPHORE    LpcReplySemaphore;
    /* +0x208 */ void                *LpcReplyMessage;
    /* +0x20c */ void                *ImpersonationInfo;
    /* +0x210 */ struct LIST_ENTRY    IrpList;
    /* +0x218 */ unsigned long        TopLevelIrp;
    /* +0x21c */ void                *DeviceToVerify;
    /* +0x220 */ struct EPROCESS     *ThreadsProcess;
    /* +0x224 */ void                *StartAddress;
    /* +0x228 */ void                *Win32StartAddress;
    /* +0x22c */ struct LIST_ENTRY    ThreadListEntry;
    /* +0x234 */ unsigned long        RundownProtect;
    /* +0x238 */ unsigned long        ThreadLock;
    /* +0x23c */ unsigned long        LpcReplyMessageId;
    /* +0x240 */ unsigned long        ReadClusterSize;
    /* +0x244 */ unsigned long        GrantedAccess;
    /* +0x248 */ unsigned long        CrossThreadFlags;
    /* +0x24c */ unsigned long        SameThreadPasssiveFlags;
    /* +0x250 */ unsigned long        SameThreadApcFlags;
    /* +0x254 */ unsigned char        ForwardClusterOnly;
    /* +0x255 */ unsigned char        DisablePageFaultClustering;
};

__END_DECLS

#endif
```

APPENDIX C.   WPMOA.C

```
/**********************************************************************
Name: Windows Physical Memory Offline Analyzer
File: wpmoa.c
Version: 1.0
Author: John Schultz

Description: Contains main as well as functions for running the user
interactive menus. Contains conversion functions for various data
reported by the dissector program.
**********************************************************************/

#include"wpmoa.h"

/**********************************************************************
Main opens the required i/o files and controls the main menu commands.
When the program is quit, main closes all opened i/o files and returns
0.
**********************************************************************/
int main(  int argc, char **argv ) {
    FILE * restrict imgFile = NULL;
    char command[64];
    char *action;
    char argstr[32];
    long address = 0;
    int argint = 0;
    int argint2 = 0;
    int numofProcs = 0;
    char outfile[32] = "STDOUT";
    const char welcomescreen[] =
      "Windows Physical Memory Offline Analyzer\n"
      "Use the help command for information\n";
    const char helpmenu[] =
      "Commands are caseinsensitive\n"
      "HELP - prints this menu\n"
      "PRINT ALL|1-999 - prints information about a process\n"
      "QUIT - exits program\n"
      "TRANSLATE - translate a virtual address to physical address\n";
    const char usage[] =
      "Usage: wpmoa [-f file] image\n";

    /*Open the image file, specified by last argument desired to be
analyzed*/
    FILE * restrict output = stdout; /*file handle for output, default
to stdout*/

    if( argc < 2) {
      puts(usage);
      exit(-1);
    }

    imgFile = fopen(argv[argc-1],"r");
    if( imgFile == NULL )
```

```
      fprintf(stderr,"Error: Could not open %s!\nTry
another?\n",argstr);

    for( int n=1;n<argc-1;n++ ) {
      if( strcmp(argv[n],"-f") == 0 ) {
      if( strlen(argv[n+1]) > 32 ) {
        fprintf(stderr,"Error: File name too long. Cannot exceed 32
chars. Exiting...\n");
        return -1;
      }
      strncpy(outfile,argv[n+1],sizeof(outfile));
      output = fopen(outfile, "w");
      if( output == NULL ) {
        fprintf(stderr,"Error: Could not open \'%s\'!
Exiting...\n",outfile );
        return -1;
      }
      }
    }

    /*Print the welcome screen*/
    system("clear");
    puts(welcomescreen);

    /*search the input file containing RAM image for EPROCESS blocks
and report how many were found*/
    numofProcs = process_count(imgFile);
    printf("%i processes found in %s\n\n",numofProcs,argv[argc-1]);
    struct EPROCESS *procSet[numofProcs];
    enumerate_processes(imgFile,procSet);

    /*user interaction loop - print command line prompt for the user to
      enter commands*/
    while( 1 ) {
      for( int k=0;k<numofProcs;k++) {
      printf("[%.3i] %-16s ",k+1,procSet[k]->ImageFileName);
      if( (k+1)%4 == 0 )
        puts("");
      }
      puts("\n");

      printf("> ");

      fgets(command,sizeof(command),stdin);
      action = strtok(command," \n");

      if( action == NULL )
      continue;
      else if( strncasecmp(action,"quit",1) == 0 ) {
      system("clear");
      break;
      }
      else if( strncasecmp(action,"help",1) == 0 )
      puts(helpmenu);
      else if( strncasecmp(action,"print",1) == 0 ) {
      action = strtok(NULL," \n");
      if( action == NULL )
```

