ANALYSIS OF WINDOWS MEMORY FOR FORENSIC INVESTIGATIONS

Seyed Mahmood Hejazi

A thesis in The

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science in Quality Systems Engineering

Concordia University

Montréal, Québec, Canada

April 2009

Your file Votre référence
ISBN: 978-0-494-63196-6
Our file Notre référence
ISBN: 978-0-494-63196-6

Abstract

Analysis of Windows Memory for Forensic Investigations

Seyed Mahmood Hejazi

Containing most recently accessed data and information about the status of a computer system, physical memory is one of the best sources of digital evidence. This thesis presents new methods to analyze Windows physical memory of compromised computers for cyber forensics. The thesis includes three distinct contributions to cyber forensics investigation. Firstly, by digging into details of Windows memory management, forensically important information and data structures are identified. Secondly, we proposed different methods to find files and extract them out of memory in order to rebuild executable and data files. This helps investigators obtain valuable information available in executable or data files that have been in use at incident time. Thirdly, we presented two methods for extraction of forensically sensitive information such as usernames or passwords from memory. The first method is based on fingerprints of applications in memory. In the second method, we have been able to locate and extract arguments used in function calls. This method, leads to the acquisition of important and forensically sensitive information from the memory stack. Finally, to bring these contributions to application level, a framework for cyber forensics investigations has been developed that helps finding sensitive information.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Considering the day by day increase of IT role in our routine life, human society feels the need for awareness about increasing number of criminals that use IT and digital devices or networks to commit crimes such as frauds, theft, abuse, and even violent crimes.

Being similar to traditional forensic science, digital forensic can be defined as forensics for computer crimes where digital media are somehow involved in the case. Cyber forensics is a relatively new field that encompasses issues of traditional forensics in addition to technical issues of cyber space. In addition to having good knowledge of the regulations of the investigated environment, members of the investigation team should be equipped with state-of-the art tools and techniques to be able to overcome the technological challenges that they are facing.

1.1 Motivation

Due to the growing number of disputes over IT-related issues such as mass marketing frauds, identity thefts, and even national security threats, forensic investigation of cyber crimes as well as solving the problem of IT disputes is vital. In addition to engineering secure software and systems, hardening networks, and security monitoring, investigating cyber incidents is necessary. Physical memory, holding a big amount of useful data, is a valuable source of digital evidence while conducting a digital investigation. Physical memory contains pieces of evidentiary information that might not be found in any other computer media, it is harder to tamper with (comparing to information available on disks), and it is much more volatile. Thus, special tools and techniques are needed for forensic analysis of this potential source of evidence. One of the most important activities of a digital investigation is preservation of the crime scene that can be any digital device. It is well known that almost all contents of physical memory can be lost by rebooting or overridden by altering the status of the compromised machine. Thus, one of the most helpful steps of memory contents preservation could be dumping the contents of the physical memory for further analysis, as early as possible. Furthermore, physical memory contains pieces of information such as names, telephone numbers, usernames, and passwords that can play a vital role in an investigation. Extraction of these items as well as knowing the context that they belong to (such as certain files), can lead to identification of a criminal.

On the other hand, due to the increasing volume of the available physical memory

and complexity of its analysis, automated and user-friendly tools, decrease dependency of the analysts to thorough knowledge of memory management details. Paying attention to the aforementioned motivations, the result of this research is aimed to address these special needs. It is important to note that due to large differences between memory management details in various operating systems, we focus on a single operating system family. Sine most of the machines used by computer users are still running under Windows, we choose to do our research for Windows operating system. It should be noticed that due to the differences between structures of kernel object in different versions of Windows, we select Windows XP (SP1 and SP2) for our research. The research and the developed toolkit are extensible to other versions of Windows by making changes to kernel objects and virtual address translation mechanism.

1.2 Objectives

The overall objective of this thesis is to develop new techniques and tools for forensic analysis of physical memory. This can add to the value of analysis of compromised computers and consequently may decrease analysis time by finding more evidence that can also be correlated with other evidence found by other types of analysis. Moreover, since tampering memory contents is harder than contents of other media, physical memory analysis can yield more reliable evidence for the court of law.

To provide more details, we select two critical areas in physical memory investigation:

1. Extraction, partial or full, of files from memory contents.

2. Extraction of forensically sensitive information from memory contents, which usually gives clues about the incident.

To meet these objectives, it is necessary to carefully study the memory management in the target operating system (Microsoft Windows) combined with forensic analysis techniques, state-of-the-art methods, free-ware and commercial tools in addition to current nation-wide and world-wide standards. Extraction of files as well as other memory resident sensitive information (such as usernames, passwords, and IP addresses) can complement existing techniques, which mainly focus on processes, threads and in some cases rootkit detection. Since none of the previous works is able to extract files or pieces of sensitive information, by reaching these goals, we believe that the community will benefit from accessing more reliable and detailed evidence.

1.3 Contributions

In order to meet our first objective, we have developed two new methods for extraction of files (executable and data files) from memory contents of a suspect system. These methods are implemented as a part of the Memory Plug-in in Forensic Analysis Framework, created in Computer Security Laboratory of Concordia University. The file extraction methods have been successfully tested and used to extract files associated with each process. In many cases we have been able to extract text, HTML,

PDF, and many other data file types as well as most (more than 90 percent) of the executable files of the processes. The extracted executable files can be used to reproduce the behavior of the program and the data files can be analyzed to find clues.

To achieve our second objective, sensitive information extraction, we adopted two different approaches as follows:

- The first approach is based on finding protocol or application-specific patterns (conventionally called fingerprints) that precede or follow sensitive information when they are present in memory. By knowing these fingerprints, the investigator can search for them in memory contents (or the memory image) and locate sensitive information adjacent to them. Not only have we presented some of these fingerprints, but we have presented the method of finding these fingerprints for different applications. Thus, investigators can follow the same method to build their own database of fingerprints.

- The second approach is based on analyzing stack frames and comparing them to functions that deal with sensitive information (such as usernames or passwords) in order to locate this information, extract them, and present them to the investigator.

All of the aforementioned contributions have been implemented as a part of the memory analysis plug-in, which is capable of performing a thorough analysis of the memory images. This analysis includes extraction of Windows kernel objects, presenting forensically important attributes of these objects, file extraction, and stack

analysis. The functionalities of this plug-in are explained more in Chapter 6.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 starts with definitions of digital forensic science and related terms. It then describes why physical memory is a good source of digital evidence and sheds some light on the way memory is managed and used in Windows operating system.

Having this knowledge, we move to the next chapter. Chapter 3 introduces the related work that has been done on acquisition and analysis of physical memory for the purpose of forensic investigations. This chapter summarizes the main contributions in this context and enumerates advantages and shortcomings of these works.

Chapter 4 explains our new methods for extraction of process information and then file extraction from physical memory. It demonstrates how extracted executable and data files along with process information can be helpful during an investigative case.

Chapter 5 introduces other sensitive information that can be extracted from physical memory. It explains our new approaches to finding sensitive information such as usernames, passwords, visited URLs, and encryption keys. It then describes how our methods are different from existing ones and how these methods are able to find information that is not accessible by using existing approaches.

In Chapter 6, we introduce our developed forensic memory analysis plug-in, which

is a part of our forensic analysis framework. This toolkit takes advantage of the implementation of proposed methods in previous chapters and helps investigators perform a thorough analysis of physical memory modules of suspect computers.

Chapter 7 summarizes our research and emphasizes on the usefulness and originality of the proposed methods.

Chapter 2

Background

In this chapter, we will introduce key concepts of digital forensics and especially memory analysis in order to ease the understanding of the techniques that will be presented throughout the thesis.

In order to extract data from memory, we should know where they are located during their presence in memory and how to access them. Therefore, we should leverage our knowledge about physical memory and its allocation. The first step in our analysis is to find memory internal structures. Most of these structures maintain to other structures, which help us extract information about processes, threads, files, etc. Hence, we begin with knowing these structures. In order for these structures to point to other structures or memory locations, they use virtual addresses that need to be translated to physical addresses. Thus, we should also know the addressing modes and the address translation mechanism.

