NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF LINUX RAM FORENSICS

by

Jorge Mario Urrea

March 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
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Analysis of Linux RAM Forensics			**5. FUNDING NUMBERS**
6. AUTHOR(S) Urrea, Jorge Mario			
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**

13. ABSTRACT (maximum 200 words)

During a forensic investigation of a computer system, the ability to retrieve volatile information can be of critical importance. The contents of RAM could reveal malicious code running on the system that has been deleted from the hard drive or, better yet, that was never resident on the hard drive at all. RAM can also provide the programs most recently run and files most recently opened in the system. However, due to the nature of modern operating systems, these programs and files are not typically stored contiguously—which makes most retrieval efforts of files larger than one page size futile. To date, analysis of RAM images has been largely restricted to searching for ASCII string content, which typically only yields text information such as document fragments, passwords or scripts.

This thesis explores the memory management structures in a SUSE Linux system (kernel version 2.6.13-15) to make sense out of the chaos in RAM and facilitate the retrieval of files/programs larger than one page size. The analysis includes methods for incorporating swap space information for files that may not reside completely within physical memory.

The results of this thesis will become the basis of later research efforts in RAM forensics. This includes the creation of tools that will provide forensic analysts with a clear map of what is resident in the volatile memory of a system.

14. SUBJECT TERMS Computer Forensics, Physical Memory Analysis, RAM Analysis, Volatile Memory			**15. NUMBER OF PAGES** 89
			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

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ANALYSIS OF LINUX RAM FORENSICS

Jorge M. Urrea
Civilian, Federal Cyber Corps
B.S., California Polytechnic State University, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2006

Author: Jorge Mario Urrea

Approved by: Christopher S. Eagle
Thesis Advisor

George Dinolt
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

During a forensic investigation of a computer system, the ability to retrieve volatile information can be of critical importance. The contents of RAM could reveal malicious code running on the system that has been deleted from the hard drive or, better yet, that was never resident on the hard drive at all. RAM can also provide the programs most recently run and files most recently opened in the system. However, due to the nature of modern operating systems, these programs and files are not typically stored contiguously—which makes most retrieval efforts of files larger than one page size futile. To date, analysis of RAM images has been largely restricted to searching for ASCII string content, which typically only yields text information such as document fragments, passwords or scripts.

This thesis explores the memory management structures in a SUSE Linux system (kernel version 2.6.13-15) to make sense out of the chaos in RAM and facilitate the retrieval of files/programs larger than one page size. The analysis includes methods for incorporating swap space information for files that may not reside completely within physical memory.

The results of this thesis will become the basis of later research efforts in RAM forensics. This includes the creation of tools that will provide forensic analysts with a clear map of what is resident in the volatile memory of a system.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

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, Retired for providing the idea for this thesis and for discussing problems and offering great advice throughout the whole process.

I would also like to thank Dr. George Dinolt for his input and constructive criticism because it helped improve my technical writing and, therefore, any reader's comprehension of this thesis.

This material is based upon work supported by the National Science Foundation under Grant No DUE-0114018 and by the Office of Naval Research. I would like to thank the National Science Foundation and the Office of Naval Research for their contributions. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or of the Office of Naval Research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Computer-aided crime has been a significant problem for industry in recent years. The FBI estimated the financial losses related to computer incidents in the United States at 67.2 billion for the year 2005 [Ref. 10]. These incidents included (but were not limited to): viruses, worms, financial fraud, network intrusion, and the sabotage of data or networks. Computer crime has also posed a threat to national security. Credit card information stolen from compromised systems might have been used to fund terrorist activities [Ref. 11]. As criminals become more adept at breaking laws through the use of computers, law enforcement agents must hone their ability to investigate these types of cases. Computer Forensics can be used to establish who committed the crime and to reconstruct how the crime was executed.

One specific branch of forensics gaining momentum concerns itself with RAM analysis. Traditionally, when a forensic investigation is performed on a computer of interest, one of the first things done is to gather any volatile information that can be gleamed from the victim system. Sometimes this includes making a copy of the system's RAM content, which is analyzed with simple searches for ASCII or Unicode string content because few tools exist and few people are trained to perform a more in-depth analysis of the memory dump [Ref. 1].

Some of the key pieces of volatile information that a forensic analyst is looking for are the currently running processes of a system and the files most recently used. An educated investigation of a RAM dump could yield this information. Some might ask why it is a good idea to use such a technique if there are programs available (such as ps[1]) that will enumerate running processes. The answer is that these programs can be subverted if the system they are running on is compromised with a loadable kernel module rootkit—a piece of malware that can manipulate the execution of system commands [Ref. 1]. In addition, advanced malware techniques allow for the injection of

[1] The program ps lists currently running processes in a Linux system.

malicious code directly into running processes such that no new process is visible to standard tools. A forensic exploration of physical memory can look at kernel structures directly and, consequently see through any such deceptions.

Recent worms, including SQL Slammer and Code Red, do not write any data to disk [Ref. 12]. All data remains in physical memory. This renders standard disk forensics useless and becomes yet another reason why thoroughly inspecting RAM is a growing necessity. It may be the only way to directly detect the presence of malware and give an investigator an opportunity to retrieve full and accurate information from a compromised system.

A. RAM FORENSICS BACKGROUND

The field of computer forensics is young. The FBI created a Computer Analysis and Response team (CART) in 1984—which did not become fully functional until 1991—to supplement its well-established investigation protocols for terrorism and violent crime [Ref. 2]. Since then, other public and private organizations have followed suit and now, twenty-one years later, forensics is beginning to take shape.

Within the forensics community, a large share of attention has been paid to analyzing non-volatile media such as hard drives or storage peripherals. More recently the rise of networks has created an interest in the study of network-based evidence as well. Both of these subjects have existing, extensive bodies of knowledge—as exemplified by [Ref. 1] and [Ref. 7]. This is not the case for RAM analysis. The analysis of volatile memory is such a young area, in fact, that one is hard pressed to find more than one paper directly addressing analysis of Linux RAM contents. As an example of the lack of attention to this critical need, the popular book *Incident Response & Computer Forensics* [Ref. 1] devotes 7 lines of coverage to RAM analysis in a twenty-two-page chapter devoted to live data collection from Unix systems.

RAM analysis, like all other forensic endeavors, is concerned with the retrieval of information that can serve as evidence in criminal investigations. More specifically, it is the attempt to use memory management structures in computers as maps to extract files and executables resident in a computer's physical memory. These files/executables can be used to prove that a crime has transpired or to trace how it came to pass. The

usefulness of this type of investigation lies in the fact that any information found in RAM is known to have been recently running on the victim system. Additionally, volatile memory examination can stand up to conventional attempts at thwarting forensic efforts—such as function hooking [Ref. 3]—which is a way to attach a chosen function to the normal flow of control in a computer system. For example, if a rootkit has hooked itself into the Linux kernel and is intercepting calls to ps, it can exclude whatever process it wants to hide from the returned list of processes.

B. PURPOSE OF STUDY

The immediate purpose of this research is to discover what forensic techniques can be used effectively on the physical memory of a SUSE Linux system running the new 2.6 kernel. Some techniques for volatile memory forensics have been developed for the 2.4 kernel [Ref. 3] but they have not yet been tested in the new version of the Linux kernel—in which some of the fundamental structures involved in memory management have been modified.

The more general goal of this research is to improve the methods of analyzing RAM dumps. Currently, the typical way to analyze physical memory on a computer is to run a string search on the entire memory image in the hopes of finding information such as passwords, the cleartext of a recently typed encrypted message, or the contents of a file [Ref. 1]. Unfortunately, during this type of search, valuable context information is lost. For example, it becomes impossible to determine whether recovered string fragments represent the contents of executable files, data files, or runtime program data. This is an unsophisticated “stab in the dark” type of analysis that can only yield a small amount of useful information—an unfortunate result when the contents of physical memory are a rich source of forensic evidence. As criminals become more adept at creating malware that can elude current methods of digital forensic investigation, forensics methods must evolve to meet the challenge. When the author of a piece of malware decides to design it to reside exclusively in physical memory—and thereby evade any hard drive investigation—the forensic analyst must have a way to detect it. The goal of this research is to provide the basis for the development of tools that the forensic analyst can use in a detailed analysis of Linux kernel 2.6 memory images.

C. THESIS ORGANIZATION

This paper will present forensic techniques, including a few proof-of-concept Perl scripts, which will facilitate the development of a tool that will be able to extract files and executables stored in a computer's physical memory. Chapter II will detail the structures that are involved in Linux memory management and their relationship. This chapter will also outline some of the essential calculations and concepts that play a part in the analysis of RAM images later in the paper. Chapter III will discuss the current state of RAM forensics. Chapter IV will provide a description of the analysis performed on a SUSE Linux system running the 2.6.13-15 kernel.

The analysis consists of two parts. The first is placing specific files in memory, imaging the memory, and seeing if the file in question can be retrieved. The second is to see what other useful information can be extracted from the memory image using data from swap space. The main thrust of Chapter IV is to verify whether forensic techniques developed for the 2.4 Kernel [Ref. 3] are applicable to the 2.6 Kernel and to see if any supplementary techniques can be discovered for this new Kernel. Chapter V will summarize the results of this thesis and problems encountered along the way. Additionally, the chapter will describe what future work can be performed in the field of RAM forensics.

II. BACKGROUND

This section describes all of the major components of the Linux virtual memory management system. A short description of each component is provided—as are the locations of the corresponding definitions in the Linux source code. The base directory location assumed throughout is /usr/src/Linux-2.6.13-15. This was the default location created by the SUSE 10 distribution used in this thesis, which can be downloaded at the following url: http://en.opensuse.org/Welcome_to_openSUSE.org. Please note that depending on the version of the Linux kernel used, the version number will vary.

A. PARAMETERS OF INVESTIGATION

The research conducted in this paper was performed on a SUSE 10 system running Linux kernel version 2.6.13-15. The system resided in a Dell Dimension 4400 computer with 512 MB of RAM and an Intel Pentium 4 running at 2.0 GHz. All of the programming was done in Perl version 5.8.7—except for a small c program used to aid in the analysis of swap space. The amount of RAM was deliberately chosen to ease the burden of translating from virtual to physical memory addresses by avoiding high memory (> 896 MB) translations. Use of high memory would not allow for the simple memory conversion scheme outlined in section C of this chapter. Additional tools used during the course of this research are as outlined below.

1. Source Navigator 5.1.4

This program was run on Mac OS X 10.4.3 through X11. It was downloaded using DarwinPorts (http://darwinports.opendarwin.org/getdp). Source Navigator is a source code analysis tool that provided a quick way to search through the sizable kernel code for items of interest.

2. Gcalc 5.6.31

This is a calculator included in SUSE 10. It was used for converting decimal numbers to hexadecimal numbers and vice versa. It was also used for arithmetic in both bases. These calculations were useful when adding offsets to the beginning of a struct to find certain values and pointers within that struct.

3. DD (Part of Coreutils 5.3.0)

This tool version was included in SUSE 10. In the simplest of terms, it copies data from one place to another. In the context of this thesis, it was used to create all the images of RAM and swap. The full documentation for dd can be found at http://www.hmug.org/man/1/dd.php.

4. GHex 2.8.1

This hex editor is included in SUSE 10. All the analysis of images created for this thesis was done using this tool. Substantial image files, as large as 400 MB, slowed the program down while being opened, but GHex responded quickly after that initial step.

5. Memory (Part of KDE 3.4.2 Level "b")

This tool is included in SUSE 10 as part of the KDE installation and can be run with the command "kcmshell memory" from a terminal window. It was used as a way to assure that data was being written to the swap space in response to the large memory demands made by heap.c [Appendix A].

6. Kate Editor 2.4.1

This editor is included in SUSE 10. It was used to write all of the code involved in this thesis. A helpful feature of this editor was that it allows a user to open multiple files and navigate back and forth through them in a single window with a forward and back arrow like a web browser.

B. MEMORY MANAGEMENT STRUCTURES

1. The Big Picture

Linux implements virtual memory management through a series of C data structures that are interrelated as shown in Figure 1 [Ref. 4].

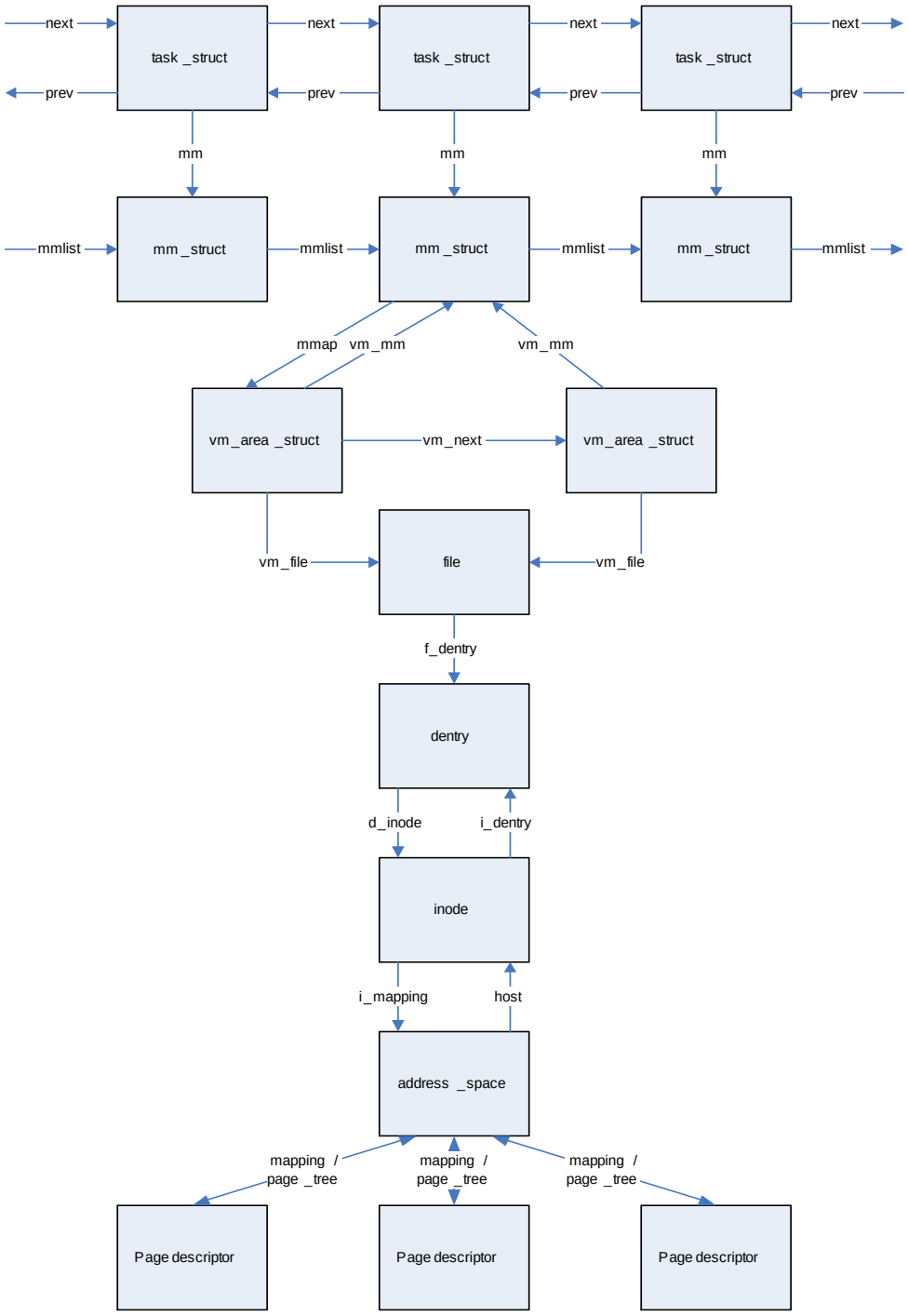

Figure 1. Overview of Memory Management Structures. (From Ref. 3)

2. Task Struct

The kernel creates a task_struct for every process running on a computer at any given time. It is defined in /usr/src/linux-2.6.13-15/linux/sched.h and holds a wealth of information about the current state of each process [Ref. 3]. For the purposes of this

thesis, the relevant information contained in each task_struct are the following: a pointer to the mm_struct, the process id (pid) of the program, and the executable name. The reader should note that kernel threads have NULL mm_struct pointers because they do not need this kernel structure to exist [Ref. 6]. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 1. These offsets allow an investigator to find the location of the elements easily when viewing the struct in a hex dump. A developer can also use these offsets to create a program that parses this struct.