```
        continue;
      strncpy(argstr,action,sizeof(argstr));
      if( strcmp(argstr,"all") == 0 ) {
        for( int k=0;k<numofProcs;k++ )
          print_report(procSet,k,output);
        printf("All processes added to %s\n\n",outfile);
        continue;
      }
      argint = atoi(argstr);
      if( argint <= numofProcs && argint > 0 ) {
        print_report(procSet,argint-1,output);
        printf("%s process added to %s\n\n",procSet[argint-1]-
>ImageFileName,outfile);
        if( enumerate_workingsetlist(imgFile,procSet[argint-1]) < 0) {
          puts("Working Set Files could not be created.");
        }
        else
          puts("Working Set File created successfully.");
      }
      }
      else if( strncasecmp(action,"translate",1) == 0 ) {
      printf("Virtual Address? (in hex) ");
      scanf("%x",&argint);
      printf("Page Directory? (in hex) ");
      scanf("%x",&argint2);
      address = virtual_to_physical(argint,argint2,imgFile);
      if( address != -1 )
        printf("Virtual address %#1.8x maps to physical address
%#1.8lx\n\n",argint,address);
      else
        printf("Virtual address %#1.8x does not map to a valid physical
address\n\n",argint);
      }
      sleep(3);
      system("clear");
      puts(welcomescreen);
    }

    /*clean up dynamic memory*/
    for(int y=0;y<numofProcs;y++) {
      free(procSet[y]);
    }

    fclose(output);
    fclose(imgFile);
    return 0;
}

/***********************************************************************
Prints a report of information found in the physical memory image which
would be useful for forensic analysis
***********************************************************************/
void print_report(struct EPROCESS *eprocess[], int i, FILE * restrict
out) {
  const char * restrict format =
    "\nVirtual Address:        0x%1.8x    Image name: %21s\n"
    "ID: %8lu   Parent ID: %8lu    Created: %24s\n"
    "Virtual Size:          %8lu MB    Exited: %25s\n"
    "Peak Virtual Size:     %8lu MB    Page Directory:
%#1.8lx\n"
    "Handle count: %20s    Number of threads:       %8lu\n"
    "Open Handles:\nType           Name(if one exists)\n"
    "%s\n";
  unsigned long eprocessAddress;
  char handleCount[16];
  char created[32];
  char exited[32];
  char notfoundmsg[] = "None Found";
  char *handles = NULL;

  /*if the current eprocess block is the first in the list, then there
is no other process in the array that points to this, so it must be
explicitly set*/
  if( strncmp(eprocess[i]->ImageFileName,"System",6) == 0 )
    eprocessAddress = PS_ACTIVE_PROCESS_HEAD;
  else
    eprocessAddress = (long)eprocess[i-1]->ActiveProcessLinks.Flink-
0x88;

  /*if the object table was not resident in physical memory, then this
fact is relayed to the user by printing Does Not Exist as the number of
handles opened by the process*/
  if( eprocess[i]->ObjectTable == NULL ) {
    strncpy(handleCount,"Nonexistent",sizeof(handleCount));
  }
  else {
    snprintf(handleCount,sizeof(handleCount),
           "%lu",eprocess[i]->ObjectTable->HandleCount);
  }

  /*convert the times the the process was created and exited*/
  win_time(eprocess[i]->CreateTime,created);
  win_time(eprocess[i]->ExitTime,exited);

  /*format the handle list for printing*/
  if( eprocess[i]->ObjectTable == NULL )
    handles = notfoundmsg;
  else
    handles = eprocess[i]->ObjectTable->DebugInfo;