Since other contributions of this thesis include extraction of data files and executable files as well as arguments of functions on the stack, we should also study how files are mapped into the physical memory and how stack allocation is done in the target operating system.

Next sections shed some light on Windows memory management and the sensitive information that can be extracted from physical memory.

2.1 Digital Forensics

As one of the pioneers in development of digital forensic science, in 2001, the Digital Forensic Research Workshop (DFRWS) [1] gathered definitions proposed by several groups and proposed the following definition considering multiple perspectives.

> *"The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations."*

In digital forensics science, investigators, while considering the judiciary regulations, try to find, preserve, seize, and analyze digital evidence (defined below) in order to make court-admissible conclusions for the purpose of disputes. Investigators search for document files, images, videos, or log files inside the digital media. This media

can be a CD, a USB key, a hard disk drive, a network utility, a physical memory module, etc. This way, they can find clues about the incident, its time, and involved persons or groups and their intents.

For a better understanding of the digital forensics and eliminating ambiguities, we surveyed definition of some important concepts:

Incident An incident is a sequence of actions that compromises an information system's security, privacy, or functionality. The scale of the information system stated above can vary from a single desktop computer to a very large enterprise network [2, 3].

Investigation An investigation is a series of actions performed in a systematic and professional manner in order to gather and assess information to determine if a crime has been committed. An investigator is an authorized-by-law individual or team who has the permission to do the investigation or a part of it [4].

Event As stated in [5],

> *"A digital event is an occurrence that changes the state of one or more digital objects."*

An example of a digital event could be establishing an FTP connection to a server.

Digital evidence A digital evidence is piece of data, metadata, or information in any known digital format that can help to clarify an incident [3] such as a

document or a single line of a log file.

Physical evidence A physical evidence is any media that potentially carries digital evidences [5] such as a physical memory module or a CD.

Evidence admissibility Evidence admissibility is the degree to which evidences are acceptable by the jury and the court of law. As evidence would eventually be used to convict people of crimes, or to make an appeal, there exist certain requirements for evidence to make it admissible. In digital forensics, because of the volatile and easily tampered nature of digital media and data, this attribute of evidences is more considerable.

As explained above, one of the physical evidence that can be used to find digital evidence is the physical memory module of the suspect computer. The rest of this chapter explains where and how the digital evidence is stored in physical memory and how we can extract and analyze it.

2.2 Windows Memory Internal Structures

Windows keeps information about objects available in or accessed through memory, in various *kernel structures*. These structures have some fields inside that are actually pointers (links) to other structures. There is no thorough documentation about Windows memory structures and the uses of their member fields. Thus, in order to reach a specific field or piece of information about a particular Windows object, we have reverse-engineered these structures and followed the links between them. Tools

such as *Windows Debugger* [6] and utilities from *Sysinternals* [7] helped us very much to do this research. Figure 1 shows a part of the relations between most important Windows memory structures.

Below is a description of some of the structures that we used and their most important members:

- *EPROCESS*: All running processes in Windows are represented by an `EPROCESS` (Executive Process) block. The `EPROCESS` structure is an opaque structure that serves as the process object for a process [8]. This data structure contains many important attributes related to the running process such as:
  - `CreateTime`: Holds the time of the creation of the process.
  - `ExitTime`: Holds the time of the exit of the process.
  - `ProcessID`: Holds the unique ID assigned to this process.
  - `ImageFileName`: Holds 16 characters of the name of the process image (if more, other characters are truncated).
  - `SectionBaseAddress`: Points to the beginning address of the process image in memory (very helpful for file extraction).

  Beside these attributes, `EPROCESS` block points to some other structures such as *PCB (KPROCESS)*, *ObjectTable*, *SectionObject*, *PEB (Process Environment Block)*, and many more.

- *Process Environment Block (PEB)*: Contains some attributes and properties about the running process. We can point to *ImageBaseAddress*, which is the

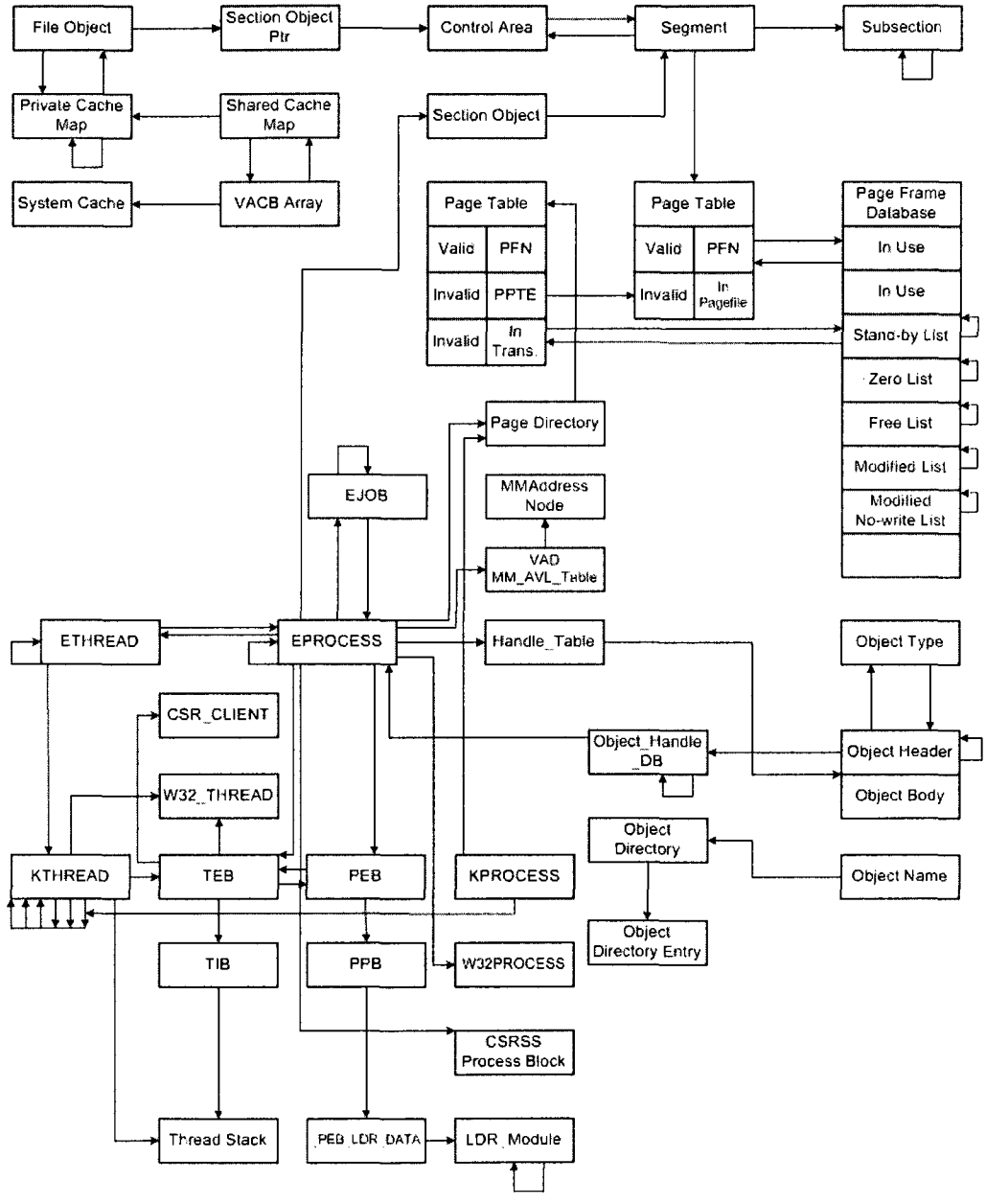

Figure 1: Windows Internal Structures and Their Inter-Relationships

starting address of the executable file loaded into the memory, as one of the important attributes of this structure.