Data Type	Element	Offset
struct mm_struct*	mm	0x78
pid_t	pid	0x9C
char[]	comm (executable name)	0x1A4

Table 1. Elements of task_struct

3. MM Struct

This structure describes a process' address space and, as such, there is only one mm_struct per process. In the case of userspace threads, one mm_struct is shared by all of them. The pgd member of this struct holds a pointer to the page global directory of the process. This is the pathway to all the pages owned by a process—even those that are in swap space. The mm-struct also outlines the start and end addresses of memory sections such as code, data, and the heap. It is defined in /usr/src/linux-2.6.13-15/linux/sched.h. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 2.

Data Type	Element	Offset
struct vm_area_struct*	mmap	0x00
pgd_t*	pgd	0x20

Table 2. Elements of mm_struct

4. VM Area Struct

A process typically only uses small portions of the memory allocated to it and, in order to reconcile this fact, the vm_area_struct is used. It contains a start and end address

for the memory area it describes as well as the access permissions for that area. An example of such a memory region is a read-only library loaded into the address space or the process heap [Ref. 6]. The vm_struct is defined in /usr/src/linux-2.6.13-15/linux/mm.h. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 3.

Data Type	Element	Offset
struct mm_struct*	vm_mm	0x00
struct file*	vm_file	0x4C
struct	vm_set (where i_mmap pointer from address_space points to)	0x28

Table 3. Elements of vm_area_struct

5. File Struct

If a memory region represents a file, then the vm_file field of the corresponding vm_area_struct will point to a file struct. The file struct is outlined at /usr/src/linux-2.6.13-15/linux/fs.h. It contains information about the interaction between a process and an open file and is only in kernel memory while a process accesses the file in question [Ref. 4]. This struct includes a pointer to the file operations available to that particular process and a pointer to the filesystem containing the file. Other interesting data that can be gleamed from this structure is the user's UID and the file pointer [Ref. 4]. The offset of the element used during this research—measured from the beginning of the struct—is shown in Table 4.

Data Types	Element	Offset
struct dentry*	f_dentry	0x08

Table 4. Elements of File Struct

6. Dentry Struct

These structures are created by the Linux virtual file system for every directory used in memory [Ref. 4]. For instance, if a user looks up the file "document.txt" in the path /home/Ted, then three dentry objects will be created: one for / (root), one for home, and one for Ted. The declaration of dentry structure can be found in /usr/src/linux-

2.6.13-15/linux/dcache.h. It holds information like the name of its related file. A pointer to the name is held in the d_name element—which usually points to the d_iname element further down in the dentry struct if the name is short enough. If the name is long enough, the d_name element will point to a location with sufficient space. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 5.

Data Type	Element	Offset
struct inode*	d_inode	0x08
unsigned char	d_iname (name of assoc. file)	0x64

Table 5. Elements of Dentry Struct

7. Inode Struct

All the information required by the filesystem to manipulate a file can be found in this structure [Ref. 4]. The inode definition can be found in /usr/src/linux-2.6.13-15/linux/fs.h. The information in this struct includes: the number of the inode, the uid of the file, the length of the file in bytes, the last access time, the last write time, and the last inode change time. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 6.

Data Type	Element	Offset
struct address_space*	i_mapping	0x9C

Table 6. Elements of Inode Struct

8. Address Space Struct

The address_space struct can be found at /usr/src/linux-2.6.13-15/linux/fs.h and is created by the kernel for every memory mapped file [Ref. 3]. Some of the information of interest here is a radix tree with links to all the pages belonging to this particular file (page_tree element) and the total number of pages owned by the file. In Kernel 2.4 it used to hold doubly linked lists of clean, dirty, and locked pages. However, none of these lists exist in kernel 2.6. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 7.

Data Type	Element	Offset
struct inode*	host	0x00
struct radix_tree _root	page_tree	0x04
struct prio_tree_root	i_mmap (pointer to vm_area_struct)	0x14
unsigned long	nrpages (number of pages owned	0x28

Table 7. Elements of address_space Struct

9. Page Struct

This is the page descriptor structure that holds information relevant to individual pages of a file. The information found here that is of most relevance for this thesis is the index element which holds the offset within the mapping of the file. For instance, if it is the second page of a file, the index field will hold the number one (because the indices begin at 0). In the 2.4 kernel, the virtual field held the virtual address of the page being described. In kernel 2.6, this is an optional field that is only required when a computer has high memory—more than 896 MB of RAM—to hold the dynamically assigned virtual address. The offsets of the elements used during this research—measured from the beginning of this struct—are shown in Table 8.

Data Types	Element	Offset
struct address_space*	mapping	0x10
pgoff_t	index (offset within mapping)	0x14

Table 8. Elements of Page Struct

10. Memory Map

All page descriptors are part of the mem_map array, which is typically stored at address 0x1000000 [Ref. 6]. The size of each page descriptor on the system used in this paper is 0x20 bytes so, by beginning at 0x1000000 and incrementing by 0x20, it is possible to traverse the entirety of this array from one page descriptor to another.

C. USEFUL CALCULATIONS

1. Virtual to Physical Memory Conversion for Kernel Addresses

All pointers encountered in the structures described above will contain virtual addresses and, since this paper examines physical memory dumps, these addresses must be converted to their physical equivalent. For example, if an investigator examines the host element of the address_space object (which points to an inode), s/he must be able to derive the corresponding address in the memory dump to determine where to look. This translation is accomplished by subtracting the constant PAGE_OFFSET—hex value 0xC0000000 in x86 hardware—from the virtual address [Ref. 6]. This means that the kernel virtual address 0xC1234567 is, in reality, physical address 0x01234567. The reason behind this conversion is that the virtual address space in the typical x86 machine is 4 GB and the topmost Gigabyte is assigned to the kernel [Ref. 6]. This upper Gigabyte begins at 0xC0000000. So physical memory begins at virtual address 0xC0000000, which is equivalent to physical memory address 0x00000000. This memory layout can be seen in Figure 2.

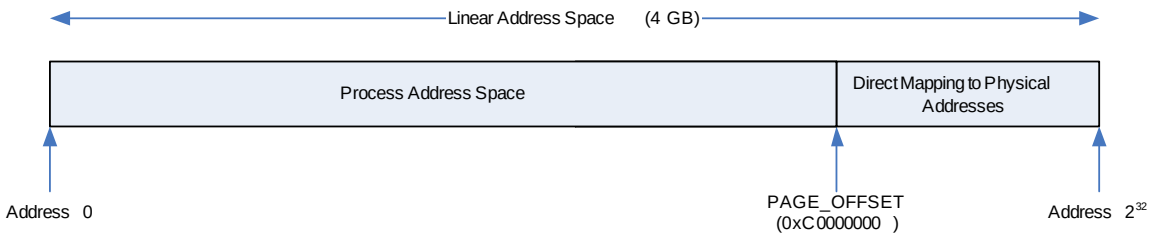

Figure 2. Mapping from Linear/Virtual Address Space to Physical Addresses

2. Page Frame to Page Descriptor Conversion

An investigator might want to find the page descriptor associated with a certain page frame. For example, if the investigator searches physical memory and finds a page frame that matches the signature for a known computer worm, s/he will want to find any related page frames. One way to do that is to find that page's page descriptor, follow the mapping pointer to the corresponding address_space struct, and read all the pages that have page descriptors in the radix tree (see Chapter IV, Section B, Subsection 3). The following equation will convert a page frame address to its related page descriptor address:

[(Page Frame Addr. >> PAGE_SHIFT) x Page Desc. Size] + mem_map start

PAGE_SHIFT is typically 12 bits—the shift is equivalent to dividing by the size of a page frame (0x1000 bytes). As discussed above, mem_map begins at 0x1000000 and the size of a page descriptor in the system used for this paper is 0x20 bytes. The page frame address is meant to be a physical address.

3. Page Descriptor to Page Frame Conversion

An investigator might also want to find the page frame associated with a certain page descriptor. In the example in subsection 2 above, when the investigator wants to look at the page frames associated with the page descriptors found in the radix tree, s/he could use the following equation:

[(Page Desc. Addr. – mem_map start) / Page Desc. Size] << PAGE_SHIFT

As the inverse of the previous equation, the PAGE_SHIFT in this case is equivalent to multiplying by the size of a page frame. The variables and constants for this equation are the same as the equation in subsection 2 above. The page descriptor address is meant to be a physical address.

D. PAGE TABLES

The standard way for the Linux operating system to translate between virtual and physical addresses is to use paging. Paging is a way to break down a virtual address into sections, each representing an offset into a table that is a part of a different level of indirection. Earlier versions of the Linux kernel made three levels of indirection available—even though most systems only required two. The third level was only used in conjunction with a special feature in the x86 architecture called Physical Address Extension that allows a system to address 64 GB of physical memory. However, in order to provide compatibility with new 64-bit architectures, a need arose to accommodate a greater amount of memory. Beginning in kernel 2.6.11 there are now four levels of indirection used in Linux systems: Page Global Directory (PGD), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry (PTE). PGD's are tables that point to PUD's, which are tables that point to PMD's, which are tables that point to PTE's, which point to the final level of page tables, which point directly at page frames. This sequence is illustrated in Figure 3.

For 32-bit systems without Physical Address Extension Linux sets the length of the upper and middle directory fields to zero. Collapsing both of those page tables to one entry and keeping them in the chain of pointers maintains compatibility with 64-bit systems.

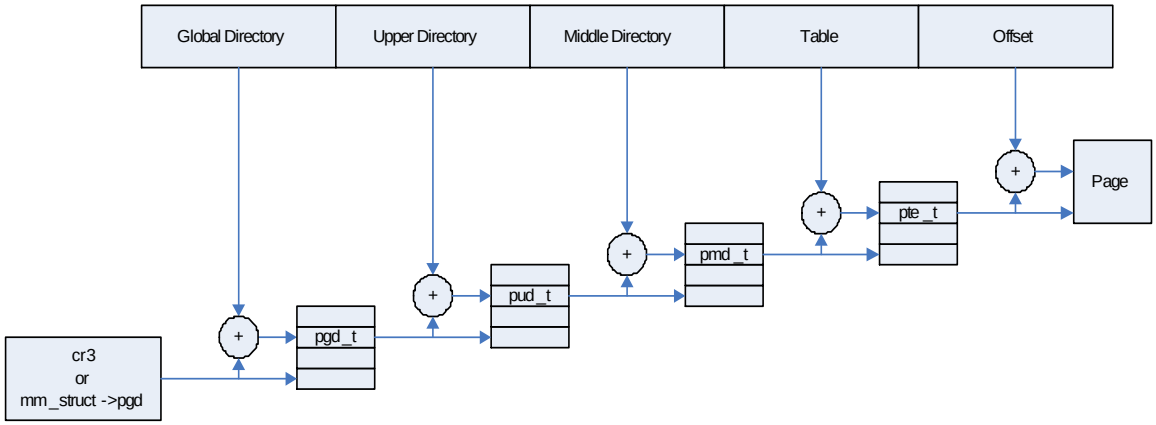

Figure 3. Page Table Function. (From Ref. 5)

E. SWAP SPACE

In order to expand the address space that is effectively usable by a process and to expand the amount of dynamic RAM, modern operating systems use the method known as swapping [Ref. 4]. Technically, swapping is the practice of transferring a whole process address space to disk, but modern operating systems only transfer portions of a process' address space to backing storage when the amount of free physical memory becomes scarce. In Linux systems this typically shows up in the form of a hard disk partition devoted to this task, but swap areas can be stored in files as well. A Linux system can have as many as 32 swap areas as defined by the MAX_SWAPFILES constant [Ref. 6]. The system used in this paper makes use of the hard disk partition method.

Only certain types of data are eligible to be placed in swap space. They are as follows (Ref. 4):

- Pages that belong to an anonymous memory region of a process. In other words, a region of a process that does not map a file on disk.
- Modified pages belonging to a private memory mapping of a process. Private memory mapping is when a process associates a memory region to a portion of a file on disk or on a block device—the private part means

that it only reads from this mapping; there is no writing. The process can still modify these pages even if they are read only. However, the modified pages cannot be written back to the disk or block device they belong to.

- Pages belonging to an inter-process communication shared memory region.

1. Swap Area Descriptor

Every swap area is described by a swap_info_struct—which is defined in /usr/src/linux-2.6.13-15/linux/swap.h. This structure contains a pointer to the swap area (either partition or file) and information including the size of the swap area, the number of usable pages, and a pointer to the swap map. There is a statically declared array called swap_info that holds all of the existing swap_info_struct objects in a system.

2. Swap Map

The swap_map array is an array of counters representing page slots in the swap area. There is an array element for every page slot in the corresponding swap area and the counter represents the number of processes using the page in that particular page slot. If the counter that represents a certain slot contains the number SWAP_MAP_BAD (32, 768), then it is an indication that the slot is defective.

3. How the Pieces Fit Together

The following figure illustrates how all the pieces in the management of swap space are related.