  /*print out the physical memory information to output user
specified*/
  fprintf(out,format,
        eprocessAddress,
        eprocess[i]->ImageFileName,
        eprocess[i]->UniqueProcessId,
        eprocess[i]->InheritedFromUniqueProcessId,
        created,
        eprocess[i]->VirtualSize/1048576,
        exited,
        eprocess[i]->PeakVirtualSize/1048576,
        eprocess[i]->Pcb.DirectoryTableBase[0],
        handleCount,
        eprocess[i]->ActiveThreads,
        handles);

  fflush(NULL);

  return;
}


/**********************************************************************
Translates virtual address to a valid physical address, if the virtual
address is mapped to a physical address.
**********************************************************************/
long virtual_to_physical(long ptr32, long pageDirBase, FILE * restrict
in) {

    if( (ptr32 & 0xf0000000) == 0x80000000 )
      return ptr32 - KERNEL_OFFSET;

    unsigned long pageDirectoryIndex = ptr32 & 0xffc00000;
    pageDirectoryIndex = pageDirectoryIndex >> 22;
    unsigned long pageTableIndex = ptr32 & 0x003ff000;
    pageTableIndex = pageTableIndex >> 12;
    unsigned long byteIndex = ptr32 & 0x00000fff;

    ptr32 = pageDirBase + pageDirectoryIndex * 4;

    fseek(in,ptr32,0);
    fread(&ptr32,4,1,in);

    if( (ptr32 & 0x00000001) != 1 ) {
      return -1;
    }

    ptr32 = (ptr32 & 0xfffff000) + pageTableIndex * 4;

    fseek(in,ptr32,0);
    fread(&ptr32,4,1,in);

    if( (ptr32 & 0x00000001) != 1 ) {
      return -1;
    }

    ptr32 = (ptr32 & 0xfffff000) + byteIndex;

    return ptr32;
}

/**********************************************************************
Convert the Windows time format to Unix format and convert the Unix
time to GMT with the existing Linux Time library functions.

       Unix    epoch is 1970-01-01 00:00:00 resolution is seconds
       Windows epoch is 1601-01-01 00:00:00 resolution is 100ns
**********************************************************************/
void win_time(long long winTime, char date[] ) {
  long unixTime = 0;
  /*convert by subtracting the difference in time between Windows and
Unix epochs*/
  unixTime = (winTime/1e7) - 11644473600;
  /*if the time is zero or negative (as a result of the conversion for
example) then make the time zero and make it so the printout displays
that the time is "No time reported" This applies to both the create
time of System and the exit times of processes that have not exited
yet*/
  if( unixTime <= 0 ) {
    unixTime = 0;
    strncpy(date,"No time reported",32);
  }
  /*format the time and copy it into the date buffer for later
printing*/
    else
      strftime(date,32,"%Y-%m-%d %H:%M:%S %Z",gmtime(&unixTime));

  return;
}

/**********************************************************************
convert unicode string to ascii string and put it in tempString
**********************************************************************/
void unicode_to_ascii(char *string, unsigned short length) {
  if( length <= 0 )
    return;

  char *tempString = malloc(length/2);

  for(int m=0;m<length;m+=2) {
    memcpy(tempString+m/2,string+m,1);
  }
  memset(string,0,length);
  memcpy(string,tempString,length/2);

  free(tempString);
  return;
}

/**********************************************************************
Compares 2 4-byte numbers for greater than, less than, or equal to
condition
**********************************************************************/
int longcmp( const void *n1, const void *n2 ) {
  unsigned long a = *(unsigned long *)n1;
  unsigned long b = *(unsigned long *)n2;

  return (a < b) ? -1 : ((a == b) ? 0 : 1);
}
```

APPENDIX D. DISSECTOR.C

```
/*******************************************************************
Name: Windows Physical Memory Offline Analyzer
File: dissector.c
Version: 1.0
Author: John Schultz

Description: Houses all the functions that dissect the Windows physical
memory layout and structures. Interprets the raw data into human
readable information.

*******************************************************************/

#include"wpmoa.h"

#define TOTAL_TYPES 32

/*Cache the type names to avoid unnecessary file seeks and reads*/
typedef struct {
  long address;
  char name[16];
} typename;

typename typenameset[TOTAL_TYPES];
int typeCount = 0;

/*******************************************************************
Follows the active process list pointers through memory, keeping a
count of how many processes exist
*******************************************************************/
int process_count(FILE * restrict in) {
  struct EPROCESS *dummyProcList;
  int count = 0;

  /*A dummy process list is used to traverse the active process list,
counting the total number of processes that will be enumerated*/
  dummyProcList = malloc(sizeof(struct EPROCESS));

  fseek(in,PS_ACTIVE_PROCESS_HEAD-KERNEL_OFFSET,SEEK_SET);
  fread(dummyProcList,sizeof(struct EPROCESS),1,in);