- *FILE_OBJECT*: Being generated for each file that is created by Windows I/O functions, this structure represents a file in the kernel. To be more precise, a file object presents an open instance of a file, directory, device, or volume. This structure, includes some important attributes such as `FileName`, `SectionObjectPointer`, and `PrivateCacheMap` (explained later).

- *ETHREAD*: This structure represents a thread object that is created by a process. During the course of execution. These threads are represented by an `ETHREAD` structures that are linked together. `EPROCESS` structure points to the head of this list (*ThreadListHead*) so that it keeps reference to all threads that it has created. `ETHREAD` structure keeps valuable information about each of these threads including:

  - `Thread Control Block (TCB)`: A `KTHREAD` structure (kernel thread, explained afterwards).

  - `Cid`: Contains the process ID and thread ID for this thread.

  - `ThreadsProcess`: Points back to the `EPROCESS` structure associated with this thread.

- *KTHREAD*: The kernel presentation of a thread that is mostly used by the kernel for thread scheduling. The `KTHREAD` structure is extensively used in our

stack analysis method. This structures is pointed to by `ETHREAD` (via `TCB` field) and includes important member fields such as:

- `KernelTime`: The amount of time during which the thread was executing.
- `InitialStack`: Contains the base address of the stack.
- `StackLimit`: The largest address to which the stack can be extended.
- `KernelStack`: The current address of the stack pointer.

2.3 Virtual Memory and Address Translation

In Windows and most of other modern operating systems, applications and processes, access physical memory through a system of Virtual Addresses. However, only a few parts of operating system's kernel can use physical addresses to access Random Access Memory (RAM).

It should be noticed that all of the addresses that we find and use to follow links are described in virtual addressing mode. Since physical addresses are finally used to read fragments of data from memory image, it is necessary to have a good understanding of the virtual memory and the process of translating virtual addresses to physical addresses. In Windows (all versions), as in any operating system, a process is basically expected to have:

- A virtual address space.
- One or more threads of execution [9].

As described by Russinovich and Solomon [10], in order to handle the memory needs of a process, 32-bit operating systems that implement virtual memory support, allocate a 4 GB Virtual Address Space to each active process. The virtual address space of a process can be either less or greater than the volume of the available physical memory. The term "Working Set" refers to the portion of the virtual address space that is present in the physical memory. The addresses manipulated by programs' instructions are virtual addresses, which should be translated to physical addresses. Each virtual address is translated by the hardware to a physical address using a "virtual to physical address translation mechanism". Depending on the operating system mode, the hardware uses 32 bits (normal mode) or 36 bits (`PAE` mode) for addressing the physical memory [11]. On most 32-bit (IA-32) Intel Pentium Pro and later platforms, the Physical Address Extension (`PAE`) supports addressing up to 64 GB of physical memory for running applications [12]. Page directories and page tables are extended to 8-bytes format, which in turn allows extending base addresses of page tables and page frames to 24 bits. These extra four bits (base addresses of page table and page frames are 20 bits in normal mode) allow the `PAE` mode to use 36 bits instead of typical 32 bits. Figure 2 demonstrates how bits of a virtual address are used to translate it to a physical address in `PAE` mode.

As we have to access the memory image file and read streams of bytes directly by specifying the bytes to be read (physical addresses), we have developed a module that performs virtual to physical address translation. This module is now limited to systems working in the `PAE` or IA-32e memory model [13]. Therefore, to use our

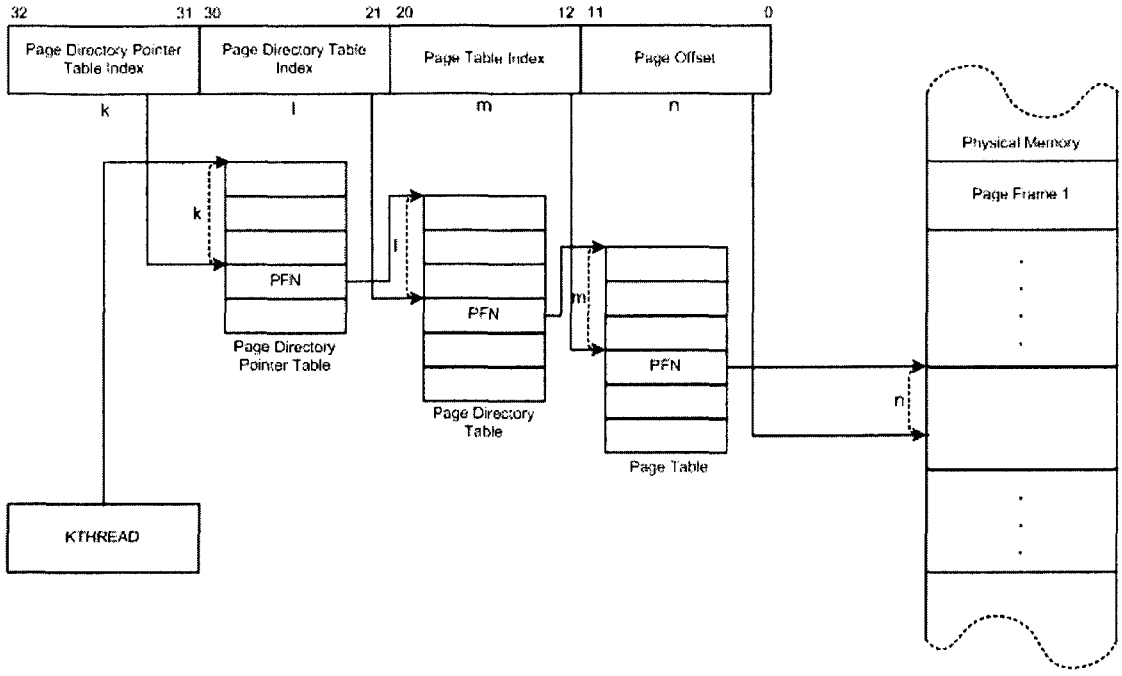

Figure 2: Address Translation in PAE Mode

toolkit's address translation module, the operating system should satisfy one of the conditions listed below:

- The `/PAE` switch is present in the file boot.ini.
- The `DEP` (Data Execution Prevention) feature is enabled (`/NOEXECUTE` switch is present). In this case, the `PAE` mode can be enabled automatically without the `/PAE` switch.
- The processor supports hardware-enforced `DEP`. Presence of the `/NOEXECUTE` switch on a system with a processor that supports hardware-enforced `DEP` implies the presence of `/PAE` switch [14].

Note that the address translation mechanism for *non-PAE* machines is very similar to

(and even less complex than) the translation mechanism used for PAE mode. Hence, it is easy to adopt this mechanism and use it for *non-PAE* machines. The reason that the current implementation is based on PAE mode is that most of the machines nowadays, have large volumes of physical memory and work in PAE mode.

2.4 Windows Memory Layout

Windows operating system uses the virtual memory concept to manage the system memory. In this context, the set of all virtual addresses that are available to a process is called its virtual address space [15]. The virtual address space is divided into two ranges: *user space* and *system space*. *User space* is the range of addresses that user-mode processes, process specific data and user-mode DLL files are mapped into. This range is from address 0x00000000 to address 0x7FFFFFFF. *System space* a.k.a. *kernel space* is the range of addresses in which the operating system resides and is only accessible to kernel-mode code and through kernel data structures. This restriction provides a security level in which threads cannot read/write the data from/to the memory space that does not belong to them. This space ranges from address 0x80000000 to address 0xFFFFFFFF.

Almost all the implementations of virtual memory divide the virtual address space into blocks of adjacent virtual addresses called *pages*. These *pages* can be active; therefore, they reside in the physical memory. Pages can also be inactive; therefore,

they would be stored on the disk. In modern programming, dynamic memory allocation enables programs to allocate and use memory space at runtime [16]. Memory pool allocation is a method adopted for dynamic memory allocation. Many operating systems as well as Windows, provide pools of paged and non-paged memory that can be allocated to processes. Pool memory is a not necessarily contiguous space of memory that is available to the processes and threads.