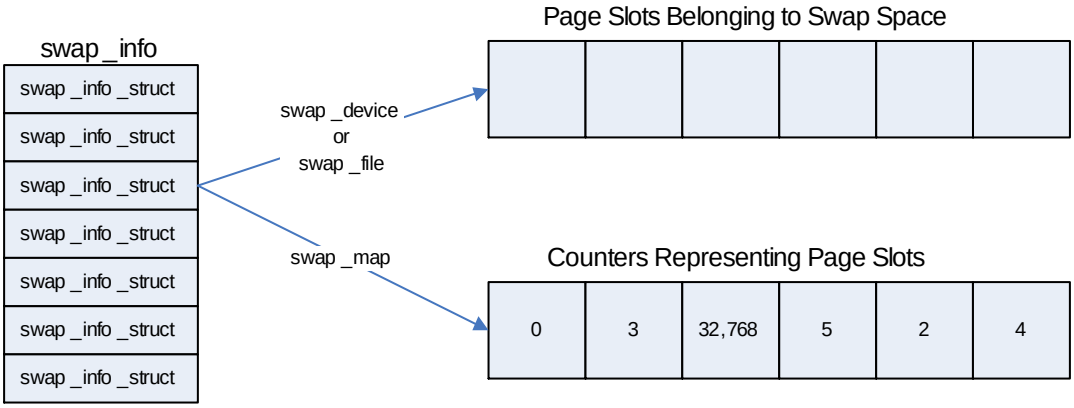

Figure 4. Swap Space Function. (From Ref. 4)

F. IMAGING RAM IN LINUX

The tool used to image physical memory and swap space throughout the course of this thesis was dd. This is a simple and powerful command line tool included in Linux distributions that can be used to copy the standard input to the standard output [Ref. 15]. To create an image of RAM, the directory /dev/mem was copied to a file. This directory represents RAM and, as such, when it was copied, so were the contents of memory. The actual command used looked like this:

"dd if=/dev/mem of=whole_mem.img bs=1"

The "if" operand defines an input file and the "of" operand defines an output file. The output file for this particular run of dd is "whole_mem.img". The "bs" operand sets the block size to n bytes—in this case n is equal to one.

Images of swap space were taken using a similar set up. One of the differences is that the swap space, as it was set up for this thesis, was represented by the /dev/hda3—the third hard disk partition. The other difference is the use of the "count" operand. It specifies the number of input blocks to copy. The actual command used looked like this:

"dd if=/dev/hda3 of=swap_space2.img bs=1 count=245760000"

III. CURRENT STATE OF RAM FORENSICS

Ram forensics is a blossoming field and, as such, has not yet reached maturity. In fact, it can be safely said that this field is in its infancy stage. Documents such as [Ref. 7] and [Ref. 8] describe the collection procedure for volatile memory in the most general of terms. They both agree that an investigator handling an incident should begin by collecting volatile data, which includes physical memory. Unfortunately, they don't specify how an investigator should approach the analysis of a system's memory. A list in [Ref. 8] mentions some useful programs such as ps that can be used to extract volatile data from a system. One incident response text [Ref. 1] only goes as far as mentioning that few people go further than running a string search of a memory image.

The most recent of the documents referenced in the above paragraph is [Ref. 6] which was written in 2004. In March of 2005, a man by the name of Mariusz Burdach released a paper [Ref. 3] providing a more in-depth look at the forensic analysis of physical memory of a Linux system. Burdach later presented [Ref. 9] at Black Hat Federal in 2006—which was largely a re-telling of his previous paper with some additional information regarding Windows physical memory forensics. He has developed tools based on the principles outlined in his works that are available at http://forensic.seccure.net. The only problem at this time is that the focus has been on the Linux 2.4 kernel and, since this kernel was superseded by the next stable release version (2.6) in 2004, the focus needs to shift to the newer kernel. The 2.6 kernel has changed aspects of its virtual memory management that must be explored to determine if the same techniques that worked on kernel 2.4 remain effective.

One aspect of the slides he presented that was an improvement over the paper was Burdach's explanation of the relationship between page table entries and swap space. He explains that when a page is swapped out, its page table slot will be filled with a pointer to swap space. He also mentions that the way to know that a page table entry is an index to a swapped out page is to check that the least-significant bit is cleared. This is true, but it is not always the case. Bit 0 (the least-significant bit) corresponds to the _PAGE_PRESENT flag so it would make sense to assume that if it is cleared, the page is not present in memory. The missing piece of information is that bit 7 corresponds to the

_PAGE_PROTNONE flag. When a memory region must be protected from user space processes this bit is set and bit 0 is cleared [Ref. 6]. This means that the page is present, just not accessible. A programmer must keep this case in mind when determining if a page table entry contains a swap space index or not.

On the Windows side, the latest demonstrations of physical memory forensics techniques can be found in the solutions to the 2005 Memory Analysis Challenge presented by the Digital Forensic Research Workshop website [Refs. 13, 14]. Some of the theory discussed by the two winning answers can be applied to a Linux investigation and some cannot. The following paragraphs address the application of windows forensic theory to Linux.

To begin with, both answers looked at all the processes listed in the doubly linked eprocess list to create a printout of all the processes running on the system. This is equivalent to walking the task_struct list shown in Figures 1 and 6 to enumerate all running processes on a Linux system. The files representing each process were recovered and compared to known versions of those programs to check their integrity. In this way, a few suspicious processes were found. Using the radix tree structure (Chapter IV, section B, subsection 3) to rebuild process files could allow an investigator to use this method in a Linux environment.

The second response also went one step further in that it searched windows objects such as the handle tables—which contain references to their owning process—to make sure that there were no hidden processes. This is because a technique known as Direct Kernel Object Manipulation could unlink a process from the eprocess list to avoid detection through a simple walk of that table. An analogous technique in Linux to verify if a task_struct has been unlinked from the doubly linked list of task_structs could begin by enumerating all mm_structs. An investigator could then see if, after checking the list of task_structs, there are any mm_structs that are not accounted for. The difficulty here would be that there is no pointer from a mm_struct to its parent task_struct so the search for the corresponding task_struct would require a heuristic approach.

Another technique discussed in the second answer to the challenge was the search of segment descriptor information through objects such as the Global Descriptor Table and the Local Descriptor Table. Suspect entries in these tables could reveal the presence

of a kernel function hook. This is a way to insert a function of choice between the kernel and user space processes to filter unwanted information flow to user processes. It is a good way to hide processes and/or files from programs that would otherwise detect their presence. These techniques work in a Windows environment because Windows uses segmentation. However, since Linux does not use segmentation, gathering GDT or LDT data would be of no use to an investigator examining a Linux system.

The final two methods used in the forensics challenge were checking MAC times and performing string searches for known malware strings. The MAC times provided information about when a suspicious executable file was opened and, consequently, the probable time that the attack against the target system began. The inode struct in a Linux system has this information available. Finally, the string search would be possible in a Linux environment because performing such a search is not OS specific.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ANALYSIS

A OVERVIEW

The next few sections go through specific examples of how to perform certain analysis methods on the physical memory of a Linux system. However, this section offers the reader a blueprint of those methods so that s/he can better understand them when they are explained in detail.

The first method allows an investigator to enumerate all of the processes present in physical memory. This function is accomplished by traversing the doubly linked list of task_structs shown in Figures 1 and 6. The steps required are as follows:

1. Find the address of init_task—the first element in the list—from the system's symbol table.
2. Go to that address and retrieve the desired information from the elements of the init_task struct—it is of type task_struct. For example, the name of the process represented by the task struct can be found in the char array called comm at offset 0x1A4 from its first byte.
3. Find the task element of init task because it contains the pointer to the next task_struct in the list and follow the pointer.
4. Gather required information again—keep in mind that offsets are not from the beginning, but from the task element now (see next section).
5. Follow these pointers and gather information until the pointers wrap around to init_task.

The next method is rebuilding a file. The next section provides a couple of ways to do this. The first requires searching the mem_map array and gathering pages into bins according to their parent address_space struct. This is not an efficient operation, but it can rebuild just about any file in physical memory. The second method uses a radix tree structure found in the address_space struct to rebuild a file that represents a process. In other words, if you run the program "prog", then this method finds the file from which "prog" was run. Here are the steps required for the second method (since the first was easily summarized above):

1. Traverse pointers illustrated in Figure 1 from a task_struct down to it's corresponding address_space struct.

2. Find the page_tree element of the address_space struct. Then find its third member, which is a pointer to a root node. Traverse the tree as shown in subsection 3 of the next section.

3. Convert all the page descriptors found in the radix tree into page frame numbers.

4. The page frame numbers are in order. Copy them in order into the same file.

Finally, swap space analysis is incorporated. This is useful when the method described above does not work because certain pages belonging to the file were swapped out. The necessary steps are:

1. Follow mm pointer from task_struct to mm_struct.

2. Follow pgd pointer in mm_struct to pgd.

3. Each pointer in the pgd goes to a page table. Follow these links to page tables and search the page tables until you find the group of pointers to the pages referenced by the radix tree—it is assumed that an analysis of the radix tree was already performed.

4. Once those pages are found, look at the adjacent pointers in that particular page table. If some of those addresses look like indices into swap space—bits 0 and 7 are cleared—then look in the appropriate swap space page slot for the needed page.

B. PHYSICAL MEMORY

1. Traversing task_struct Linked List

Figure 1 shows that the topmost virtual memory structure is the task_struct. Since this is the most general structure, it is where the analysis of physical memory begins. The doubly linked list of task_structs can provide the names of every process currently running in a system just like the Linux ps command. However, the advantage of this method over ps is the fact that it is immune to the manipulation of a rootkit because it does not rely on the target system's kernel. This list of task_structs is anchored by a special task_struct named init_task that can be found through the address listed in the kernel symbol table. The symbol table contains the addresses of many important structures that the kernel needs to access and it resides in /boot/System.map-2.6.13-15.8-default. The reader should note that systems running other versions of the kernel would have different values than 2.6.13-15.8. For example, System.map-2.6.11-default could be the symbol table name for a system running the 2.6.11 version of the kernel.

Using the command “cat /boot/System.map-2.6.13-15.8-default | grep init_task”, provided a list that included the address of init_task. This output can be seen in Figure 5.

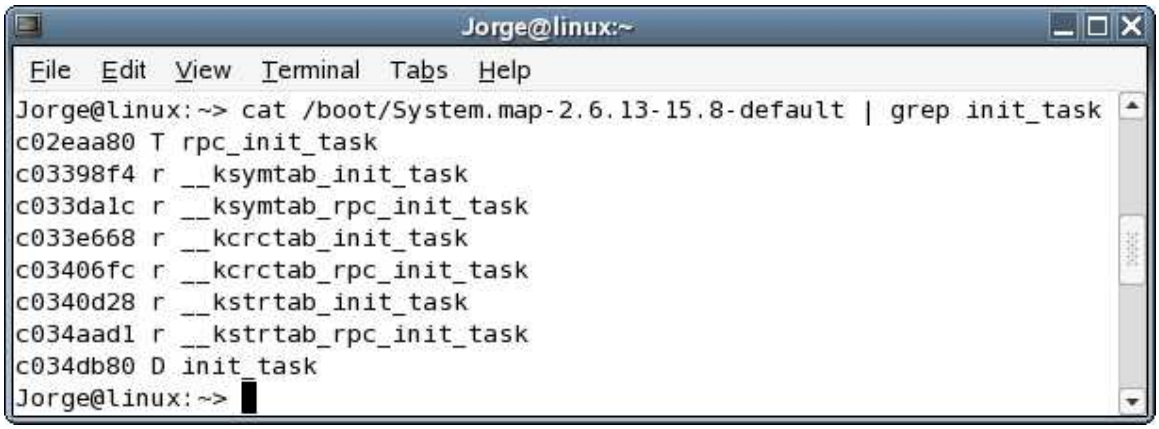

Figure 5. Finding Address of init_task

The address of init_task can be seen at the bottom of the list as 0xC034DB80. The command

“dd if=/dev/mem of=task_struct.img bs=1 count=200 skip=3464064”

outputs 200 bytes from the address at which init_task resides. The value used in the skip parameter is merely the transformation of 0xc034DB80 to a physical address 0x34DB80 and finally to a decimal number. Any virtual address used in the context of a dd command must go through the same transformation. Also, the value of 200 in the count parameter was chosen arbitrarily to cover enough of the task_struct to see its tasks element, ie, it is greater than or equal to sizeof(task_struct).

The tasks element of a task_struct is of type list_head, which means that it is made up of a next and prev pointer. The next pointer for the init_task structure was found 0x60 bytes from the beginning of the struct and contained the value 0xDFE01AA0. Dd was again used to carve memory from this address, after which it became apparent that the first word of data did not look like the beginning of a task_struct. More specifically, it looked like an address (a pointer) instead of a typical number of type long because it was a value above 0xC0000000. The reason why this value looked suspicious was that no typical number of type long was ever found to be above 0xC0000000 during the course of this research. This incongruence was caused by the fact that the next and prev pointers in task elements of task_structs point to the task elements of the next and

previous task_structs in the doubly linked list. In other words, the linked list goes from task element to task element, not from task element to the beginning of the next (or previous) task_struct. The prev element of this task_struct pointed, as expected, to address 0xC034DBE0, which is the address of the task element of init_task.

Repeating this exercise, I found that the prev element of the next task_struct in the list contained the address 0xDFE01AA0—the address of the second task_struct in the list. The layout of the links can be seen in Figure 6. Note that it is a doubly linked list so, at a certain point, it will wrap around and the next element of a task_struct will contain the address of the next element of init_task. Likewise, the prev element of init_task will point to the next element of that "last" task_struct.

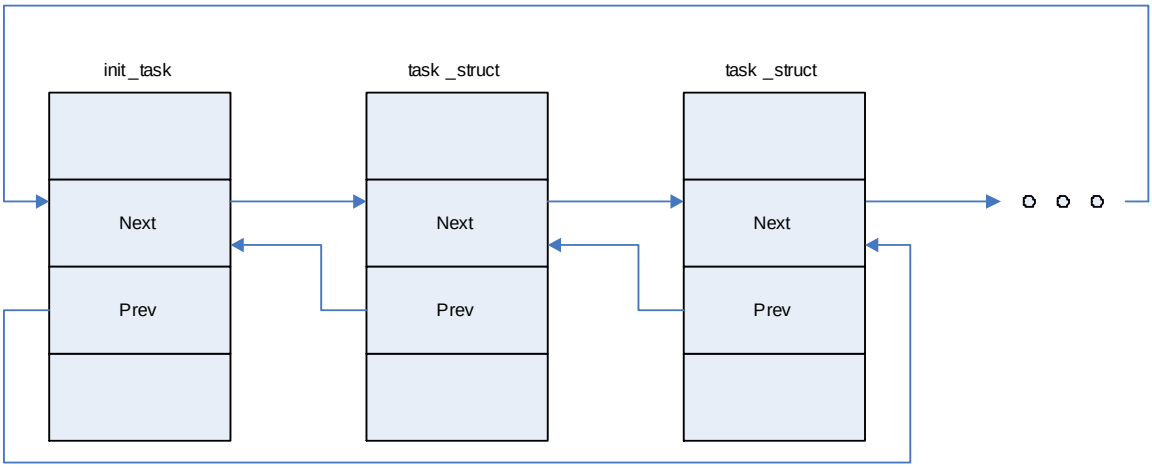

Figure 6. Doubly Linked Task List

Once the traversal of the task_struct list was mastered, a Perl script was developed to enumerate all tasks in a physical memory dump and output their executable name, pid, pointer to associated mm_struct, and the pointer to the next task_struct. Appendix A contains the code for this script under the name find_task.pl.