  /*The active process list is traversed, meanwhile keeping count of
how many processes will be enumerated*/
  while( 1 ) {
    if( count != 0 ) {
      fseek(in,(long)(dummyProcList->ActiveProcessLinks.Flink-
KERNEL_OFFSET-0x88),SEEK_SET);
      fread(dummyProcList,sizeof(struct EPROCESS),1,in);
      if( (unsigned long)(dummyProcList->ActiveProcessLinks.Flink-0x88)
==
        PS_ACTIVE_PROCESS_HEAD )
      break;
    }
    count++;
  }

  /*clean up the dummy list*/
  free(dummyProcList);

  return count;
}

/**********************************************************************
Given the pointer to the first process in the active process list, this
function follows the list of active processes, reading and storing each
executive process block into memory (the heap).
**********************************************************************/
int enumerate_processes(FILE * restrict in, struct EPROCESS
**processList) {
  /*create a new array of pointers to EPROCESS blocks that will exactly
provide enough space for the processes to be enumerated in the
follwoing while loop*/
  processList[0] = malloc(sizeof(struct EPROCESS));

  /*read the System process block into the array*/
  fseek(in,PS_ACTIVE_PROCESS_HEAD-KERNEL_OFFSET,SEEK_SET);
  fread(processList[0],sizeof(struct EPROCESS),1,in);

  /*read and store all the executive process blocks from the RAM image
file into
    memory*/
  int n = 0;
  while( 1 ) {
    if( n != 0 ) {
      fseek(in,(long)(processList[n-1]->ActiveProcessLinks.Flink-
KERNEL_OFFSET-0x88),SEEK_SET);
      fread(processList[n],sizeof(struct EPROCESS),1,in);
      if( (unsigned long)(processList[n]->ActiveProcessLinks.Flink-
0x88) ==
        PS_ACTIVE_PROCESS_HEAD )
      break;
    }

    /* enumerate the process block's handle table */
    processList[n]->ObjectTable =
enumerate_handle_table(in,processList[n]);

    n++;
    processList[n] = malloc(sizeof(struct EPROCESS));
  }

  free(processList[n]);

  /*return number of processes found*/
  return 0;
}
/**********************************************************************
Enumerate the Object Table containing a process block's opened file
handle info
Returns the pointer to dynamic memory containing the HANDLE_TABLE
structure or
```

```
NULL is the handle table is not mapped to physical memory
**********************************************************************/
void *enumerate_handle_table(FILE * restrict in, struct EPROCESS
*eprocess) {
  long marker = 0;
  long subhandles[1024];
  long *subhandlePtr = subhandles;
  long currentObject = 0;
  long midhandles = 0;
  long pageDir = eprocess->Pcb.DirectoryTableBase[0];
  int numofHandles = 0;
  int typeID = 0;
  unsigned short length = 0;
  char *objNames = NULL;
  char readbuffer[PAGESIZE];
  char volName[32];
  struct OBJECT_HEADER *objectHdr = NULL;
  struct OBJECT_TYPE *objectType = NULL;
  struct FILE_OBJECT *fileObj = NULL;
  struct DEVICE_OBJECT *deviceObj = NULL;
  struct DRIVER_OBJECT *driverObj = NULL;
  struct VPB *vpbObj = NULL;
  struct ETHREAD *ethread = NULL;

  /*set marker variable to the physical address of the process Object
Table*/
  marker = virtual_to_physical((long)eprocess->ObjectTable,pageDir,in);

  /*if the object table is not resident in physical memory, set it to
point to NULL if it is resident, follow the link to the handle tables.
Despite the name of the HandleTable field, it does not point to the
actual table of handles. The TableCode field does. If the TableCode
field ends in 0x0, then that address is the virtual address of the
subhandle table for that process. If the field ends in 0x1, then that
address points to a mid-level handle table. The mid-level handle table
contains addresses of subhandle tables (up to a page of unique entries,
equates to 1024 entries with typical page size of 4096 bytes)*/
  if( marker > 0 ) {
    fseek(in,marker,SEEK_SET);
    eprocess->ObjectTable = malloc(sizeof(struct HANDLE_TABLE));
    fread(eprocess->ObjectTable,sizeof(struct HANDLE_TABLE),1,in);

    marker = eprocess->ObjectTable->TableCode;

    if( (marker & 0x03) == 0 ) {
      subhandles[0] = virtual_to_physical((marker &
0xfffffff8),pageDir,in);
    }
    /*when the TableCode field indicates that there are more than one
      subhandle table, store the subhandle table addresses in the array
      called subhandles*/
    else if( (marker & 0x03) == 1 ) {
      midhandles = virtual_to_physical((marker &
0xfffffff8),pageDir,in);
      marker = midhandles;
      int r = 0;
      while( r < 511 ) {
```