Another method of allocating dynamic memory is stack allocation in which data are added and removed in a Last-In-First-Out and in a faster manner. The kernel stack is a limited source of memory for holding local variables of functions and parameters passed to them.

2.5 Call Stack and Stack Frames

A call stack, being a part of memory, is a stack (Last In First Out) structure used by the operating system to store information about the active subroutines of each program. This structure is also known as execution stack, control stack, or simply stack. The stack is used to pass arguments from a *caller* subroutine to a *called* subroutine, store local variables of subroutines, and store the address to which the control of the program should be transferred after a subroutine finishes (*return address*). Stacks are usually allocated to each thread of process execution. Thus, each thread has its own stack. Below, comes a short description of uses of the stack:

- *Storing the return address:* Whenever a function calls another one, before the

control reaches the *Call* instruction, the caller should *push* the arguments (to be passed to the called function) onto the stack as well as the address of the instruction that comes immediately after the *Call* instruction (*return address*) in the caller. When the *Call* instruction executes, it transfers the control of the program to the called function. However, when the called function finishes its execution, the program control should return to the caller function right after the call instruction. Hence, the return address is *popped* from the top of the stack. Let us consider now the case in which nested functions call each other. For example, a function that draws a rectangle - `DrawRect()` - calls a function that draws a line - `DrawLine()`. If the `DrawLine()` in turn calls another function, all of these nested functions will stack up their own return address and so on.

- *Local variables storage:* A function usually needs local variables (variables that are known only in the context of the function and do not retain their value and usage outside the scope of the function) for the proper functioning. These variables can be held on the stack, since each called subroutine has its own stack space and the access would be faster than *heap* allocation.

- *Parameter passing:* Most of the time, functions pass parameters to called subroutines. When there are few small parameters, caller may use CPU registers for this purpose (hence, does not use memory), but when it comes to more and bigger sized parameters, stack area can be used to pass parameters.

A *stack frame* is the block of information stored on the call stack as a result of a subroutine call. A stack frame, in general, contains all the information required to save and restore the state of a procedure. These frames, each associated with one procedure call, contain arguments (parameters) passed to the function, local variables, and the return address. Physically, a function's stack frame is the fragment of memory between the addresses contained in the `ESP` register (the stack pointer) and the `EBP` register (the frame pointer or base pointer in Intel terminology). The most important registers that interfere in call instructions are the following:

- *EIP*: Instruction Pointer holds the address of the instruction, which will be executed by the CPU.
- *EBP*: Base Pointer, also known as frame pointer, is used to allow access to function arguments and local variables in the stack frame.
- *ESP*: Stack Pointer always points to the top of the stack, which is the last element used on the stack.

2.6 Sensitive Information in Memory and Possible Correlation

Memory is like a game table for all running applications and processes. To be a part of the game, data should be brought to this table. These data include, but are not limited to executable code of the processes, data files accessed by processes, URLs

accessed via web browsers, usernames, and passwords. We have classified the data resident in memory into the following categories:

1. Metadata
2. Files
3. Sensitive data
4. Case irrelevant data

2.6.1 Metadata

Metadata is the data that explain or clarify other data. Examples of metadata existing in memory are (1) the number and names of running and terminated processes, (2) start and end time (if the process has ended) of each process, (3) names of the files that have been accessed by each process, and (4) DLL files that have been used by processes during the course of their execution. Metadata can be very important from the forensics point of view and may yield conclusive evidence. As an example, assume that an investigator has been able to find the names (and types) of files accessed by a specific process (metadata), but the method in use could not reveal the contents of the files. If a file named "`John_Doe.jpg`" is found among the accessed files, although the investigator does not see the picture file, he can assume that a photo that is probably related to John Doe has been accessed at the time of incident. Then the investigator knows that she/he has to gather information about John Doe.

2.6.2 Files

Files are of great significance when conducting an investigation. Files contain valuable information such as images and other sensitive data. For the purpose of forensic analysis, we have classified files into two categories; *executable files* and *data files.* Data files, being in various formats such as PDF, DOC, XLS, TXT, and image files, may contain valuable information about the owner of the suspect computer, the possible unauthorized user, or the persons contacted via the suspect computer. Finding pieces of log files or documents can be extremely helpful since they may reveal evidence such as names, addresses, account credentials, and other directing information. As an example, note the sample scenario stated in a challenge question of *Honeynet* web site [17]:

> *"A suspect has been arrested on charges of selling illegal drugs to high school students. The police also seized a single floppy disk, but no computer and/or other media was present in the house..."*

In this scenario, investigators have found names of the schools in which the suspect wanted to sell drugs, as well as the nickname of another contact person encoded in an image, using *Excel* application. The captured media in this example is a floppy disc, but suppose that the file was never saved on a disk and could be extracted from memory contents. Regardless of the media that files can be extracted from, file contents can yield valuable information in an investigative case.

2.6.3 Sensitive Data

By sensitive data, we mean pieces of data such as usernames, passwords, encryption keys, or URLs that have been used by users as they were interacting with the machine under investigation. In many cases, these pieces of information are not parts of files and are passed to functions inside processes' code as parameters. Regardless of the parameter passing mechanism used in the application, parameters are stored in memory locations before being used by the application. In many cases these memory locations will not be overwritten after the parameters are used. Finding these kinds of data in memory can help the investigator to reach the right person, right time, or right physical locations.

2.6.4 Case Irrelevant Data

In any investigative case, there are some data that seem to be irrelevant to the case. Irrelevant data include pieces of data that do not hold any clues about what an investigator is searching for such as operating system specific data, or data that fall outside of times of interest [18, 19]. By vast increases in volume of memory modules and complexity of the applications, the volume of irrelevant data also increases. On the other hand, sensitivity of investigation time in forensic cases adds to the importance of eliminating irrelevant data from analysis.

2.6.5 Evidence Correlation

When investigating and analyzing the memory, finding possible correlation between gathered evidence can add to credibility of conclusions or even refuse assumptions. Thus, detailed and precise information obtained from memory can be very important when analyzing the correlation between evidence. For instance, a deleted record in a log file can be acquired from memory as a part of an extracted log file. Likewise, metadata acquired from process information can be used to reject a manipulated log file. On the other hand, assume that a suspect has entered a username and a password in a Secure Socket Layer (SSL) enabled web page. This information can be extracted from memory and afterwards be used for analyzing other collected evidence.

Chapter 3

State-Of-The-Art

In this chapter, we will introduce principle methods and tools available in the field of forensics memory analysis. After an introduction on general digital forensics frameworks, this chapter introduces state of the art techniques and tools in memory analysis and enumerates their advantages and shortcomings. Based on our detailed study, memory forensics techniques (and their corresponding tools) fall into two main categories of "Acquisition Methods" and "Analysis Methods". To be able to correctly identify weaknesses and strengths of present techniques, we have used the present methods and compared them. Based on this study, the chapter ends with an evaluation and comparison of the previous works.

3.1 Digital Forensics Workstations and Platforms

There are numerous tools available for digital forensic investigation in order to either collect or analyze data and provide functionalities for finding data on a computer and protecting evidence in a synchronized manner. These tools can be divided into the following categories:

3.1.1 Hardware-Included Platforms

These tools are based on the interaction between software and hardware components. In this category there are some packages that include forensic equipments such as workstation, servers, laptops and so on. Examples of these all-inclusive packages are:

- DIBS Mobile, Advanced, and Aircapture WLAN 14 Workstations [20] are three different versions of computer forensic hardware-included forensics tools. They are "specifically designed to copy, analyze and present computer data in a forensically sound manner." They can be used on-site, in the laboratory and even to capture, store and analyze the contents of airborne communications. This portable and dependable system is accepted in court of low throughout the world.

- DRAC Hardware [21] is "a series of forensic computer systems that make the investigation and recovery of *digital artifacts* as fast and straightforward as possible". It includes various series each of which has high data recovery capabilities such as high volume hard drives and many expandable slots for different

storage media.