2. Rebuilding File

A simple jpeg image of Waldo—the character from the "Where's Waldo?" series of children's books—served as a way to test the reconstruction of a file from information found in the mem_map array. The test began by noting the first 8 bytes of each page to use as a signature that could be searched for in an image of physical memory. Only four such signatures had to be written down because the length of the file was four pages.

Then, while having the waldo.jpeg file open in a file viewer, a RAM image was obtained utilizing dd. The resulting memory image was opened in a hex editor so that the signatures could be used to find each page that made up the image. The idea behind the signatures was to provide a way to recognize the pages in memory and work backwards to find their corresponding address_space struct. A couple of false positives were easily filtered because they were not found at 4K boundaries where the beginnings of pages should be found. Pages should be found at addresses that are multiples of 0x1000 (one page size). For instance, a signature found at physical address 0x4935184 could not be one of the image's pages. The real one was actually found at physical address 0x509C000. The rest of the search yielded the data presented in Table 1.

Page Number	Physical Address
1	0x509C000
2	0x4405000
3	0xBE79000
4	0x5D2A000

Table 9. Pages Found in Image

These physical addresses were turned into page descriptor addresses using the calculation covered in section C of the Background chapter. Analyzing each descriptor's index element—which is 0x14 bytes from the beginning of the page descriptor and refers to the page offset within the mapping—reaffirmed the fact that the pages were in correct order. The page descriptor addresses corresponding to each page and their indices are outlined in Table 2.

Page Number	Page Descriptor Address	Index
1	0x10A1380	0
2	0x10880A0	1
3	0x117CF20	2
4	0x10BA540	3

Table 10. Page Descriptors and Indices

A piece of information proving that these pages were part of a cohesive section of memory representing an image was found in the mapping element of their page descriptors. The mapping element was found at 0x10 bytes from the beginning of each page descriptor and contained the address to the same parent address_space struct. All of these pages belonged to the address_space struct found at virtual address 0xC6D3E3B4 (physical address 0x6D3E3B4). A Perl script was developed to take in the virtual address of an address_space struct and print out all the pages, in order, that belong to it. Parsing all the pages in the mem_map array and seeing which ones have a mapping field that matches the input address is the method used to perform this task. Appendix A contains the code for this script under the name find_waldo.pl.

However, a typical forensic investigation would not begin with a search for a particular address_space object in memory because the investigator would not initially know what address_space struct to look for. The ideal thing for an investigator to do would be to walk the whole mem_map array and group pages in bins according to the address_space that owns them. Once this was done, the investigator would possess every file existing in memory—provided that none of that file's pages were swapped out.

To find the name of the waldo.jpeg file in memory it was necessary to follow the host pointer from the address_space struct to the related inode, and then the i_dentry pointer from the inode struct to the dentry struct. The host pointer is the very first word in the inode struct. In other words, it can be found at offset 0x0 from the beginning of the address_space struct. The i_dentry pointer in the inode struct was at offset 0x18 and was found to point to the d_alias element of the dentry struct. At a location 0x30 bytes ahead of d_alias, the d_iname element held the name of the image file: waldo.jpeg. The structure of these links is illustrated in Figure 7.

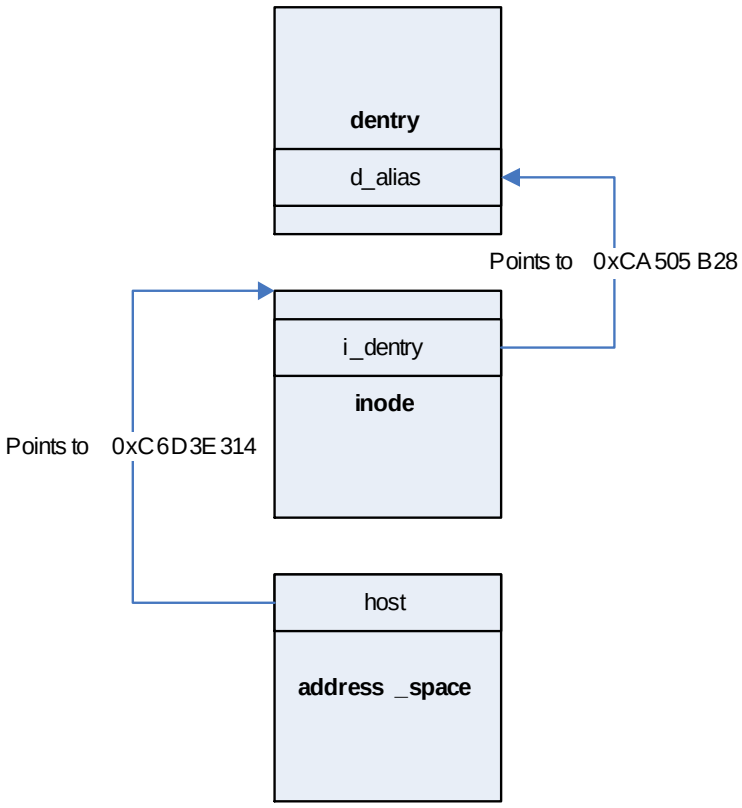

Figure 7. Pointer Structure from address_space to dentry

3. Radix Tree Structure

While noting some of the differences between the address_space struct in kernel 2.4 and kernel 2.6, a new addition was found: the radix tree. This structure is rooted at the page_tree element of the address_space struct. A radix tree is made up of a root, and a series of nodes. The root element, defined in /usr/src/linux-2.6.13-15/linux/radix-tree.h, consists of the following:

```
struct radix_tree_root {
unsigned int                height;
int                         gfp_mask;
struct radix_tree_node      *rnode;
};
```

The node element, defined in /usr/src/linux-2.6.13-15/linux/radix-tree.c, consists of the following:

```
Struct radix_tree_node {
Unsigned int    count;
Void            *slots[RADIX_TREE_MAP_SIZE];
Unsigned long tags[RADIX_TREE_TAGS][RADIX_TREE_TAG_LONGS];
};
```

The traversal process for this structure, using the same memory image as in section 2, began with the address_space struct related to the Waldo.jpeg file at physical address 0x6D3E3B4. The page_tree element was found 0x4 bytes into the address_space struct. It had a height value of one and an rnode pointer with a value of 0xC531D034. Following this pointer to physical address 0x531D034 yielded a radix_tree_node with a count value of 4—which was followed by the virtual addresses of all four page descriptors associated with the Waldo file in correct ascending order. It should be noted that the total number of pages owned by this particular address space was known to be four and that this number was derived from the nrpages element of the address_space struct at offset 0x30. This traversal seemed to suggest that the height element of the radix_tree_root struct was responsible for the number of indirection steps needed. In addition, it suggested that the count element of the radix_tree_node element outlines how many pointers must be followed. This relationship is illustrated in Figure 8.

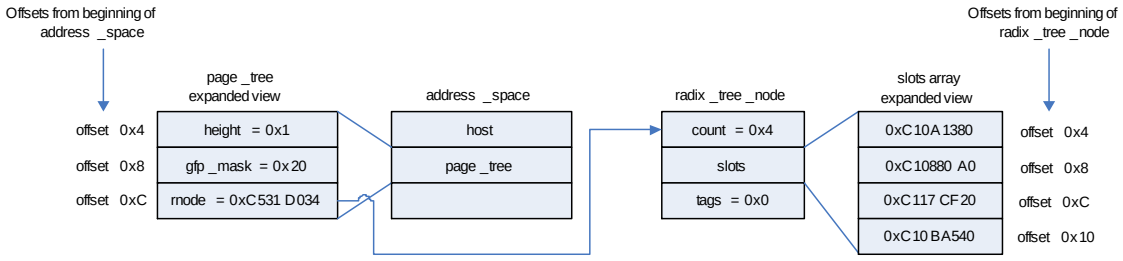

Figure 8. Radix Tree Structure

An analysis of radix trees was later conducted with another address_space struct in the same memory image to confirm these conclusions. The new address_space struct was located at physical address 0x1F064E44 and had a page_tree height value of two. The rnode link of the page_tree element pointed to the virtual address 0xDF0307C0. This node had links to 5 other nodes that, in turn, had links to numerous page descriptors. Since the height of the tree was two, there were two levels of indirection: one from the

primary nodes to secondary nodes and one from secondary nodes to page descriptors. The total number of pages was listed in the address_space struct as 287. This coincided with the number of page descriptor links in the radix tree because the first four links from the primary node had pointers to 0x40 page descriptors while the fifth link from the primary node had pointers to 0x1F page descriptors. Summing these yields 4 x 0x40 + 0x1F = 0x11F = 287_{10}. This scheme is shown in Figure 9.

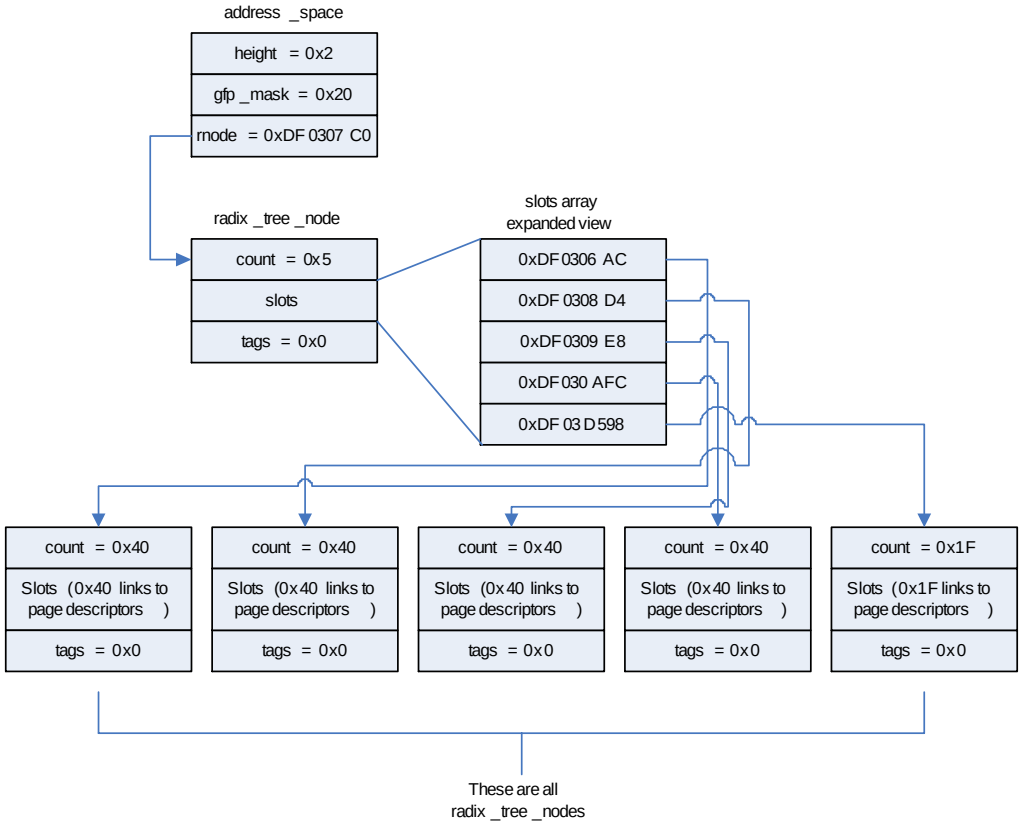

Figure 9. Structure of Second Radix Tree Explored

With this information, the general structure of radix trees was discovered. The height value in a root node controls the levels of indirection and the count value in a node is equal to the number of addresses—either pointers to other nodes or to page descriptors—contained in the slots array.

C. SWAP

Section E of the Background chapter gave an overview of the reasons behind using swap space and the structures that comprise it. In the following, a more in depth

discussion will provide a way to use this repository of data to complement the analysis of physical memory outlined in section A. The idea is that if an investigator can analyze the page tables belonging to a process, s/he will be able to retrieve any pages that have been swapped out to swap space. This is made possible by the fact that the kernel always leaves an index into swap space—actually the swap_map array—as a placeholder for a page frame that has been swapped. The format of this index entry can be seen in Figure 11.

One of the experiments performed for this thesis involved using the C program heap.c (Appendix A). This code was written to place 400MB worth of pages with predetermined signatures in the program's heap space. The idea was that such a large space requirement would force the kernel to put some of those heap pages in swap space. The 64 bit signatures were structured to provide a program's pid and the page number. A page would be filled with a repeating pattern of the signature so that the hex dumps of the pages would be meaningful and easy to recognize. The exact structure of the signatures is shown in Figure 10. This program provided a foundation for the analysis that follows.

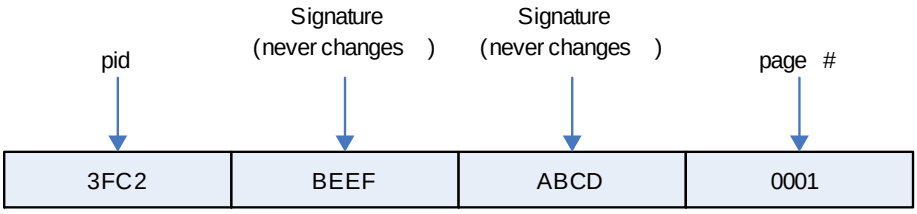

Figure 10. Signature Used in heap.c

An image of both memory and swap space was taken using dd while heap.c ran. The count number for the swap image command—245760000—was calculated after noticing that around 19% of swap space (1 GB) was being used by heap.c. The count number above is around 23% of swap space to be on the safe side. It should be noted that page slots in swap space can be assumed to fill from the front of the available space because the kernel tries to store pages in contiguous slots to reduce disk seek time during swap space access [Ref. 4]. Using the Perl script find_task.pl (Appendix A) to enumerate all the processes from the image file yielded the address of heap.c executable's task_struct. The following table outlines the complete tracing of this process:

Structure	Address (Physical)
task_struct	0xB45A590
mm_struct	0x11335300
vm_area_struct	0xB55BBCC
File	0xB56D680
Dentry	0x132DAA6C
Inode	0x132D64EC
address_space	0x132D658C

Table 11. Tracing of Heap Process

Using the radix tree structure from the address_space struct to enumerate all pages belonging to this address space yielded only one page at physical address 0xBBA7000. This meant that any page table present in memory that was related to the heap process would have an entry with that address. Since the last three hex digits of a page table entry are flags, the value to search RAM for was either 0x0BBA7*** or 0xCBBA7*** (the virtual equivalent)—where the asterisks represent wildcards. This search was performed with the help of the find_pattern.pl script (Appendix A) using the following commands:

perl find_pattern.pl 0bba7… (for physical address)

perl find_pattern.pl cbba7… (for virtual address)

The first search yielded 8 hits, while the second search yielded none. This result suggested that the values in page tables represent physical rather than virtual addresses.