```
      fseek(in,marker,SEEK_SET);
      fread(subhandlePtr,4,1,in);
      if( subhandles[r] == 0 )
        break;
      subhandles[r] = virtual_to_physical((subhandles[r] &
0xfffffff8),pageDir,in);
      marker+=4;
      r++;
      subhandlePtr = &subhandles[r];
      }
    }

    /*allocate memory for a name for each open file handle of the
process*/
    numofHandles = eprocess->ObjectTable->HandleCount;
    objNames = calloc(numofHandles,128);

    /*traverse a subhandle table, then move to the next subhandle table
in the mid-level handle table until all open handles/objects have been
discovered*/
    int table = 0; int count = 0; int handle = 0;
    while( count < numofHandles && subhandles != 0 ) {

      while( count < numofHandles && handle < 511 ) {

      /*clear the file read buffer before each use*/
      memset(readbuffer,0,sizeof(readbuffer));

      /*set the marker to the address of the object to be examined*/
      fseek(in,subhandles[table]+handle*8+8,SEEK_SET);
      fread(&marker,4,1,in);

      /*if the object's address is null (meaning this spot is a free
handle), do not count this handle and move on to the next position in
the subhandle table*/
      if( marker == 0 ) {
        handle++;
        continue;
      }

      /*translate the address from virtual to physical*/
      marker = virtual_to_physical((marker & 0xfffffff8),pageDir,in);

      /*if the physical address is not mapped then count the object and
        move on to the next handle in the subhandle table*/
      if( marker == -1 ) {
        count++;
        handle++;
        continue;
      }

      /*the file location of the object pointed to by the current
subhandle position  is saved for use later in determining the object's
name*/
      currentObject = marker;
      /*allocate and read data into memory for the object header. use
the object header to determine the address of the information about the
object's type*/
      objectHdr = malloc(sizeof(struct OBJECT_HEADER));
      fseek(in,marker,SEEK_SET);
      fread(objectHdr,sizeof(struct OBJECT_HEADER),1,in);
      marker = (long)objectHdr->Type;

      /*check the object type address cache for the read type address*/
      for( typeID=0;typeID<TOTAL_TYPES;typeID++ ) {
        if( marker == typenameset[typeID].address && marker != 0 ) {
          strncpy(readbuffer,typenameset[typeID].name,16);
          objNames = strncat(objNames,readbuffer,16);
          break;
        }
      }

      /*if the type was not already in the cache, then add it*/
      if( typeID == TOTAL_TYPES ) {
        /*first add the address of the object type structure*/
        typenameset[typeCount].address = marker;
        /*convert the virtual address to physical*/
        marker = virtual_to_physical(marker,pageDir,in);
        /*if virtual address not mapped, then ignore the object type*/
        if( marker == -1 ) {
          free(objectHdr);
          count++;
          handle++;
          continue;
        }
        /*copy the object type into dynamic memory*/
        objectType = malloc(sizeof(struct OBJECT_TYPE));
        fseek(in,marker,SEEK_SET);
        fread(objectType,(sizeof(struct OBJECT_TYPE)),1,in);
        /*fetch the unicode name by its address*/
        marker = (long)objectType->Name.Buffer;
        marker = virtual_to_physical(marker,pageDir,in);

        if( marker == -1 ) {
          count++;
          handle++;
          free(objectHdr);
          free(objectType);
          continue;
        }
        /*read length of unicode name*/
        length = objectType->Name.Length;
        /*if name has a length greater than 64, then it is deemed
invalid and ignored*/
        if( length > 64 || length <= 0 ) {
          handle++;
          count++;
          free(objectHdr);
          free(objectType);
          continue;
        }
        fseek(in,marker,SEEK_SET);
        fread(readbuffer,1,length,in);
        /*convert unicode to ascii and copy it into the cache*/
        unicode_to_ascii(readbuffer,length);

        strncpy(typenameset[typeCount].name,readbuffer,16);
        typeCount++;

        objNames = strncat(objNames,readbuffer,length/2);