- FRED (Forensic Recovery of Evidence Device) is a family of forensic workstations that are "highly integrated, flexible and modular". According to the product specifications, "FRED systems are optimized for stationary laboratory acquisition and analysis. FRED will acquire data directly from IDE, EIDE, ATA, SATA, ATAPI, SCSI-I, SCSI-II, and SCSI-III hard drives and storage devices and save forensic images to DVD, CD, or hard drives. FRED systems are also able to acquire data from floppies, CD-ROM, DVD-ROM, Compact Flash, Micro Drives, Smart Media, Memory Stick, Memory Stick Pro, xD Cards, Secure Digital Media and Multimedia Cards."

3.1.2 Software-Based Toolkits

These tools are based on software components only. As of digital forensics software, we can mention the following:

- Sleuth kit and Autopsy Browser [22] are open source digital forensic tools that allow analyzing file systems of any operating system (Windows, Unix, etc.). They can be used to perform forensics disk analysis of `NTFS`, `FAT`, `Ext2`, `Ext3`, `UFS1`, and `UFS2` file systems. The Sleuth Kit is a collection of command line tools while the Autopsy Forensic Browser is an HTML-based graphical interface for the command line tools in the Sleuth Kit.

- The Access Data Forensic Toolkit [23] is one of the most popular commercial

tools that performs forensics analysis. This toolkit provides a panoply of tools including forensic toolkit, registry viewer and password recovery toolkit, distributed network attack and other tools in order to investigate and analyze digital data.

- EnCase Enterprise [24] is a popular network-enabled and multi-platform enterprise investigation toolkit with an intuitive GUI and powerful scripting engine providing an immediate response throughout forensic analysis. Encase Enterprise, being comfortable to use, works by combining the following five components:
  - The SAFE (Secure authentication for EnCase): A server into which the *Examiner* logs for authentication and authorization.
  - The Enterprise Examiner: A software that sends requests to and analyzes volatile data from a target node.
  - Servlet: A nonintrusive, auto-updating passive piece of software installed on workstations and servers to analyze suspect computers. Connectivity is established between the *SAFE*, the *Servlet*, and the *Examiner* to analyze and acquire devices that have the EnCase servlet installed.
  - Enterprise Connection: It is a secure virtual connection established between the Examiner and target machines.
  - Incident response analysis: Captures volatile data and generates reports.

3.1.3 Digital Evidence Presentation Systems

To provide the most effective courtroom presentation, trial team should be supported by trial presentation technologies that are able to generate demonstrative evidence graphics, create imaging solutions, and providing mobility for the team.

DOAR [25] is one of the tools aiming to make connections in court room and can provide trial graphics and support. This system presents a technological philosophy and relies on the integration of video and digital display technologies for combining real time reporting, video conferencing, and computer-based evidence system.

3.2 Memory Forensic Techniques and Tools

In past, forensics analysis mostly focused on information and data obtained from file systems, but in recent years, specially after DFRWS 2005, which introduced forensics analysis of physical memory as its challenge, researches have been done on obtaining as much information as possible from memory. Since acquisition and analysis of physical memory contents need special tools and techniques, there are diverse methods to perform these tasks. Research on memory analysis mostly consider the processes that were running at the time of data collection and try to provide as precise information as possible about the execution history of these processes. Results of these works can be divided into two major categories: *memory acquisition methods* and *memory analysis methods.*

3.2.1 Memory Acquisition Methods

Unfortunately, there are only few hardware-based techniques for gathering physical memory contents that obey the *Locard Exchange Principle* [26] and alter memory contents as little as possible. Christian G. Sarmoria and Steve J. Chapin [27] presented a runtime monitor to log read and write operations in memory-mapped files. The basic concept of this approach is to insert a page fault monitor in the kernel's memory management subsystem. This monitor guarantees the correct ordering of the logs related to memory access events when two or more processes operate on a file in memory. Apparently, such monitoring technique relies on having access to and preparing systems before any incident happens, which is not usually the case for cyber forensics.

Bradley Schatz [28] proposed a method of acquiring the contents of physical memory from various operating systems. His method provides snapshots of the host operating system memory. This method injects an independent, acquisition operating system into the potentially subverted host operating system kernel, snatching full control of the host's hardware. As a proof of concept, the author provided a tool named BodySnatcher, which acquires volatile memory of i386 machines that employ Windows 2000 and above versions as operating systems.

Brian Carrier and Joe Grand in their research [29] explained a hardware-base memory acquisition method that uses a `PCI` expansion card to dump the exact contents of memory to an external device. This approach does not modify memory contents since it does not load any process into the memory. Also, it halts the system

in order to prevent it from changing the memory contents while taking the image. Therefore, it keeps the consistency of the image being taken. Having a hardware proof-of-concept named *"Tribble"*, their approach has limitations such as the need to install the card before the incident happens.

Firewire is a bus technology designed for connections between devices. This technology was introduced by Apple in 1986 and was standardized according to the IEEE 1394 specification in 1995. *Firewire* uses Direct Memory Access (DMA) to improve data transfers. This specification allows clients' devices to directly access a host memory, bypassing the operating system, but it also has its own limitations when it comes to its use for forensics investigation and incident response. Limited availability of firewire ports on computer systems and alteration of the state of the system can be listed as the limitations of firewire. This approach is discussed in a white paper by Antonio Martin [30].

Recently three new tools have been developed with respect to memory imaging methods in Windows platform. The first one is *WinEn* from Guidance Software that ships with *EnCase* Forensic version 6.11 and above. This tool works for both Windows 32bit and 64bit and produces memory images with three different levels of compression. Images generated by this tool are not in a raw format and contain headers which makes them confusing for free, open-source analysis tools. The other downside of using this tool is that it creates a Windows service (and changes registry). Consequently, this registry entry remains in the system and if it is run from local filesystem, the service starts every time the system restarts.

The second tool, *ManTechs Memory DD* (simply called *MDD*), released at no charge under the GPL license, acquires an image of physical memory and dumps it in a raw format file [31]. In order to verify data integrity, the dump file generated by *MDD* is checked by MD5, the common Internet standard used in security applications. The binary file can then be analyzed using external tools. Since this tool generates a *dd-style* image file, the result can be used with other tools that recognize this simple format.

The author of Win32dd, another memory acquisition tool released soon after ManTech's *MDD*, claims that his tool is *completely* open-source [32]. This tool used for capturing memory images under Windows 2003 or Vista, is mainly a kernel mode application that mainly uses native functions and hence, dumps faster than other tools.

3.2.2 Memory Analysis Methods

There has been focus on the forensic analysis of physical memory in recent researches. These researches mostly consider the processes that were running at the time of data collection and try to provide as precise information about those processes as possible.

One of the pioneers in forensic analysis of memory tools is *MemParser* [33], a program that enables its user to load a physical memory dump of certain Windows systems, reconstruct the list of processes, and extract information relating to specific processes. This tool lists all the active processes found in the memory dump files, gives information about each process, and can dump the memory allocated to a

specific process. Although being useful and reliable, this tool does not give detailed information about the memory dump file and does not satisfy the extensive need for forensic memory analysis. The tool does not have a GUI and runs in command line mode, furthermore, it suffers from lack of documentation.