An inspection of the mm_struct revealed that the pgd pointer contained the address 0xB338000. The value 0xBE55067 was found at offset 0x80 into the pgd. It did not look like it was a virtual address, that is, it was not a number above PAGE_OFFSET (0xC0000000). This meant that it was a physical address—which correlated with the results of the image search above. Since the least significant bit of this number is set, the page is resident in memory and, therefore, the pointer could be followed to a page frame at 0xBE55000 because the last three hex digits are flag values. The values 0xBBA7025 and 0x1113C00 where found at offset 0x120 in this page frame. The first of these values is equivalent to page frame 0xBBA7000, which was the same page frame found through

the radix tree traversal above. The correlation was strong enough to determine that the pgd actually pointed to a page table that contained page frames belonging to the heap process. The second value looked like a swap space index entry of type swap_entry_t, which is illustrated in Figure 11. The entry had bit's 0 and 7 cleared, so it was not in physical memory. The offset into the swap_map array was 0x1113C.

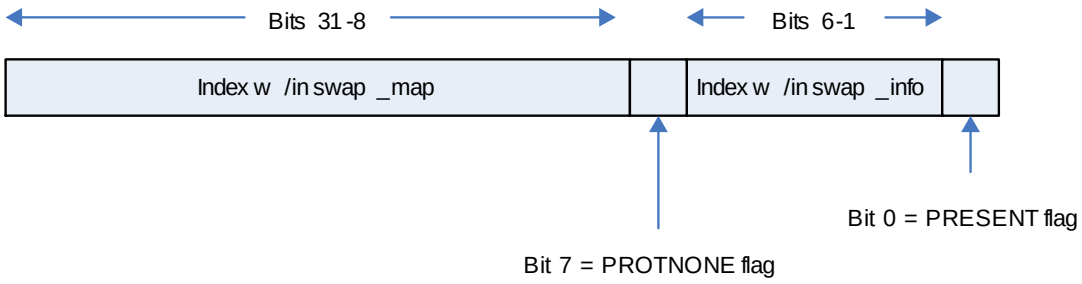

Figure 11. swap_entry_t layout (After Ref. 6)

Unfortunately, the image of swap space was not large enough to be able to include the index found above and, consequently, the swap page slot could not be confirmed to be a part of the heap process. Nonetheless, it is very likely that this was the case.

Another round of RAM and swap space images was then created to recreate the results found above. The size of the swap space image specified in the count field of the dd command was increased to 429,496,673—roughly 40% of the entire swap space—to avoid another situation where the index into swap was beyond the bounds of the image. In the new image, following the heap executable links from task_struct to mm_struct to find the pgd pointer and, finally, to the page table yielded two addresses. The first address corresponded to the page descriptor address found through the radix tree in the address_space struct. The second page was, unfortunately, not a swap_entry_t value. It was actually a match for the first half of the first page in the heap executable, which suggests that it was a page captured in transition during the imaging process. This is the type of problem that arises from capturing an image from the same system that the imaging tool is running on. Running the process at the highest priority level could have mitigated this type of problem. Unfortunately, this change could not have completely stopped the process scheduler or, for that matter, the constant changes that the kernel makes to physical memory over time.

A further look into the pgd of the heap process revealed that there were 3 different areas of pointers. The first was the single pointer that was traced in the previous paragraph. The second and third areas held approximately 100 pointers each. Following some of these links to the corresponding page tables proved that they were pointers to page frames filled with either ELF binary code—possibly libraries—or values allocated to the heap by the heap executable. This suggests that the pgd is not a solid block of pointers and that any gaps are inconsequential. The heap tracing in this new image can be seen in Table 12.

Structure	**Address (Physical)**
task_struct	0x113D3590
mm_struct	0x176C9300
vm_area_struct	0x1E608A14
File	0x5A36980
Dentry	0x525B494
Inode	0x14BC69F4
address_space	0x14BC6A94

Table 12. Second Heap Process Trace

The integration of swap space data was made evident by tracing from the pgd to a page table that contained a mix of swap space indices and page frame addresses. The page frame was at address 0xF00000 and two of its pointers had the values 0xF2C200 and 0xF1C047. The first value indicated that the page was not in memory because bits 0 and 7 were not set. This meant that 0xF2C200 was actually referring to the swap map index 0xF2C2 and, consequently, the page slot at swap address 0xF2C2000. The second value had bit 0 set, so it was resident in physical memory. These two pointers sat next to one another in the page table so they had to point to consecutive pages. When the page values—from the heap.c signatures—were checked for both programs they were consecutive. The page slot from swap space had a page value of 0xA2CF, while the page frame in physical memory had a page value of 0xA2D0. A page missing in RAM was therefore successfully retrieved from swap space. The tracing is shown in Figure 12.

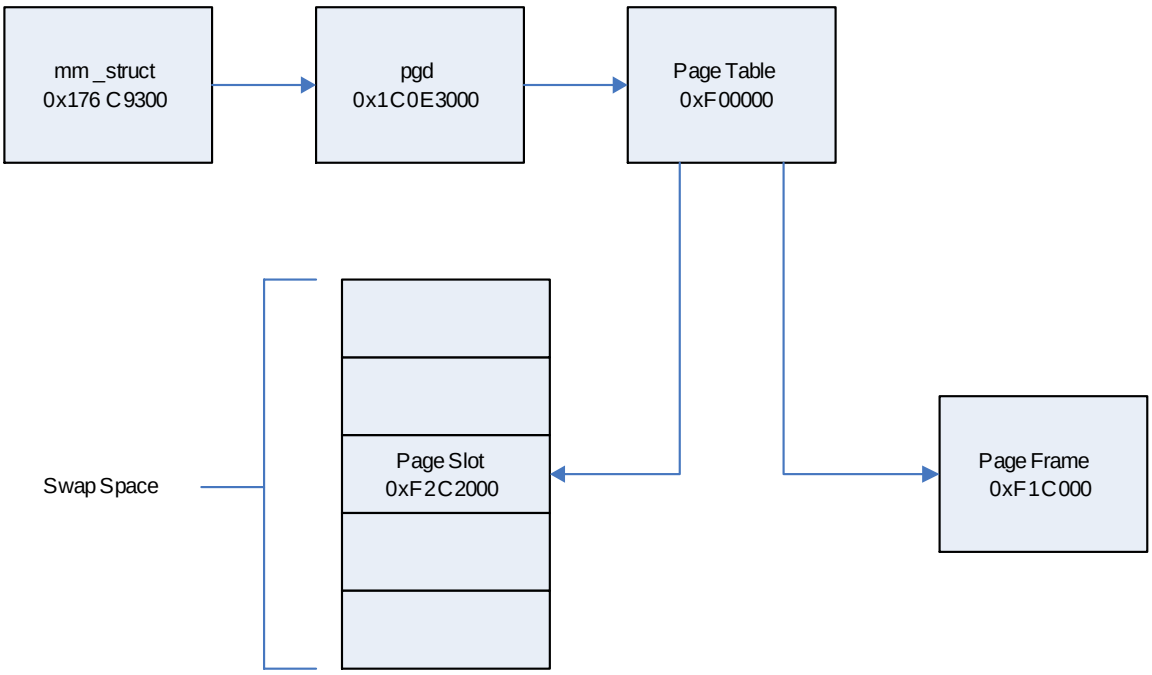

Figure 12. Tracing of Pages in Swap

V. CONCLUSION

A. SUMMARY

The memory management changes in the new 2.6 Linux Kernel have not adversely affected the forensics techniques developed for the 2.4 kernel. In fact, the introduction of the radix tree in the address_space struct made the enumeration of all the pages belonging to an address space much easier and more efficient. The discovery of data belonging to a process inside swap space is possible—as shown in the previous chapter—and should add value to the full exploration of processes that reside in physical memory. The techniques outlined in this paper, or programs based on them should now be integrated into any in-depth forensic analysis of a computer system.

There is a new breed of malware based on the concept that physical memory is largely out of the hands of an examiner. The effects of this insidious software can be mitigated by the application of the findings in this paper. However, unless these analysis techniques are properly automated, many investigators will shy away from performing detailed analysis of physical memory and potentially miss many pieces of evidence vital to such an investigation.

B. PROBLEMS

Throughout the course of the analysis, a few problems were encountered. The first presented itself in the early stages of project development when the assumption was made that kernel space addresses had a range from 0xC0000000 to 0xCFFFFFFF—when it is actually from 0xC0000000 to 0xFFFFFFFF. This made many of the pointers used in the linking of memory management structures seem invalid and caused confusion as to how these structures were linked.

Another issue that consumed a great deal of time was related to the page tables belonging to a process. Much of the integration of swap space data depended on finding page tables so that the swap indices could be used to find the missing page in swap space. However, the pgd contained addresses that did not look like virtual addresses and that were not in one solid block. Previous addresses found in other structures had always been virtual kernel addresses (> 0xC0000000) and consecutive if they were part of a list of pointers. These two factors contributed to the belief that the pgd pointer was either not

pointing to the correct location, or that the pgd was full of garbage. It was not until the addresses in the pgd were taken at face value as physical addresses that the page table structure began to unfold.

A third problem arose when the interconnection of memory management structures was being analyzed. When there was a pointer from one structure to another, it was erroneously assumed that the pointer would always point to the beginning of that structure. This created some situations in which the pointer from one structure did not seem to point to another. An example of this fact is the doubly linked task_struct list. The links from one task_struct to another connect at the tasks element of one struct to the tasks element of another. This element is at offset 0x60 from the beginning of a task_struct. Looking at a task_struct at offset 0x60 and thinking it should look like the beginning of it would obviously lead someone to think it was not a task_struct.

A final problem was inherent in the way that the experimentation for this thesis was carried out. Due to budget and time constraints, the methodology used during the analysis of memory was completely software based. This meant that, as images of RAM and swap space were being collected, memory was constantly changing. The programs doing the imaging, heap.c, and typical kernel processes were constantly changing the landscape of memory over time. The best option would have been to pursue a hardware based method for imaging physical memory that would minimize any software/kernel interference. One such method would be to include a card in the systems PCI slot that could image memory at the flip of a switch [Ref. 12]. In lieu of imaging hardware, dd could have been satically linked and executed from a CD. However, since there was no concern over the images' validity in a court of law, it was deemed acceptable to run these commands from the target system.

The only artifact of these issues that was ever encountered arose during the final analysis of the heap process in the previous chapter. The first step in the tracing was to run the find_task.pl script to enumerate all the processes in ram and find the value of their corresponding task_struct. When it was run, the heap process did not appear. Further investigation revealed that if the links from the first task_struct—known as swapper—

were followed backwards, the heap process would be found. The doubly linked list read differently if the links were followed in one direction than in another. The list was caught at a moment of transition.

C. FUTURE WORK

The most obvious extension to the work in this paper would be to write a tool that could automate the techniques discussed. This work would include the fine-tuning of the concepts presented here so that they can be translated into the specific language of a program. Such a tool would be a powerful asset to an investigator because tracing hex dumps is not something that many investigators will want to do. That is, the exploration of physical memory would often be skipped in the absence of a good tool to do automate it elegantly and efficiently.

Another future focus of research would be to perform similar analyses on systems with different characteristics. This could take the form of research on other operating systems such as Windows or Solaris. It could also take the form of research on systems running on non-x86 platforms such as PowerPC or 64-bit systems. Systems with more than 896MB of physical memory should be explored as well. Unraveling the process of translating virtual addresses in ZONE_HIGHMEM to physical addresses and back would be very useful as the number of systems with memory in excess of this limit is increasing daily.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX. SOURCE CODE

This appendix includes all of the Perl scripts that were developed throughout the project to aid in the analysis of physical memory and swap space. A small c program is also included, which was used to force swap space to contain a series of pages that I tagged to ease their retrieval in hex dump analysis.

A. FIND_WALDO.PL

```
#!/usr/local/bin/perl
#This is a program that will take in the virtual address of
#an address_struct object and output all the pages (in order)
#that are owned by that particular address_space struct.
use strict;
use utils;

#-------------------------------------------------------------
#Declare Variables
#-------------------------------------------------------------
my $file;               #Image file to look into
my $address;            #Address input from commandline
my $user_in;            #User input
my $length;             #Length of address_space address

#-------------------------------------------------------------
#Get Inputs
#-------------------------------------------------------------

#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-f”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then kill program
if($user_in eq “-p” or $user_in eq “”)
      {
die “Name of file to be searched is invalid. Error found”;
      }

#Save string after -f in dump_file variable
$file = $user_in;

#Advance input
$user_in = shift;

#If a -p is found, then continue
if($user_in eq “-a”)
```

```
      {
#Advance input
$address = shift;

#If the address is empty then kill program
if($address eq “”)
            {
die “There is no address specified.  Error found”;
            }

      }
#Otherwise kill program
else
      {
die “No pattern flag -a is specified. Error found”;
      }
}
#Otherwise kill program
else
{
die “User did not use -f flag. Error found”;
}

#-------------------------------------------------------------
#Body
#-------------------------------------------------------------
#Make sure that the input is 8 bytes.  I check
#for 8 bytes because a pattern like 3a5d3876 looks
#like 8 bytes on the surface, but as it is stored
#in memory, it is really 4 bytes because it is in
#hex and each digit only needs 4 bits to be
#represented
$length = length($address);

#If it’s not correct length exit program
($length == 8) or die “The address input should be 8 bytes, not
$length. Found”;
#If it is not a valid hex number, then exit program.
#The period character is allowed as wildcard.
($address =~ m/[(a-f|A-F|0-9|\.)]{8}/) or
die “The input is not a valid hex number. Found”;
#Change any hex letters to lowercase to prepare pattern
#for later matches with data from memory that is read
#in as lowercase
$address =~ tr/[A-Z]/[a-z]/;
#Use find all pages matching this address space
Utils::find_address_space($file, $address);
```