        free(objectType);
      }
      /*copy the object type name into the list of opened objects of
the current process*/
      objNames = strncat(objNames,"\t",1);
      /*If the object type is a process, then fetch the process name
from the object body and add it to the list of opened objects*/
      if( strncmp( readbuffer,"Process",7 ) == 0 ) {
        fseek(in,currentObject+0x18c,SEEK_SET);
        fread(readbuffer,16,1,in);
        readbuffer[16] = '\0';
        objNames = strncat(objNames,"\t",1);
        objNames = strncat(objNames,readbuffer,16);
        objNames = strncat(objNames,"(",1);
        fseek(in,currentObject+0x9c,SEEK_SET);
        fread(&marker,1,4,in);
        sprintf(readbuffer,"%lu",marker);
        objNames = strncat(objNames,readbuffer,5);
        objNames = strncat(objNames,")",1);
      }
      /*If the object type is thread, then fetch the thread's process
name and ID as well as the thread's unique ID number and add it all to
the list of opened objects*/
      else if( strncmp( readbuffer,"Thread",6 ) == 0 ) {
        ethread = malloc(sizeof(struct ETHREAD));
        fseek(in,currentObject+0x18,SEEK_SET);
        fread(ethread,sizeof(struct ETHREAD),1,in);
        marker = (long)ethread->ThreadsProcess;
        marker = virtual_to_physical(marker,pageDir,in);
        fseek(in,marker+0x174,SEEK_SET);
        fread(readbuffer,16,1,in);
        readbuffer[16] = '\0';
        objNames = strncat(objNames,"\t",1);
        objNames = strncat(objNames,readbuffer,16);
        objNames = strncat(objNames,"(",1);
        snprintf(readbuffer,5,"%li",(long)ethread->Cid.UniqueProcess);
        objNames = strncat(objNames,readbuffer,5);
        objNames = strncat(objNames,"):",2);
        snprintf(readbuffer,5,"%li",(long)ethread->Cid.UniqueThread);
        objNames = strncat(objNames,readbuffer,5);
        free(ethread);
      }
      /*If the object type is File, then fetch the name, if one exists,
        from the object body and add it to the list of opened objects*/
      else if( strncmp( readbuffer,"File",4 ) == 0 ) {
        fileObj = malloc(sizeof(struct FILE_OBJECT));
        fseek(in,currentObject+0x18,SEEK_SET);
```

```
    fread(fileObj,sizeof(struct FILE_OBJECT),1,in);
    length = fileObj->FileName.Length;
    if( length <= 2 ) {
     marker = (long)fileObj->DeviceObject;
     marker = virtual_to_physical(marker,pageDir,in);
     if( marker != -1 ) {
      deviceObj = malloc(sizeof(struct DEVICE_OBJECT));
      fseek(in,marker,SEEK_SET);
      fread(deviceObj,sizeof(struct DEVICE_OBJECT),1,in);
      marker = (long)deviceObj->DriverObject;
      marker = virtual_to_physical(marker,pageDir,in);
      if( marker != -1 ) {
      driverObj = malloc(sizeof(struct DRIVER_OBJECT));
      fseek(in,marker,SEEK_SET);
      fread(driverObj,sizeof(struct DRIVER_OBJECT),1,in);
      length = driverObj->DriverName.Length;
      marker = (long)driverObj->DriverName.Buffer;
      marker = virtual_to_physical(marker,pageDir,in);
      fseek(in,marker,SEEK_SET);
      fread(readbuffer,1,length,in);
      unicode_to_ascii(readbuffer,length);
      objNames = strncat(objNames,"\t",1);
      objNames = strncat(objNames,readbuffer,length/2);
      free(deviceObj);
      free(driverObj);
      }
     }
    }
    /*if the name length of the file is less than 512, then it is
deemed valid and will be added to the list of opened objects*/
    else if(length < 512) {
     marker = (long)fileObj->Vpb;
     marker = virtual_to_physical(marker,pageDir,in);
     if( marker != -1 ) {
      vpbObj = malloc(sizeof(struct VPB));
      fseek(in,marker,SEEK_SET);
      fread(vpbObj,sizeof(struct VPB),1,in);
      memcpy(volName,vpbObj->VolumeLabel,32);
      free(vpbObj);
      unicode_to_ascii(volName,32);
      objNames = strncat(objNames,"\t",1);
      objNames = strncat(objNames,volName,16);

      marker = (long)fileObj->FileName.Buffer;
      marker = virtual_to_physical(marker,pageDir,in);
      fseek(in,marker,SEEK_SET);
      fread(readbuffer,1,length,in);
      unicode_to_ascii(readbuffer,length);
      objNames = strncat(objNames,readbuffer,length/2);
     }
    }
    free(fileObj);
   }
   /*If the object type is anything other than the ones previously
    mentioned then use the default method to fetch the object name
    from the object name structure and add it to the list of open
    objects*/
```