In 2005, George M. Garner Jr. and Robert-Jan Mora presented a tool called *KnTList* [34] as a solution to the DFRWS 2005 challenge [35]. *KnTList* is a commercial command line tool that extracts evidence from physical memory dumps and maintains an audit log and integrity checks. It reconstructs the virtual address space of the system and other processes. After 2005, *KnTList* was improved to generate XML outputs for use by a cross-view detection algorithm and to support systems running Windows 2000 SP4, Windows 2000 Server SP4, Windows XP with SP1 or SP2, Windows 2003 Gold, and Windows 2003 with SP1 or SP2. *KnTList* ships in a package called *KnTTools*, that has a dumper (*KnTDD*) for acquiring the memory contents from suspect computer. The downside of this solution is its dependence to Windows kernel file (`ntoskrnl.exe`) that was running on the system from which the physical memory dump was captured. Also according to DFRWS 2005 challenge solution [36],

> *"It is necessary to place this kernel file in a folder and create a subdirectory called "drivers", which contains the file "tcpip.sys" from the subject system."*

Andreas Schuster in his article entitled "Searching for processes and threads in Microsoft Windows memory dumps" [37] presented a work that is more relevant to the

topic of our research. He mainly analyzed the structures of the kernel and processes inside Windows memory. Instead of following the links provided in these structures, he developed search patterns to scan the whole memory dump for traces of said objects. As demonstrated by a proof-of-concept implementation, this approach could reveal hidden and terminated processes and threads. Under some circumstances, hidden and terminated processes and threads can be revealed even when the system under examination has been already rebooted. As the author states, this approach will encounter serious problems when being used for Windows Vista. `DISPATCHER_HEADER` structure contains some constants, which Schuster uses to identify the objects in memory. Because Windows Vista reuses parts of `DISPATCHER_HEADER` after object creation, this reuse shows that it is not possible to rely on structure fields to identify objects. Also searching in a memory dump for patterns seems to be time-consuming particularly when the RAM volume is relatively large. The tool developed by Andreas Schuster, which is called *PTFinder*, is now able to analyze Windows 2000, Windows XP, Windows XP SP1, Windows XP SP2, and Windows Server 2003. According to the author, *PTFinder* is intended to identify `EPROCESS` and `ETHREAD` structures in Windows memory dumps, but it does not analyze these structures. Consequently, the schema contains only the information needed to locate the structures in a dump file, like file offset, Process ID, and Thread ID. This tool does not extract or reconstruct files.

Jesse Kornblum [38] presented a procedure for extracting executable files corresponding to in-memory processes. He used pointers to recreate modules of executables. Also as a new contribution, he paid attention to Packed Programs. He clarified that due to many factors, such as change of variables at the run time, it is not possible to extract files that are exactly similar to their corresponding ones on the disk.

Brendan Dolan-Gavitt [39] used the Virtual Address Descriptor (VAD) tree to locate and parse structures and provide investigators with useful information about a memory dump. One of the tools developed by Brendan, called vaddump.py, is able to dump pieces of memory image to disk. The tool has been improved a lot after being presented and is now a part of Volatility Framework [40].

Harlan Carvey and Dave Kleiman developed a tool in perl script that retrieves data from the Windows crash dump file, parses structures and handles translations between virtual addresses (and pointers) to physical offsets within the dump file itself. Some of the information that gets pulled for each process includes the FLINK/BLINK values (pointers to previous/next `EPROCESS` block in the doubly-linked list), creation time (exit time, if applicable), whether exit has been called or not, and the location of the Process Environment Block. This tool takes a Windows crash dump file as the input so it needs the system to generate a dump file of the physical memory, which usually happens when the system crashes. It also needs the compromised system to be set up in a way that it generates a full crash dump. The tool comes in their book'd DVD toolkit [41].

Van Baar, Alink, & Van Ballegooij [42] presented three different methods for

recovering mapped files. The implementation of these methods is based on *PTFinder*. The first method is based on walking through the *VAD* tree and locating the shared files and then going through the Object Table. The second method is based on carving memory for structures that are related to files that have no structures pointing to them. In the third method, based on the comparison between hashes of memory pages and hashes of file pages on hard disk, they carved for pieces of files in memory. This method seems not to be useful since it needs access to the hard disk. In addition, when pages of files are mapped into the memory, they are most probably modified. Therefore, their hashes would be different from the hashes of their corresponding pages on disk. The methods used in their research are different from those used in our research and one can use them in parallel to complement results.

There are some general purpose tools that can be used to perform some basic forensic memory analysis as well. *"Strings"* by Mark Russinovich [43] is a tool adapted from Linux to Windows that searches for predefined string values in a given file. *"WinHex"* [44] also is capable of such a search as well as *"Windows Grep"* [45], which does a better and more powerful search using regular expressions. In order to use them, the investigators should know what they are looking for. They also require the investigators to do the analysis in their minds, or use other tools and then just search for proofs.

Finally, in a recent paper, Qian Zhao and Tianjie Cao [46] pointed to the usefulness of collecting sensitive information from memory. They explain that they have been able to extract some sensitive information such as userIDs or passwords through

various means such as using *hiberfil.sys* (the hibernation file that contains a memory dump when the operating system hibernates), Windows crash dump file, pagefile, and direct memory access. Although this work proposed to look for interesting patterns in the memory that may lead to sensitive information, it did not give valuable hints on how to obtain these patterns. This method is covered in Section 5.2.2. One important contribution of our thesis is leveraging this work by explaining the process of obtaining fingerprints and how it can be automated.

3.3 Evaluation

To make readers able to compare the studied tools and techniques and have a detailed understanding of available functionalities, we have prepared some criteria for evaluation and compared the significant existing tools and methods based on these criteria. It should be noted that since the acquisition of memory images is not in the interest of this research, we will only evaluate the analysis tools and techniques. The items of our comparison and their brief explanations are as follows:

- *Processes:* Does the method find processes that were running at the time of imaging?
- *Threads:* Does the method find threads of processes that were running at the time of imaging?
- *File names:* Does the method find files that have been used by the process?

- *File Extraction:* Does the method extract files?
- *Other sensitive information:* Does the method extract other sensitive information such as usernames and password that are inaccessible through use of other methods?
- *Tools:* Does the method have a developed tool?
- *GUI:* Does the tool (if it exists) have a user-friendly GUI?

Table 1 summarizes a comparison of these tools and their capabilities.

Table 1: Comparison Between Current Tools and Their Capabilities

	Tool/ Method	**Processes Info.**	**Threads Info.**	**File Names**	**File Extraction**	**Sensitive Info.**	**Tool**	**GUI**
1	MemParser	✓	✓	✗	✗	✗	✓	✗
2	KnTList	✓	✓	✗	✗	✗	✓	✗
3	PTFinder	✓	✓	✗	✗	✗	✓	✗
4	Volatility	✓	✓	✓	✗	✓	✓	✗
5	Carvey's Toolkit	✓	✓	✗	✗	✗	✓	✗
6	Van Baar et al.	✗	✗	✓	✓	✗	✓	✗
7	Qian Zhao	✗	✗	✗	✗	✓	✗	✗
8	Our Toolkit	✓	✓	✓	✓	✓	✓	✓

Chapter 4

Memory Information Extraction

Physical memory contains lots of pieces of data that are not necessarily relevant to the incident. The first step in forensic memory analysis, after preserving the memory contents, is to extract forensically relevant data. To be analyzable, these data should be in understandable forms such as data structures or files. Only after extraction and presentation of memory resident data, the investigator can identify incident-relevant information and perform further analysis. In this chapter, we focus on extraction of data from memory and its presentation in form of meaningful data structures or files. The first section describes our method for extraction of *Windows Kernel Objects* from memory. The extracted structures, contain information about active processes, execution threads, loaded DLLs, accessed files, etc. These structures, in addition to providing information about execution history of the suspect machine, are used as building blocks for file extraction that is one of our main contributions and is discussed in next sections of this chapter.

4.1 Process Information Extraction

A program is a set of instructions that when executed sequentially, make an executing instance of that program that is called a *process*. Processes, when being executed, load lots of data into the physical memory that enables them to function. Memory management module of the operating system also stores some information about each process to be able to manage its execution. This information includes process name, program location on the secondary memory, external modules used by the process (such as DLL files), execution time, and many more.