B. FIND_PATTERN.PL

```
#!/usr/local/bin/perl
#This program attempts to find a 4 byte pattern
#from a file of choice.  Any hits will be stored
#in an array.
```

```
use strict;
use utils;

#-----------------------------------------------------------
#Declare variables
#-----------------------------------------------------------
my $length;             #This is the length of the input
                        #and later the file (in bytes)
my $file;               #File to be searched
my $read;               #Value read from file
my $address = 0;        #Address where to read next value
                        #from
my $pattern;            #4 byte pattern from commandline
my $user_in;            #User input
my @hits;               #Array of matches to pattern

#-----------------------------------------------------------
#Get Inputs
#-----------------------------------------------------------
#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-f”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then kill program
if($user_in eq “-p” or $user_in eq “”)
      {
die “Name of file to be searched is invalid.  Error found”;
      }

#Save string after -f in dump_file variable
$file = $user_in;

#Advance input
$user_in = shift;

#If a -p is found, then continue
if($user_in eq “-p”)
      {
#Advance input
$pattern = shift;

      }
#Otherwise kill program
else
      {
die “No pattern flag (-p) is specified. Error found”;
      }
}
#Otherwise kill program
else
```

```
{
die “User did not use -f flag. Error found”;
}

#-----------------------------------------------------------
#Body
#-----------------------------------------------------------
#Make sure that the input is 8 bytes.  I check
#for 8 bytes because a pattern like 3a5d3876 looks
#like 8 bytes on the surface, but as it is stored
#in memory, it is really 4 bytes because it is in
#hex and each digit only needs 4 bits to be
#represented
$length = length($pattern);

#If it’s not correct length exit program
($length == 8) or die “The pattern input should be 8 bytes, not
$length. Found”;
#If it is not a valid hex number, then exit program.
#The period character is allowed as wildcard.
($pattern =~ m/[(a-f|A-F|0-9|\.)]{8}/) or
die “The input is not a valid hex number. Found”;
#Change any hex letters to lowercase to prepare pattern
#for later matches with data from memory that is read
#in as lowercase
$pattern =~ tr/[A-Z]/[a-z]/;
#Open file in question
open(FILE, $file) or die “Memory image file could not be opened.  Error
found “;
print “Opened the file: $file \n”;
#Find out the size of the file
$length = -s $file;

#Divide length by four to turn it from bytes to words
#(4 bytes = word). This will control the # of times
#memory will have to be read to traverse the whole
#file
$length = $length/4;

for(my $i=0; $i<$length; $i++)
{
#print “Length in hex: $length\n”;
$read = Utils::read_val(*FILE, $address);
if($read =~ m/$pattern/)
      {
print “MATCH PATTERN at Address: $address\n”;
#Add to hit array
push(@hits, $address);
      }
#print “READ: $read\n”;
#Increment address by four bytes (a word)
#It is interpreted as hex when the addition
#happens and then turned into hex once again
#in preparation for the read_val function.
$address = hex($address) + 4;
$address = sprintf “%08x”, $address;
}
```

```
foreach $length (@hits)
{
print “Address: $length\n”;
}

#Calculate length of the array to output
$length = $#hits + 1;
print “Number of hits found: $length\n”;

#Close the file
close(FILE);
```

C. FIND_SIGNATURES.PL

```
#!/usr/bin/perl
#This is a program that will retrieve signatures from each page
#of a file in a Suse 10 Linux machine.  The signatures will be
#12 bytes long and be choosen from the middle of each page to
#reduce conflicts that might arise from pages having similar
#beginnings or ends.
use strict;
use POSIX qw(ceil floor);     #Adds floor and ceiling functions

#-------------------------------------------------------------------
#Declare variables
#-------------------------------------------------------------------
my @sigs;               #Signatures lifted from file to search
my $sig;                #One of the signatures in @sigs
my $out_file;                 #Name of the file that will hold
signatures
my $user_in;                  #User input
my $in_file;                  #Name of the file that signatures will be
                        #extracted from

#-------------------------------------------------------------------
#Declare variables
#-------------------------------------------------------------------
#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-i”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then continue
if($user_in eq “-o” or $user_in eq “”)
      {
die “Input file name is invalid.  Error found”;
      }

#Save string after -i in dump_file variable
```

```
$in_file = $user_in;

#Advance input
$user_in = shift;

#If a -o is found, then continue
if($user_in eq “-o”)
      {
#Advance input
$user_in = shift;

#If the output file name is there, then
#put input value in dir variable
if($user_in ne “”)
            {
$out_file = $user_in;
            }
#Otherwise kill program
else
            {
die “Output file name is missing. Error found”;
            }
      }
else
      {
die “The -o flag is missing. Error found”;
      }
}
#Otherwise kill program
else
{
die “User did not use -i flag. Error found”;
}

#-------------------------------------------------------------------
#Body
#-------------------------------------------------------------------
#Extract signatures from each page of a file
@sigs = get_signatures($in_file);

$sig = @sigs[$#sigs];
#The offset within the last page that the
#signature was taken from
print “Offset where signature in\nlast page of file comes from:
$sig\n”;

#Open the output file for writing
open(OUT, “>$out_file”);

foreach $sig (@sigs){
#print ”Page Sig: $sig\n”;
#Write signature to file
syswrite(OUT, $sig, 27);

#Write newline char to file
syswrite(OUT, “\n”, 1);
}
```

```
#Close the output file
close(OUT);

#This function retrieves a specified number of hex
#bytes from a file and returns them as one variable
sub read_bytes
{

      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
      my $bytes = shift;            #Number of bytes to read from file
      my $handle = shift;           #File handle of file to be parsed
                                    #(file should already be open)

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------

      my $extracted_bytes = “;      #extracte bytes from file
      my $read_in;                  #byte read in from file
      my $v = 0;                    #Verbose flag

      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------

#Output number of bytes choosen by caller
if($v==1){print “The number of bytes to read is: $bytes\n”;}
#Read in $bytes number of bytes
for(my $j=0; $j<$bytes; $j++)
      {
#Read in a byte
sysread($handle, $read_in, 1);

#Change signature into hex
$read_in = sprintf “%02x”, ord($read_in);
#Append byte to extracted_bytes
$extracted_bytes = $extracted_bytes.$read_in;

      }

return $extracted_bytes;
}

#This function extracts signatures from each page of a file.
#They are taken from the middle of each page (0x500 into the
#page).  The last page is different, however, because it is
#seldom full.  For that page the signature comes from the
#middle of the space that the file actually uses.
sub get_signatures
{
      #-----------------------------------------------------------------
---
#Declare variables
```

```
      #-------------------------------------------------------------
---
      my $input_file;                         #The file that will be
scanned
      my @sig_array;                          #Array of signatures
      my $file_size;                          #Size of the file
      my $total_pages;                  #Number of pages required by
                                        #file
      my $full_pages;                         #Number of filled pages
used
                                        #by file
      my $left_over;                          #Number of bytes
actually used
                                        #in the last page of the file
      my $index = 0x500;                      #Index into the file
      my $signature;                          #Signature
      my $i;                                  #Loop index
      my $v = 1;                        #Verbose flag

      #-------------------------------------------------------------
#Get Inputs
      #-------------------------------------------------------------
$input_file = shift;

      #-------------------------------------------------------------
#Body
      #-------------------------------------------------------------
#Open the file to be scanned
open(SCANFILE, $input_file)
or die “Signature file $input_file could not be opened.
           Error found”;
#Get the size of the file to calculate how many pages
#are required
$file_size = -s $input_file;
if($v==1){print “The file size is:        $file_size\n”;}
#Find the number of bytes used in the last page of the file.  #First I
divide file size by 4096 (page size) and then strip #decimal part
(floor function).  $full_pages = floor($file_size / 4096);
if($v==1){print “Full pages used:         $full_pages\n”;}
#Now we subtract the file size by the number of bytes that
#are required to fill the number of pages found above
$left_over = $file_size - ($full_pages * 4096);
if($v==1){print “The left over is:        $left_over\n”;}
#Divide this number (which is in bytes) by 4096 and take
#the ceiling function of that result to get the number
#of pages the file takes up
$total_pages = ceil($file_size / 4096);
if($v==1){print “Pages required by file:        $total_pages\n”;}
#Extract 12 byte signatures from all full pages
for($i=0; $i<$full_pages; $i++)
      {
#print “INDEX: $index\n”;
#Go to the correct index in the file
seek(SCANFILE, $index, 0);

#Read 12 bytes from the file
$signature = read_bytes(12, *SCANFILE);
```

```
#Save page signature in array
@sig_array[$i] = $signature;

#Empty signature for the before the next
#signature is read
$signature = “;
#Update the index
$index = $index + 0x1000;
      }

#Figure out the last page index for extracting
#the signature.  First step is to figure out
#the base index of the last page
$index = ($full_pages * 4096);

#Print out the base index of the last page
if($v==1){print “Base index of last page:       $index\n”;}
#Now we add half the number of bytes used in the
#last page (the floor function is used in case this
#division has a decimal component)
$index = $index + floor($left_over / 2);

#Print out the index used for the last page in the file
if($v==1){print “Last page index:         $index\n”;}
#Go to the index calculated for the last page
seek(SCANFILE, $index, 0);
#Read the signature
$signature = read_bytes(12, *SCANFILE);

#Place it in the last array element
@sig_array[$i] = $signature;

#Change index value to be the index within a page
#not index within a whole file.  The remainder
#of the division of the index value by the size
#of a page will yield this value.
$index = $index % 4096;

#Add the index used for the last page to the
#last element of the signature array
@sig_array[$i + 1] = $index;

#Close file to be scanned
close(SCANFILE);

#Return the array of signatures
return @sig_array;
}
```

D. MATCH_SIGNATURES.PL

```
#!/usr/bin/perl
#This is a program that will look for pages in a file that match
#the signatures in a signature file.  Both files are specified
#by the user.
```

```
use strict;
use POSIX qw(ceil floor);     #Adds floor and ceiling functions

#-------------------------------------------------------------------
#Declare variables
#-------------------------------------------------------------------
my @sigs;               #Array of signatures
my $sig;                #A particular signature
my $last_index;         #The index used in the signature of the
                        #last
                        #page
my $length;             #Length of sigs array
my $v = 0;              #Verbose mode flag
my $base;               #Base value where the search of consecutive
                        #pages
                        #will begin.
my $in_file;            #File that will be parsed for matching pages
my $sig_file;           #File that holds a list of signatures to look
                        #for
my $file_size;          #Size of above file
my $user_in;            #User input

#-------------------------------------------------------------------
#Get Inputs
#-------------------------------------------------------------------
#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-f”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then kill program
if($user_in eq “-m” or $user_in eq “”)
      {
die “Name of file to be parsed is invalid.  Error found”;
      }

#Save string after -f in dump_file variable
$in_file = $user_in;

#Advance input
$user_in = shift;

#If a -m is found, then continue
if($user_in eq “-m”)
      {
#Advance input
$user_in = shift;

#If the output file name is there, then
#put input value in dir variable
if($user_in ne “”)
```

```
            {
$sig_file = $user_in;
            }
#Otherwise kill program
else
            {
die “Signature database file name is missing. Error
                  found”;
            }
      }
else
      {
die “The -m flag is missing. Error found”;
      }
}
#Otherwise kill program
else
{
die “User did not use -f flag. Error found”;
}

#-------------------------------------------------------------------
#Body
#-------------------------------------------------------------------

#Extract signatures from each page of a file
@sigs = get_sigs($sig_file);

#Get length of array received from get_signatures
$length = scalar @sigs;
if($v==0){print “LENGTH OF ARRAY: $length\n”;}

#Save the index where the last pages signature
#was extracted
$last_index = @sigs[$length -1];
if($v==1){print “Last index: $last_index\n”;}

#Remove the last page index from the end of the
#signature array
pop(@sigs);

foreach $sig (@sigs){
#print ”Page Sig: $sig\n”;
}

#Make base value zero
$base = 0;

#Open file to find matches in
open(MATCH, $in_file);

#Get the size of the input file to determine when
#to stop looping through pages
$file_size = -s $in_file;
print “SIZE: $file_size\n”;

#While the base address is smaller than the size
```

```
#of the file, try to match signatures
while($base < $file_size)
{
if($v==1){print “Base is $base\n”;}
match_against_signatures($base, *MATCH, $last_index, @sigs);
$base = $base + 4096;
}

#close file to find matches in
close(MATCH);

sub get_sigs
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
my $in_file = shift;    #A file holding a list of signatures

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------
      my $signature;          #An individual signature in $in_file
      my @sigs;               #Signature array built from $in_file
      my $i=0;                #Index of @sigs array

      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------

#Open the signature file
open(FILE, $in_file);

while($signature = <FILE>)
      {
#Take newline off of $signature
chomp($signature);

#Save signature in respective place in @sigs array
@sigs[$i] = $signature;
$i++;
      }

#Close signature file
close(FILE);

#Return built array
return @sigs;
}

#This function receives the address in an open file
#where a page begins and attemps to match that page
#to one of an array of signatures.
sub match_against_signatures
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
```

```
      my $base_addr = shift;        #Address where page begins in file
      my $handle = shift;           #File handle of file to be parsed
                                    #(file should already be open)
      my $last_page = shift;        #The index of where to look for the
                                    #signature of the last page
      my @signatures = @_;          #Array of signatures to look for

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------

      my $sig_beg;                  #signature read in from beginnig of
                                    #page
      my $sig_last;                       #signature read in from the
                                    #index in $last_page
      my $v = 0;                    #Verbose flag

      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------
if($v==1){
print “base: $base_addr\nhandle: $handle\n”;
print “last: $last_page\nSignatures: @signatures\n”;
      }

#Go to the correct place in the file
seek($handle, $base + 0x500, 0);

#Get signature from beginning of the page
$sig_beg = read_bytes(12, $handle);

#Go to the correct place in the file
seek($handle, $base + $last_page, 0);

#Get the signature from the index $last_page
#index to check if this page is the last page
#in the file
$sig_last = read_bytes(12, $handle);

#Print out both signatures
if($v==1){print “First sig: $sig_beg\nSecond sig: $sig_last\n”;}
#See if any of the two signatures extracted from the
#page match those in the signatures array
for (my $i=0; $i <= $#signatures; $i++)
      {
if($signatures[$i] =~ m/($sig_beg|$sig_last)/)
            {
print “MATCH PAGE $i\n”;
            }
      }
}

#This function retrieves a specified number of hex
#bytes from a file and returns them as one variable
sub read_bytes
{
```

```
      #-------------------------------------------------------------------
#Read in arguments
      #-------------------------------------------------------------------
      my $bytes = shift;            #Number of bytes to read from file
      my $handle = shift;           #File handle of file to be parsed
                                    #(file should already be open)

      #-------------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------------

      my $extracted_bytes = “;      #extracte bytes from file
      my $read_in;                  #byte read in from file
      my $v = 0;                    #Verbose flag

      #-------------------------------------------------------------------
#Body
      #-------------------------------------------------------------------

#Output number of bytes choosen by caller
if($v==1){print “The number of bytes to read is: $bytes\n”;}
#Read in $bytes number of bytes
for(my $j=0; $j<$bytes; $j++)
      {
#Read in a byte
sysread($handle, $read_in, 1);

#Change signature into hex
$read_in = sprintf “%02x”, ord($read_in);
#Append byte to extracted_bytes
$extracted_bytes = $extracted_bytes.$read_in;

      }

return $extracted_bytes;
}
```