```
      else {
        if( strlen( readbuffer ) < 8 )
          objNames = strncat(objNames,"\t",1);
        fseek(in,currentObject-0xc,SEEK_SET);
        fread(&length,2,1,in);
        fseek(in,currentObject-0x8,SEEK_SET);
        fread(&marker,4,1,in);
        marker = virtual_to_physical(marker,pageDir,in);
        if( marker != -1 && length > 0 && length < 2048) {
          memset(readbuffer,0,sizeof(readbuffer));
          fseek(in,marker,SEEK_SET);
          fread(readbuffer,1,length,in);
          unicode_to_ascii(readbuffer,length);
          objNames = strncat(objNames,readbuffer,length/2);
        }
      }
      objNames = strncat(objNames,"\n",1);

      free(objectHdr);
      handle++;
      count++;
      }
      table++;
      handle = 0;
    }
    eprocess->ObjectTable->DebugInfo = objNames;
  }
  else
    eprocess->ObjectTable = NULL;

  return eprocess->ObjectTable;
}

/**********************************************************************
Function enumerates a process' working set and creates files with the
contents of the working set pages in memory.
**********************************************************************/
int enumerate_workingsetlist(FILE * restrict in, struct EPROCESS
*eprocess) {
  char pname[24];
  char processID[8];
  char vaddress[24];
  char fname[56];
  char fname2[64];
  char directory[24] = "mkdir ";
  char deldir[24] = "rm -rf ";
  int i = 0;
  long marker = 0;
  long pageDir = 0;
  long pid = 0;
  unsigned int n = 0;
  unsigned int wsIndex = 0;
  unsigned int wsSize = 0;
  unsigned int wsCount = 0;
  unsigned long wsEnd = 0;
  unsigned long buffer[1024];
  unsigned long *wsList = NULL;
  unsigned long start = 0;
  FILE *fd = NULL;
  struct MMWSL *wsl = NULL;

  pageDir = eprocess->Pcb.DirectoryTableBase[0];
  wsSize = eprocess->Vm.WorkingSetSize;
  wsList = malloc(wsSize*4);

  /*find address of the workingsetlist structure*/
  marker = (long)eprocess->Vm.VmWorkingSetList;
  marker = virtual_to_physical(marker,pageDir,in);
  if( marker < 0 )
    return -1;

  /*read the contents of the workingsetlist structure into dynamic
    memory*/
  wsl = malloc(sizeof(struct MMWSL));
  fseek(in,marker,SEEK_SET);
  fread(wsl,1,sizeof(struct MMWSL),in);

  wsEnd = wsl->LastInitializedWsle;

  marker = (long)wsl->Wsle;
  marker = virtual_to_physical(marker,pageDir,in);
  if( marker < 0 ) {
    free(wsl);
    return -1;
  }

  fseek(in,marker,SEEK_SET);

  memset(wsList,0,sizeof(wsList));

  wsCount = 0;
  while( wsCount < wsSize && wsIndex < wsEnd ) {
    i = -1;
    fread(buffer,4,sizeof(buffer)/4,in);
    while( wsCount < wsSize && i < 1024 ) {
      i++; wsIndex++;

      if( virtual_to_physical(buffer[i],pageDir,in) < 0 )
      continue;

      wsList[wsCount] = (buffer[i] & 0xfffff000);
      wsCount++;
    }
    fseek(in,4096,SEEK_CUR);
  }

  /*sort the list of virtual addresses in the working set from low to
    high order*/
  qsort(wsList,wsCount,4,longcmp);