E. FIND_TASK.PL

```
#!/usr/local/bin/perl
#This is a program that will attempt to enumerate task names
#and pids from a RAM dump in a SUSE 10 Linux machine.  It
#will be able to follow the links forward or backwards depending
#on the user input.
use strict;

#-------------------------------------------------------------------------
---
#Declare variables
#-------------------------------------------------------------------------
---

my $dump_file;                      #Input file being used (default
value)
my $user_in;                        #User input
my $input;                    #The input from commands run below
```

```
my $address_command;                #Stores the command to get the
address
                              #of init_task
my $init_task;                      #Address of init_task
my @point;                    #Array containing next and prev pointers
my $dir = 0;                        #Direction of the enumeration 0 is
#following next pointers and 1 is following
#prev pointers
my $v = 0;                    #Flag that controls verbose mode

#-------------------------------------------------------------------------
---
#Get Inputs
#-------------------------------------------------------------------------
---
#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-f”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then kill program
if($user_in eq “-d”  or $user_in eq “”)
      {
die “Image file name is invalid.  Error found”;
      }

#Save string after -f in dump_file variable
$dump_file = $user_in;

#Advance input
$user_in = shift;

#If a -d is found, then continue
if($user_in eq “-d”)
      {
#Advance input
$user_in = shift;

#If the direction value is 0 or 1, then
#put input value in dir variable
if($user_in eq “0” or $user_in eq “1”)
            {
$dir = $user_in;
            }
#Otherwise kill program
else
            {
die “Direction value is invalid. Error found”;
            }
      }
}
```

```
#Otherwise kill program
else
{
die “User did not use -f flag. Error found”;
}

#-------------------------------------------------------------------------
#Body
#-------------------------------------------------------------------------
#Open the image of memory that you want to analyze
open(DUMP,$dump_file) or die “Memory image file could not be opened.
Error found”;
print “Opened the file: $dump_file \n”;
#Use a filehandle and pipe to parse through the output of this command
open(OUT, “ls /boot |”);
#Go through every line of the output
while(<OUT>)
{
#If a line matches the system map pattern then save it into input
if($_ =~ m/System.map*/){
$input = $_;
      }
}

#Take the newline out of the input variable
chomp $input;

print “Looking in \”$input\” for init_task address.\n”;
#Now get the address of init_task using the System.map file
$address_command = “cat /boot/$input | grep \”D init_task\””;
#Save the output of the address command into init_address
$init_task = ‘$address_command‘;
#Take the newline out of the init_address variable
chomp $init_task;

#Extract the address from init_address
$init_task =~ m/(([0-9]|[a-fA-F])*)\s/;
$init_task = $1;

#Turn init_address into a physical address (subract 0xC0000000)
$init_task = hex($init_task) - 0xC0000000;
#Add 0x60 to make it point to the tasks member of the task_struct
#which is required by enumerate_task $init_task = $init_task + 0x60;
#Initialize next_task to init_task
@point[$dir] = $init_task;

#Traverse the task_struct linked list and get information from
#each one.  This “for” loop caps the number of processes at 250
#in the case that the “if” statement below is never entered.
for(my $i;$i<250;$i++)
{
#Enumerate information from task_struct
@point = enumerate_task(@point[$dir]);

#Convert pointer to decimal for enumerate_task function
@point[$dir] = hex(@point[$dir]);
#If the next task address is equal to the first one in the
```

```
#list (the address of the tasks member of init_task) then
#break the loop.  This is basically looking for the
#wrap around of the linked list so that it can stop looping.
if(@point[$dir] =~ m/$init_task/)
      {
last;
      }

}

#Close the output pipe
close(OUT);

#Close the image of memory that you analyzed
close(DUMP);

#This function goes to the offset specified in the input
#(which should be in hex) and returns a word (4 bytes) at
#that location. The file it searches should be open when
#this function is called.
sub read_val
{
      #-------------------------------------------------------------------
#Read in arguments
      #-------------------------------------------------------------------
my $handle = shift;
my $contents = shift;

#This intereprets the hex number as a decimal to use in seek
$contents = hex($contents);
if($v==1){print “Address seen by read_val is: $contents\n”};

      #-------------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------------
#The reason behind splitting the input is that I needed to
      #rearrange the order of the bytes from the way they exist in
      #memory—which is backwards.  For example, the word AB123456 in
      #memory is really the hex number 563412AB.
      my $input_byte_1;             #First byte of the input
      my $input_byte_2;             #Second byte  of the input
      my $input_byte_3;             #Third byte of the input
      my $input_byte_4;             #Fourth byte of the input

      my $hx;                       #Hexidecimal version of the above
                                    #inputs

      #-------------------------------------------------------------------
#Body
      #-------------------------------------------------------------------
#Go to the correct offset in image file and read selected bytes
seek($handle, $contents, 0);
sysread($handle, $input_byte_4, 1);
sysread($handle, $input_byte_3, 1);
sysread($handle, $input_byte_2, 1);
sysread($handle, $input_byte_1, 1);
```

```
#Rearrange the bytes and assemble them into hex
$hx = sprintf “%02x”, ord($input_byte_1);
$hx = $hx.sprintf “%02x”, ord($input_byte_2);
$hx = $hx.sprintf “%02x”, ord($input_byte_3);
$hx = $hx.sprintf “%02x”, ord($input_byte_4);

if($v == 1) {print “Result of read_val function is: $hx\n”};
return $hx;
}

#This function arranges the output from read_val into ascii
#letters for the executable name garnered from a task_struct
sub ascii_ize
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
my $characters = shift;
if($v==1){print “Input into ascii_ize is: $characters\n”;}

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------
#Each letter in the name must be converted into ascii
      #individually and rearranged.  Thus the 16 variables for the
      #typical 16 character maximum in executable names in a
      #task_struct. They are arranged into 4 words that will be ordered
      #from 1 to 4.  Within the words, however, there needs to be a
      #reversal of the order.  This is due to the fact that strings are
      #stored in the opposite direction of all numbers and pointers.
my $word1;
my $word2;
my $word3;
my $word4;

      my $whole_name;         #The fully constructed name of the
                              #executable
      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------
#Convert name to ascii in the correct order
$characters =~ m/(........)(........)(........)(........)/;
$word1 = ascii_ize_word($1);
$word2 = ascii_ize_word($2);
$word3 = ascii_ize_word($3);
$word4 = ascii_ize_word($4);

#Assemble all the words into the whole_name
$whole_name = $word1.$word2.$word3.$word4;

#Strip any non alphanumeric characters out
$whole_name =~ tr/a-zA-Z0-9\./ /cs;
return $whole_name;


}
```

```
#This function takes a word output from read_val and reverses the order
#while translating the hex characters into ascii to make the string
#readable
sub ascii_ize_word
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
my $characters = shift;

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------
#Each letter in the name must be converted into ascii
      #individually and rearranged.
my $word_4;
my $word_3;
my $word_2;
my $word_1;
      my $whole_name;         #The fully constructed name of the
                              #executable
      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------
#Convert name to ascii in the correct order
$characters =~ m/(..)(..)(..)(..)/;
$word_4 = hex($1);
$word_3 = hex($2);
$word_2 = hex($3);
$word_1 = hex($4);

#Change each letter to ascii
$word_4 = chr($word_4);
$word_3 = chr($word_3);
$word_2 = chr($word_2);
$word_1 = chr($word_1);

#Construct name
$whole_name = $word_1.$word_2.$word_3.$word_4;

return $whole_name;

}

#This function receives the address of the tasks member of a
#task struct and enumerates the name of the process, its pid,
#the link to the next task struct, and the link to its associated
#mm_struct.  It also outputs the first word that is retrieved from
#the address pointed to by the pgd member of the mm_struct
#associated with the task struct. The input should be a decimal
#number. The handle of the dump file is global in this file so it can
#be used without passing below.
sub enumerate_task
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
```

```
my $init_address = shift;
#Print address passed in to enumerate_task
if($v==1){print “Address passed into enumerate_task:
            $init_address\n”;}
      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------
      my $next;                     #Pointer to the next task struct
      my $prev;                     #Pointer to the prev task struct
      my $pid;                      #Pid of the task struct in question
      my $part_name;                #Partial name before it’s assembled
                                    #below
      my $name;                     #Name of the process represented by
                                    #the
                                    #task struct
      my $mm;                       #Pointer to mm_struct from
                                    #task_struct
      my $pgd = ‘NULL POINTER’;     #Content of the address pointed to
                                    #by pgd
                                    #member of mm_struct
      my @ret;                      #array with next and prev addresses
                                    #inside that will be returned

      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------
#Derive other addresses from init_address
$next = $init_address;
$prev = $init_address + 0x4;
$mm = $init_address + 0x18;
$pid = $init_address + 0x3C;
$name = $init_address + 0x144;

#Turn addresses into hex numbers (in preparation for read_val
      #function)
$next = sprintf “%08x”, $next;
$prev = sprintf ”%08x”, $prev;
$mm = sprintf ”%08x”, $mm;
$pid = sprintf “%08x”, $pid;
$name = sprintf “%08x”, $name;

#Read the values at all of these addresses
$next = read_val(*DUMP, $next);
$prev = read_val(*DUMP, $prev);
$mm = read_val(*DUMP, $mm);
$pid = read_val(*DUMP, $pid);

#Convert mm to decimal in preparation for computation of pgd
#address.
$mm = hex($mm);
if($mm != 0){
#Compute address of pgd (0x20 from beginning) and turn
#it into a physical address (subtract 0xC0000000)
$pgd = $mm + 0x20 - 0xC0000000;
$pgd = sprintf “%08x”, $pgd;
if($v==1){print “Address of pgd within mm_struct: $pgd\n”;}
```

```
#Read the address pointed to by pgd
$pgd = read_val(*DUMP, $pgd);
if($v==1){print “Value of pgd: $pgd\n”;}

#Convert addres to physical address
$pgd = hex($pgd) - 0xC0000000;

#Convert the result to hex again for read_val function
$pgd = sprintf “%08x”, $pgd;
#Read the value stored at the address pointed to by pgd
$pgd = read_val(*DUMP, $pgd);
if($v==1){print “Value in address pointed to by pgd:
            $pgd\n”;}
      }

#Print out PID if verbose it turned on
if($v==1){print “Extracted PID is: $pid\n”;}

#For name we have to read 4 words because that is the maximum
      #length
#of an executable name in a task_struct
$part_name = read_val(*DUMP, $name);
#Read three more times to complete the 16 bytes that must be read
for(my $i=0;$i < 3;$i++)
      {
#Calculate the next address to look read in (4 bytes ahead)
$name = hex($name) + 4;
$name = sprintf “%08x”, $name;

#Read in and append to existing partial name
$part_name = $part_name.read_val(*DUMP, $name);
      }

#Put the contents of part_name into name
$name = $part_name;

#Convert name to ascii in the correct order
$name = ascii_ize($name);

#Turn next, prev and mm into physical addresses
#(subract 0xC0000000)
$next = hex($next) - 0xC0000000;
$prev = hex($prev) - 0xC0000000;

#If mm is not equal to zero, then convert to a physical address
if($mm != 0){
$mm = $mm - 0xC0000000;
      }

#Convert variable next to a hex value again
$next = sprintf “%08x”, $next;
$prev = sprintf “%08x”, $prev;
$mm = sprintf “%08x”, $mm;

#Convert variable pid to a decimal number
$pid = hex($pid);
```

```
#Print out all the information garnered from task_struct
print “-----------------------------------------\n”;
      print “Executable name is:             $name\n”;
      print “The pid of the executable is:   $pid\n”;
      print “The next task_struct is at:     $next\n”;
print “The previous task_struct is at: $prev\n”;
print “The associated mm_struct is at: $mm\n”;
print “Value pointed to by pgd:        $pgd\n”;
print “-----------------------------------------\n”;
#Assemble the return array
@ret[0] = $next;
@ret[1] = $prev;

return @ret;

}
```

F. ENUM_ADD_SPACE.PL

```
#!/usr/local/bin/perl
#This is a program that will enumerate all the different
#Address_space objects found in a memory dump
use strict;
use utils;

#------------------------------------------------------------------
#Declare variables
#------------------------------------------------------------------
my @addresses;                #Array of addresses found in dump file
my $addr_ref = \@addresses;   #Pointer to the array of addresses
my @clean_addrs;              #The version of addresses array without
                              #NULL entries or repitition
my $addr;                     #A single addr space address
my $dump_file;                #Memory image file to be searched for
my $out_file;                 #File to send all addr space addresses to
my $user_in;                  #User input

#------------------------------------------------------------------
#Read in arguments
#------------------------------------------------------------------
#Get dump file from user
$user_in = shift
or die “User did not input anything.  Error found”;
#Make sure that the input matches the correct format.
#If -f is found, then continue
if($user_in eq “-f”)
{
#Advance input
$user_in = shift;

#If the value of input is in line with a flag
#name or empty, then continue
if($user_in eq “-o” or $user_in eq “”)
      {
die “Memory image file name is invalid.  Error found”;
      }
```

```
#Save string after -i in dump_file variable
$dump_file = $user_in;

#Advance input
$user_in = shift;

#If a -o is found, then continue
if($user_in eq “-o”)
      {
#Advance input
$user_in = shift;

#If the output file name is there, then
#put input value in dir variable
if($user_in ne “”)
            {
$out_file = $user_in;
            }
#Otherwise kill program
else
            {
die “Output file name is missing. Error found”;
            }
      }
else
      {
die “The -o flag is missing. Error found”;
      }
}
#Otherwise kill program
else
{
die “User did not use -f flag. Error found”;
}

#---------------------------------------------------------------------
#Body
#---------------------------------------------------------------------
#Traverse al page descriptors and run enum_addr_space
#on them all
Utils::traverse_page_desc($dump_file, \&enum_addr_space, $addr_ref);
print “Done with initial printing!\n”;
#Remove zero elements and repeating elements from
#addresses array
@clean_addrs = clean_array(@addresses);

#Open out file for writing
open(FILE, “>$out_file”);

#Output the clean_addrs array to an output file
foreach $addr (@clean_addrs)
{
#Write out to file
print FILE “$addr\n”;
print “$addr\n”;
}
```

```
#Close output file
close(FILE);

#This function takes in an array and returns a version
#of that array without 0 values or repetition
sub clean_array
{
      #-------------------------------------------------------------
#Read in arguments
      #-------------------------------------------------------------
my @old_array = @_;           #Array to clean up

      #-------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------
      my @new_array;                #Cleaned up array
      my $old_element;              #An element of the old array
      my $new_element;              #An element of the new array
      my $add_old = 1;              #Flag for adding/not adding an
                                    #element to the new array
      my $v = 0;                    #Verbose flag

      #-------------------------------------------------------------
#Body
      #-------------------------------------------------------------
#Add the element from old array to new array only if
#it is not equal to zero or is already in the new array
foreach $old_element (@old_array)
{#print “old element: $old_element\n”;
#If the element is not zero then check if
#the element is in new array.
if(!($old_element =~ m/00000000/))
{#print “not zero\n”;
#Check if element is in new array
foreach $new_element (@new_array)
{#print “compare to : $new_element\n”;
if($old_element =~ m/$new_element/)
{#print “MATCH\n”;
$add_old = 0;
last;
                        }
                  }

#If the old element did not appear in
#new array, then add it
if($add_old == 1)
                  {
push(@new_array, $old_element);
if($v == 1){print “Added: $old_element\n”;}
                  }

#Reset flag
$add_old = 1;
            }
      }
```

```
#Return new array
return @new_array;

}

#This function enumerates all the different address_space objects
#by searching all links from page descriptors and outputs them
#to a file
sub enum_addr_space
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
      my $base = shift;       #Base address of page descriptor
      my $file = shift;       #Pointer to memory file
      my $ref = shift;        #Pointer to array where results
                              #should be stored

      #-----------------------------------------------------------------
#Declare variables
      #-----------------------------------------------------------------
      my $offset = 0x10;      #Offset where mapping (addr. Space
                              #pointer)
                              #can be found;
      my $location;           #Address get addr. space pointer from
      my $address;            #Address of the address_space object
                              #found

      #-----------------------------------------------------------------
#Body
      #-----------------------------------------------------------------
#Calculate the address to retrieve pointer from
$location = $base + $offset;
#Turn location into a hex number
$location = sprintf “%08x”, $location;

#Read pointer contents and save them to address variable
$address = Utils::read_val($file, $location);
#Write result into array
push(@$ref, $address);

}
```

G. UTILS.PM

```
package      Utils;
use strict;

#This function creates a formatted output outlining what address_space
#address was looked for in a particular run of the program.
sub heading
{
      #-----------------------------------------------------------------
#Read in arguments
      #-----------------------------------------------------------------
my $add_space = shift;
```

```
print “\n\n”;
print “*****************************************************\n”;
print “The following pages all have descriptors that point to the
same\n”;
print “address_space object.  This address is:\n\n”;
print “                          $add_space \n”;
print “*****************************************************\n”;
}

#This function takes in the address of a particular Address_space
#object and find all the page descriptors and pages associated with
#those objects in a memory dump
sub find_address_space
{
      #-------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------
      my $index;                    #The position of a page w/ respect
                                    #to other pages of same address
                                    #space
      my $page_desc;                #Address of page descriptor
      my $page_address;             #Page fram address
      my @answers;                  #Array of lines from an input file
      my $in_file = ‘match.txt’;    #Input file with page addresses and
                                    #indices
      my $dump_file;                #Memory dump file to look into
      my $line;                     #individual line in answers array
      my $address;                  #Address_space to look for

      #-------------------------------------------------------------
#Get Inputs
      #-------------------------------------------------------------
$dump_file = shift;
$address = shift;

      #-------------------------------------------------------------
#Body
      #-------------------------------------------------------------

#Traverse the page descriptors and perform get_match_info
#on each one
traverse_page_desc($dump_file, \&get_match_info, $address);
#Open the file used by get_match_info to
#output results
open(FILE, $in_file);

#Read in needed info into an array in the
#order dictated by index value of file (1st
#element of each line.
while(<FILE>)
      {
$_ =~ m/(.+)\s(.+)\s(.+)\n/;
$index = $1;
$page_desc = $2;
$page_address = $3;

#Store lines into answers array by their index
```

```
@answers[$index] = “Page $index is at $page_address w/ descriptor at
$page_desc\n”;
      }

#Print out pages found in order
foreach $line (@answers)
      {
print “$line”;
      }

#Close the file
close(FILE);

#Remove the input file because it is no longer needed
system “rm $in_file”;

}

#This function traverses the mem_map array and calls a function
#provided by the caller (through a pointer) at every page encountered.
#It passes the current page descriptor address to the function passed
#in.
sub traverse_page_desc
{
      #-------------------------------------------------------------------
#Read in arguments
      #-------------------------------------------------------------------
      my $dump_file = @_[0];        #The name of the file to look in
      my $function = @_[1];         #Pointer to function to perform as
                                    #the mem_map array is traversed
      my @arguments = @_[2..$#_];   #Array of arguments inteded for
                                    #funct pointed to by the above
                                    #argument

      #-------------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------------
      my $mem_map = 0x1000000;      #The physical address marking the
                                    #beginning of the mem_map page desc
                                    #array
      my $offset = 0x0;             #Offset for loop that controls
                                    #traversing the mem_map page desc
                                    #array
      my $location;                 #The current address being seen in
                                    #the loop
      my $upper_limit = 0x20000;    #Controls number of times the
                                    #traversal loop will run
      my $v=0;                      #Verbose flag

      #-------------------------------------------------------------------
#Body
      #-------------------------------------------------------------------
#Open the image of memory that you want to analyze
open(DUMP,$dump_file)
or die “Memory image file $dump_file could not be opened.
Error found”;
print “Opened the file: $dump_file \n”;
```

```
#Loop through page descriptor entries and apply passed function
#The loop stops at 0x20000 which is the value I calculated for
#the number of page descriptors needed to cover 512 MB of mem.
for(my $i=0; $i<$upper_limit; $i++)
      {
if($v==1){print “The offset is: $offset\n”};
#Calculate the current page address in physical memory
$location = $mem_map + $offset;
#perform function passed with function pointer
$function->($location, *DUMP, @arguments);
#Offset grows with loop
$offset = $offset + 0x20;
      }

#Close dump file
close(DUMP);
}


#This function takes in the address of a page descriptor and an address
#to match to.  If the address to match to is the same as the one held
#in the address_space pointer in the page descriptor, then we have a
#match and the page information is written to a file
sub get_match_info
{
      #-------------------------------------------------------------------
#Read in arguments
      #-------------------------------------------------------------------
      my $location = shift;         #The address of a page descriptor
      my $file = shift;             #Pointer to file to look in
      my $add_space = shift;        #The address to match to

      #-------------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------------
      my $hex_out;                  #Hexadecimal output from read_val
                                    #function
      my $page_desc;                #Address of the page descriptor
      my $page_address;             #Address of page that corresponds
                                    #to page descriptor
      my $index;                    #Index of the page in question
      my $mem_map_element_size = 0x20;    #Size of a page descriptor
      my $index_offset = 0x14;            #Offset where index resides
in page
                                    #descriptor
      my $mem_map = 0x1000000;      #Physical address of mem_map 16MB
      my $out_file = ‘match.txt’;   #File to write matches to.
      my $v=0;                      #Verbose flag

      #-------------------------------------------------------------------
#Body
      #-------------------------------------------------------------------
#Change the whole address to upper case
$add_space =~ tr/[a-z]/[A-Z]/;

#Retrieve the value of the current page descriptor’s pointer
#to the address_space struct that owns it.  The 0x10 is the
```

```
#offset into the page descriptor where the actual pointer is
#found.
$location = $location + 0x10;
#Turn location into a hex number
$location = sprintf “%08x”, $location;

if($v==1){print “The value sent in to read_val: $location\n”};
#Retrieve address_space pointer from dump file
$hex_out = read_val($file, $location);
if($v==1){print “The value returned from the read_val function:
$hex_out\n”};
#Change the output to all upper case for the comparison
$hex_out =~ tr/[a-z]/[A-Z]/;
#Compare the result to the address passed in in the parameter
if($hex_out =~ m/$add_space/)
      {
#Calculate the actual page descriptor address and interpret it
#as hex
$page_desc = hex($location) - 0x10;
$page_desc = sprintf « %08x », $page_desc;

#Calculate the page address in memory from the page
#descriptor address.  The equation is:
#((page_desc_address - mem_map_address)/0x20) << PAGE_SHIFT
$page_address = (hex($page_desc) - $mem_map);
$page_address = $page_address/$mem_map_element_size;
$page_address = $page_address << 12;
$page_address = sprintf “%08x”, $page_address;
#Retrieve the address where the index of the page is stored
$location = (hex($page_desc) + $index_offset);
$location = sprintf “%08x”,$location;
#Read the index of the page in question
$index = read_val($file, $location);

#Open out file for appending
open(FILE, “>>$out_file”);

#Write out to file
print FILE “$index $page_desc $page_address\n”;
if($v==1){print “MATCH at 0x$location, PAGE_DESC at
0x$page_desc, “;}
if($v==1){print “PAGE at 0x$page_address, INDEX is
$index\n”;}
#Close output file
close(FILE);

      }

}

#This function goes to the offset specified in the input
#(which should be in hex) and returns a word (4 bytes) at
#that location. The file it searches should be open when
#this function is called.
sub read_val
{
      #---------------------------------------------------------------
```

```
#Read in arguments
      #-------------------------------------------------------------------
my $handle = shift;
my $contents = shift;

#This intereprets the hex number as a decimal to use in seek
$contents = hex($contents);
my $v=0;
if($v==1){print “Address seen by read_val is: $contents\n”};

      #-------------------------------------------------------------------
#Declare variables
      #-------------------------------------------------------------------
#The reason behind splitting the input is that I needed to
#rearrange the order of the bytes from the way they exist in
#memory—which is backwards.  For example, the word AB123456 in
#memory is really the hex number 563412AB.
      my $input_byte_1;             #First byte of the input
      my $input_byte_2;             #Second byte  of the input
      my $input_byte_3;             #Third byte of the input
      my $input_byte_4;             #Fourth byte of the input

      my $hx;                       #Hexidecimal version of the above
                                    #inputs

      #-------------------------------------------------------------------
#Body
      #-------------------------------------------------------------------
#Go to the correct offset in image file and read selected bytes
seek($handle, $contents, 0);
sysread($handle, $input_byte_4, 1);
sysread($handle, $input_byte_3, 1);
sysread($handle, $input_byte_2, 1);
sysread($handle, $input_byte_1, 1);

#Rearrange the bytes and assemble them into hex
$hx = sprintf “%02x”, ord($input_byte_1);
$hx = $hx.sprintf “%02x”, ord($input_byte_2);
$hx = $hx.sprintf “%02x”, ord($input_byte_3);
$hx = $hx.sprintf “%02x”, ord($input_byte_4);

if($v == 1) {print “Result of read_val function is: $hx\n”};
return $hx;
}

1;
```

H. HEAP.C

```
/*heap.c
*by: Jorge Urrea
*2/15/2006
 *
*This program asks for 400MB of heap space and then
*proceeds to fill it with signatures of the form
*pid|BEEF|ABCD|page#. This is done to allow for an
*easy determination of what process wrote the page and
*what number page it is in the scheme through hex
```

```
*inspection.
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
//Declare variables
      int pid;          //The pid of this process
      int *mem;         //Pointer to heap allocated by malloc
      int *mem2;        //Pointer that advances through heap
      int cat;          //Variable to concatenate values in
      int page_count = 0;     //The page number being added to heap
      int i;                  //For loop iteration control

//Get pid
pid = getpid();

printf("The pid of this process is: %d\n", pid);
//Allocate heap space
mem = malloc(419430400);

//Initialize mem2 to mem
mem2 = mem;

printf("Mem 1 is: %d, Mem 2 is: %d\n", mem, mem2);
//Loop the placing of bytes in heap.  The number
//of iterations will fill up the whole space
//allocated.
for(i=0; i<52428800; i++){

//Assemble word to put into heap
cat = pid;
cat = cat << 16;
cat = cat | 0xBEEF;

//Put it into the heap
*mem2 = cat;

//Advance mem2 4 bytes
mem2 = mem2 + 1;

//If the sequence has been repeated 512 times, then
//increment the page count (512 x 8 bytes per loop =
//4096).  The modulo operation is to account for
//indexing at zero.  That is, if you want to change
//a page every 4 iterations you actually want iterations
//0-3 to be page 0, iterations 4-7 to be page 1, etc.
if((i%512 == 511)){
page_count++;
           }

//Assemble second word to put in heap
cat = 0xABCD;
cat = cat << 16;
```

```
cat = cat | page_count;

//Put it into the heap
*mem2 = cat;

//Advance mem2 4 bytes
mem2 = mem2 + 1;
      }

//Create an endless loop to maintain the persistance of
//This program
while(1){
      }

//Free the space allocated
free(mem);

return 0;
}
```

LIST OF REFERENCES

[1] Prosise, Chris, and Mandia, Kevin, and Pepe, Matt. *Incident Response and Computer Forensics, Second Edition*. McGraw-Hill Osborne Media, 17 July 2003.

[2] Federal Bureau of Investigation. "History of the FBI, Rise of International Crime: 1980's," http://www.fbi.gov/libref/historic/history/rise.htm, 2006. Last Visited: January 2006.

[3] Burdach, Mariusz. *Digital Forensics of the Physical Memory*. March 2005.

[4] Bovet, Daniel P., and Cesati, Marco. *Understanding the Linux Kernel (2nd Edition)*. O'Reilly Media, 1 December 2002.

[5] Bovet, Daniel P., and Cesati, Marco. *Understanding the Linux Kernel (3rd Edition)*. O'Reilly Media, 1 November 2005.

[6] Gorman, Mel. *Understanding the Linux Virtual Memory Manager (Bruce Perens Open Source)*. Prentice Hall PTR, 29 April 2004.

[7] Grance, Tim, and Kent, Karen, and Kim, Brian. *NIST Special Publication 800-61: Computer Security Incident Handling Guide*. January 2004.

[8] Brezinski, D., and Killalea, T. *RFC 3227: Guidelines for Evidence Collection and Archiving*. February 2002.

[9] Burdach, Mariusz. "Finding Digital Evidence in Physical Memory." *2006 Black Hat Federal Conference*. Sheraton Crystal City, Washington DC. 25 January 2006.

[10] Federal Bureau of Investigation, "2005 Computer Crime Survey Report," http://www.mitnicksecurity.com/media/2005%20FBI%20Computer%20Crime%20Survey%20Report.pdf, 18 January 2006. Last visited March 2006.

[11] Rollins, John, and Wilson, Clay "Terrorist Capabilities for Cyberattack: Overview and Policy Issues," http://italy.usembassy.gov/pdf/other/RL33123.pdf, 20 October 2005. Last visited March 2006.

[12] Carrier, Brian D., and Grand, Joe, "A Hardware-Based Acquisition Procedure for Digital Investigations," *Digital Investigation Journal* 1(1), http://www.computer-tutorials.org/whitepapers.php, February 2004. Last visited March 2006.

[13] Garner, George M. Jr., and Mora, Robert-Jan "Response to Specific Questions Posed by the DFRWS 2005 Memory Challenge," http://www.dfrws.org/2005/challenge/index.html. 6 August 2005. Last visited March 2006.

[14] Betz, Chris. “DFRWS 2005 Challenge Report,” http://www.dfrws.org/2005/challenge/ChrisBetz-DFRWSChallengeOverview.html. August 2005. Last visited March 2006.

[15] Linux Man Pages, “DD.” Last visited March 2006.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chris Eagle
   Naval Postgraduate School
   Monterey, California

4. Dr. George Dinolt
   Naval Postgraduate School
   Monterey, California

5. Dr. Cynthia Irvine
   Naval Postgraduate School
   Monterey, California

6. Jorge Mario Urrea
   Naval Postgraduate School
   Monterey, California