  /*Pool together consecutive memory locations and create files from
the data in those areas */
  pid = (long)eprocess->UniqueProcessId;
  snprintf(processID,sizeof(processID),"\x5f%lu",pid);
  strncpy(pname,eprocess->ImageFileName,sizeof(pname));
  strncat(pname,processID,sizeof(pname));
  strncat(directory,pname,sizeof(directory)-strlen(directory));
  if( fopen(pname,"r") != NULL ) {
    strncat(deldir,pname,sizeof(deldir)-strlen(deldir));
    system(deldir);
  }
  system(directory);

  n = 1;

  puts("Creating Working Set Files, Please Wait");

  while( n < wsCount ) {

    if( wsList[n] - wsList[n-1] == 0x1000 ) {
      if( start == 0 )
      start = wsList[n-1];
      snprintf(vaddress,11,"/%1.8lxh",start);
      strncpy(fname,pname,sizeof(fname));
      strncat(fname,vaddress,sizeof(fname)-strlen(fname));
      fd = fopen(fname,"w");
      memcpy(buffer,"\x41\x42\x43\x44",4);
      if( fd == NULL )
      return -1;
      while( ( wsList[n] & 0xfff00000) == (start & 0xfff00000) ) {
      marker = virtual_to_physical(wsList[n-1],pageDir,in);
      fseek(in,marker,SEEK_SET);
      fread(buffer,1,sizeof(buffer),in);
      fwrite(buffer,1,sizeof(buffer),fd);
      n++;
      }
      strncpy(fname2,fname,sizeof(fname2));
      strncat(fname2,"-",1);
      snprintf(vaddress,10,"%1.8lxh",wsList[n-1]+4096);
      strncat(fname2,vaddress,sizeof(fname2)-strlen(fname2));
      rename(fname,fname2);
      fclose(fd);
      start = 0;
      if( memcmp(buffer,"\x41\x42\x43\x44",4) == 0 )
      n++;
    }
    else
      n++;
  }

  free(wsl);
  return 0;
}
```

APPENDIX E. MALWARE.C

```
// MALWARE.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
      char string[32];

      while( 1 ) {
            scanf_s("%s",string);
            printf("%s\n",string);
            if( strcmp(string,"quit") == 0 )
                  break;
      }

      puts("Hello World!");

      return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Michelle Finley. “Now *That* Was a Nasty Worm.” Internet: www.wired.com/news/technology/1,36119-0.html, May 4, 2000 [Jun. 15, 2006].

2. Robert Lemos. “"Code Red" worm claims 12,000 servers,” Internet: news.com.com/2100-1001-270170.html, Jul. 18, 2001 [Jun. 15, 2006].

3. “MSBlast echoes across the Net.” Internet: news.com.com/2009-1002_3-5063226.html, Jul. 16, 2003 [Jun. 15, 2006].

4. Ed Skoudis. “Powerful payloads: The evolution of exploit frameworks.” Internet: www.coresecurity.com/common/showdoc.php?idx=490&idxseccion=12&idxmenu=32, Oct. 20, 2005 [Apr. 24, 2006].

5. Eric Chien. “W32.Witty.Worm” Internet: securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html, Mar. 22, 2004 [Apr. 24, 2006].

6. Michael G. Noblett, Mark M. Pollitt, and Lawrence A. Presley. “Recovering and Examining Computer Forensic Evidence.” *Forensic Science Communications*, Vol. 2, No. 4, Oct. 2000.

7. Chris Prosise, Kevin Mandia, and Matt Pepe. *Incident Response and Computer Forensics, Second Edition*. McGraw-Hill Osborne Media, 2003.

8. “DRFWS 2005 Forensic Challenge.” Internet: www.dfrws.org/2005/challenge/index.html [May 3, 2006].

9. Brian D. Carrier and Joe Grand. “A Hardware-Based Memory Acquisition Procedure for Digital Investigations.” *Digital Investigation Journal*, Issue 1(1), Feb 2004.

10. George M. Garner. “Forensic Acquisition Tools.” Internet: users.erols.com/gmgarner/forensics/, Aug. 17, 2004 [Jun. 15, 2006].

11. *GNU Coreutils Manual*. Internet: http://www.gnu.org/software/coreutils/manual/html_mono/coreutils.html#Basic-operations [Jun. 29, 2006].

12. “DLLs, Processes, and Threads.” *MSDN Library*. Internet: msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/exitprocess.asp?frame=true [Jun. 14, 2006].

13. Mark E. Russinovich and David A. Solomon. *Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000*. Redmond, Washington: Microsoft Press, 2005.
14. Peter J. Denning. “The Working Set Model For Program Behavior.” *Communications of the ACM*, Vol. 11, Issue 5, (1968).

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. John Schultz
   Naval Postgraduate School
   Monterey, California

4. George Dinolt
   Naval Postgraduate School
   Monterey, California

5. Chris Eagle
   Naval Postgraduate School
   Monterey, California