As described in Section 2.2, Windows keeps information about memory modules in various structures and maintains links between these structures so that it manages the memory. Extraction of this information and analyzing it gives the investigator the ability to make strong assumptions about the execution history of the compromised machine. To achieve this, we have studied and reverse-engineered Windows memory structures and used them as building blocks for further analysis and data extraction. The result of this study has been integrated into our forensics memory analysis plug-in [1]. Recalling from Section 2.2, one of the most important structures maintained by Windows operating system is the `EPROCESS` (stands for Executive Process) block. All the processes that execute on a system have their corresponding `EPROCESS` blocks. These blocks point to each other to maintain a (doubly) linked list of processes, starting by the block corresponding to the *System* process. Hence, we choose the

[1] This part of the research was done by Alireza Arasteh, another member of our research group, and is available in his thesis [47]

System EPROCESS block as the starting point for extracting other blocks. EPROCESS fields point to other structures that contain important information such as the PCB block (keeps kernel-related information) and ObjectTable block (keeps handles to objects owned by the process). By extracting each important structure, we have been able to present rich information about each process including:

- Process information:
  - process name
  - create and exit time
  - unique process ID
  - section base address (virtual address of the beginning of the process image)
  - exit status
- DLL files that has been used by this process
- Environment variables associated with the process
- Threads that have been created and executed by the process
- Objects owned by the process

Forensics analyst, having this information at hand, can perform a thorough investigation on processes that have been executing at the time of taking the memory image, their relations, and objects that they have accessed. This gives a far better understanding of the incident situation, other possible involved computers, and the attack (if any) methods.

In order to access and extract parts of files, we need to identify those processes that were running at the time of imaging and for this we firstly identify the root process (usually the *system* process) then follow the EPROCESS block's doubly-linked list. To find the information related to all the active processes that are included in the linked list. However, those processes that were hidden using the Direct Kernel Object Manipulation (DKOM) method [48] will be missed. Following the *DKOM* method, *rootkits* can hide a process by manipulating the linked list fields and removing the subject process from the list. Hence, changing the method of finding EPROCESS blocks (to overcome the problem of DKOM process hiding method) can be a future work.

Each time we find a new EPROCESS block, we construct a data structure corresponding to the EPROCESS definition and we fill the values of the attributes in the EPROCESS data structure by reading the image file from the offset corresponding to the beginning of the EPROCESS block.

While filling the values of each attribute in the EPROCESS data structure, we have to fill some other data structures that the EPROCESS block points to, such as SECTION_OBJECT, HANDLE_TABLE, and other memory structures. We repeat this procedure to generate all the nested present structures and assign values to their attributes.

4.2 How File Extraction Helps

Extracting files from one memory image can help investigators to find non-tampered information about the execution of an application and the data files that have been

accessed such as text documents, HTML files, web browser history files, and many more files that can be analyzed and correlated with other sources of digital evidence.

Extraction of files from memory is important from the forensics point of view because of many reasons such as:

- *Possible extraction of deleted files:* A file might have been accessed on a machine and after that, deleted from disk. This file, being a PDF document or other types of data files, may contain important information, and so can lead the investigation team to an important decision. Retrieving files that have been deleted from disk is not always a successful task and can even be useless in some situations. However, assume that we can extract the contents of a file from a memory image and we find out that this file does not exist on disk anymore. This may give the investigator a clue to search for this file on other storage media such as USB keys or optical media. If not found on any other media, any memory-extracted piece of a deleted file could possibly contain valuable information.

- *Extraction of files that have been accessed from a source other than local hard drives:* A Malware or a data file such as an image or a video, may be accessed from any source, including CDs, USB keys, network, etc. In such a case, at least some parts of it may still be present and could be retrieved from physical memory. Checking the contents of the extracted executable can give the investigator an assumption about the type of Malware or its intended functionality.

- *Extraction of tampered files:* In many cases the intruder or in general a suspect, after accessing the computer system involved in an incident, tampers those files that have been modified during the incident. For example, a log file generated by a network application, may record established connections, connection destinations and even the connecting user. Since many of these log files are easily accessible to users, the suspect may tamper the file and edit or remove some information. In such a case, extraction of the tampered file (even partially) will reveal the original contents of the file on disk. Since the edited part of a file in more likely present in physical memory than its other parts, in case of partial extraction, the investigator still has the chance to access important and relevant information.

4.3 File Extraction

Among all the information that can be acquired from exploring the physical memory, file extraction is helpful since files can contain information relevant to the machine owner, machine user, or other machines/individuals involved in the incident. In this research, focusing on Windows XP (SP1 and SP2) operating system, we propose various methods to find and gain access to memory regions that contain files parts. Parts of files, when extracted out of physical memory image, can be subjected to more analysis and examination such as comparison with original files on disk to find possible tampering, or correlation analysis with other sources of evidence. As stated above,

data files such as text documents, images, or browser history files may contain names, dates, or other information that help the investigator find clues about suspects.

In this part of the research, we have divided files into two major categories based on the method used to extract them from memory image:

- *Executable files:* The executable source of the active processes (in `.exe` format).
- *Data files:* Non-executable files that were accessed and used by active processes.

4.3.1 Extracting Data Files

In this section, we present two different approaches to extracting data files. Data files can be of various types such as log, text, PDF, image, or even DLL files. While analyzing physical memory, we can access different part of data files by following two different paths. In some cases, following one path is not possible because of some broken links between memory data structures (this situation may arise when the memory image is not complete or some memory regions have been overwritten). The two approaches presented in this section can be used in parallel to extract as many pages of memory resident data files as possible.

Extracting Memory-Mapped Files

As discussed before (in Chapter 2), whenever a Windows file object is created, for example when a process accesses a file through I/O, a `FILE_OBJECT` object is created and assigned to that file. This way, an active process might have several `FILE_OBJECT`

assigned to it. In Windows, each object represents an entity that is created during the operating system operation.

In Windows, each object represents an entity that is created during the operating system operation. We store these objects in instances of subclasses of a general class (`WIN_STRUCTURE`) defined in our implementation. This class provides a general representation of a Windows object. Also, each Windows object has a header of type `OBJECT_HEADER`. Figure 3 demonstrates a simplified class diagram for the classed representing main memory data structures in our developed memory plug-in. It shows how classes such as `EPROCESS`, `KTHREAD`, or `SharedCacheMap` inherit from `Win_STRUCTURE` class.

To sum up, we can say that each `EPROCESS` block points to a list of `WINDOWS_OBJECT` structures through one of its attributes (`ObjectTable`), which is of type `HANDLE_TABLE`. Each entry of this table points to a `WIN_OBJECT`. Thus, by following these links, we can find all the *Windows Objects* that are associated with this process. While filling the attribute values for attributes of each `WIN_OBJECT`, we check the name of the object type. If it is equal to *"File"* then it means that we have a `FILE_OBJECT`.

Now that we know the relation between `EPROCESS` and `FILE_OBJECT`, it is time to find pieces of the file represented by this object.

In the `FILE_OBJECT`, there is a field named `SectionObjectPointer`, which points to the `SECTION_OBJECT_POINTERS` structure created by Windows for this file. By taking a look at the structure of `SECTION_OBJECT_POINTERS`, we notice three pointer fields:

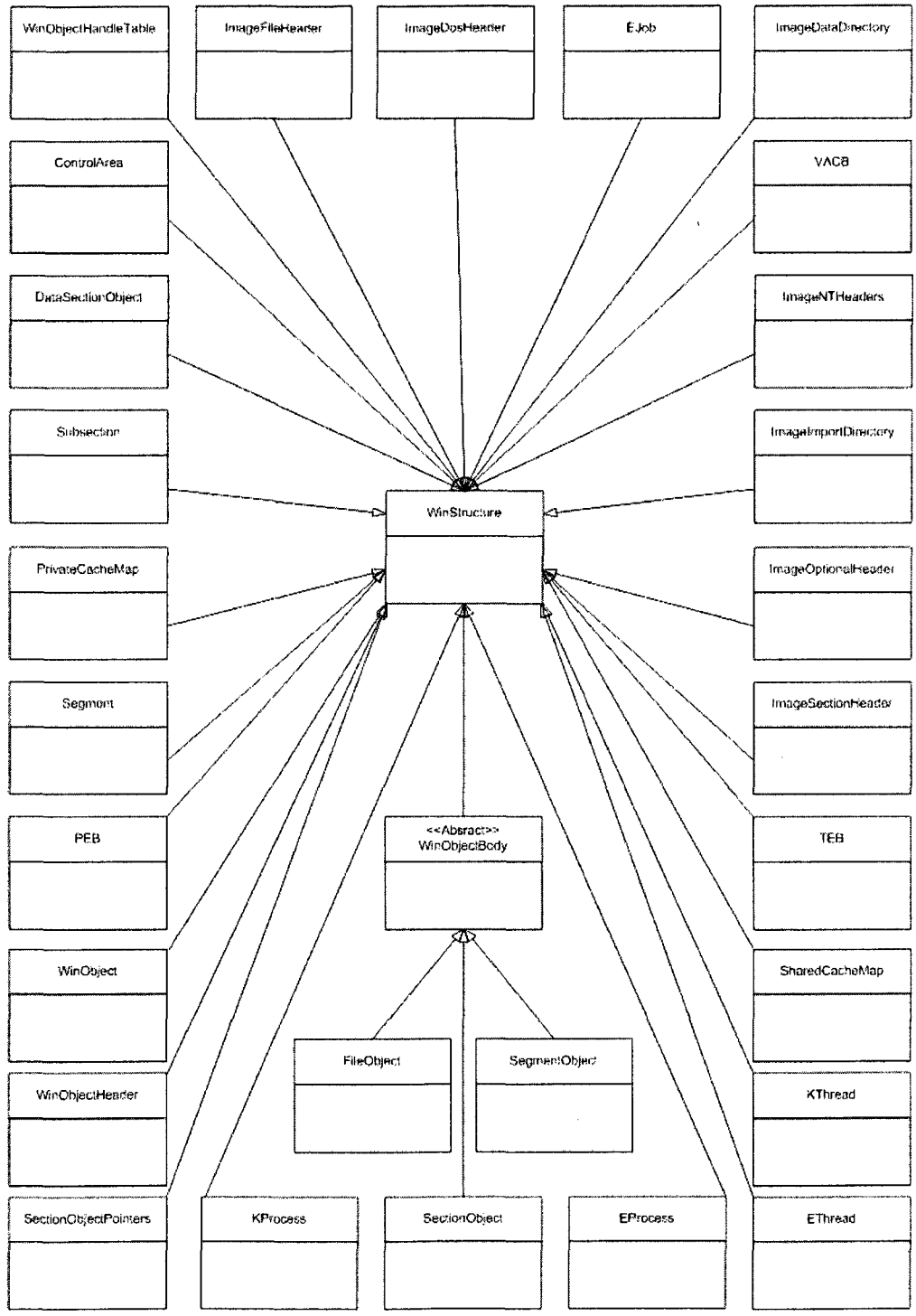

Figure 3: Data Structures Class Diagram

1. DataSectionObject

2. ImageSectionObject

3. SharedCacheMap

Both DataSectionObject and ImageSectionObject point to structures of type CONTROL_AREA. However, there is a difference between their uses:
DataSectionObject structure is used when the file is accessed as a data file and ImageSectionObject is used when the file is accessed as an executable mapped to the memory. CONTROL_AREA keeps information about the mapping of the file in the memory. We follow the pointer named Segment and reach an object of type SEGMENT_OBJECT.
Below is the structure of a SEGMENT_OBJECT:

The important field of this structure is Subsection, which has information about the physical address at which each section of the file is mapped to. The Subsection structure has a pointer to the SubsectionBase (the address of the beginning of the subsection's data in memory) and also a pointer to the next subsection.

Now we can extract a file by first finding its subsections and then copying the content from the page frames that are described by prototype page table entries that are pointed by the subsection.

Figure 4 shows the path that we followed to grab sections of the file in the memory image.

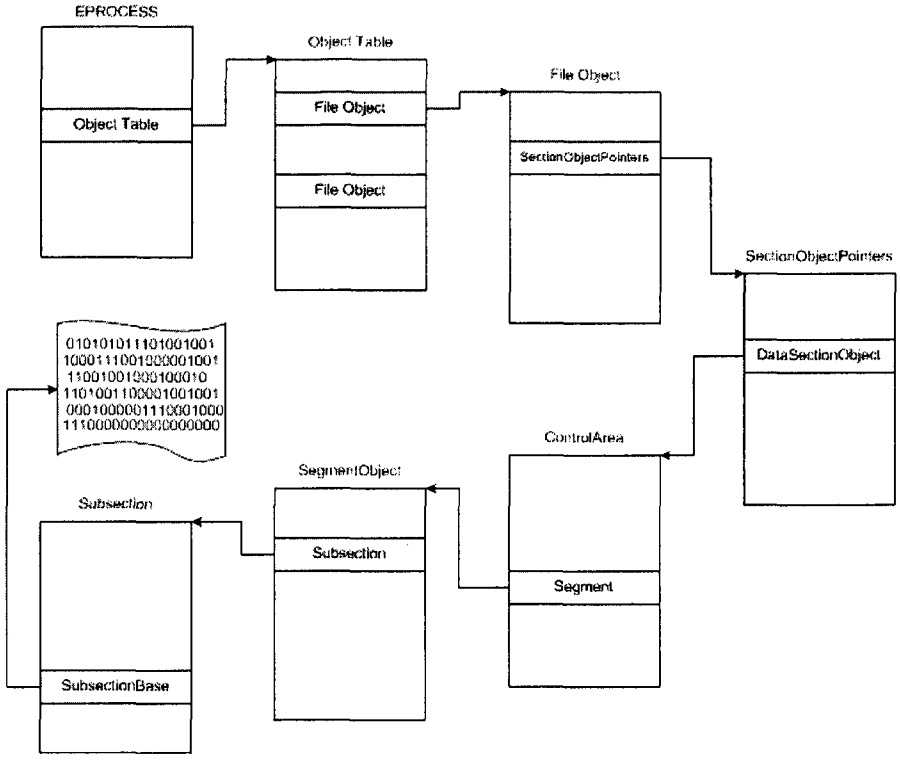

Figure 4: From EPROCESS to Data Files

Extracting Cached File Views

Another way to access pieces of data files in memory is through investigation of cache manager and `SharedCacheMap` data structure. The Windows cache manager uses section objects to map views of files into system virtual address space. Also, in liaison with memory manager, cache manager provides intelligent read-ahead and fast I/O operations. Although not all parts of a file are mapped into the cache memory, those parts of files that are in the cache, are of significance because if a part (view) of a file is found in the cache it means that this view has been in use and accessed by a process recently or frequently.

System cache, which is a part of system's memory, is divided into 256KB views.

Information about these views, including their addresses, are kept in internal structures called Virtual Address Control Blocks (VACB). `SharedCacheMap`, an internal structure that is accessible from `FileObject` (a structure designated to each file in the memory), keeps a pointer to the beginning of the `VACBs` array. This way, by finding all the `FileObjects` associated with an `EPROCESS` block in the same way as described earlier in this section and accessing its `SharedCacheMap` (through `SectionObjectPointers`), we can reach `VACBs` of the cached regions of the file. Finally, by translating the `BaseAddress` value found in `VACB` structure, to the physical address, we can copy the contents of this view of the file.

It is important to know that if a `BaseAddress` value is zero, the `VACB` is actually inactive and may not be valid anymore. Pseudo code in Appendix A provides the detailed method.

Figure 5 shows how a PDF file that has been accessed by *Adobe Reader* application is extracted and is viewed by the PDF viewer utility, which is incorporated into our plug-in.

4.3.2 Extracting Executable Files

Executable files are very important to the investigation process. An executable file will be loaded into the memory when it is running. It contains code, text fragments, resources, debug information, and other data. In order to find pieces of an executable file that is associated with a specific process, we can follow the same approach as for the data files. But when reaching the `SectionObjectPointers`, instead of following

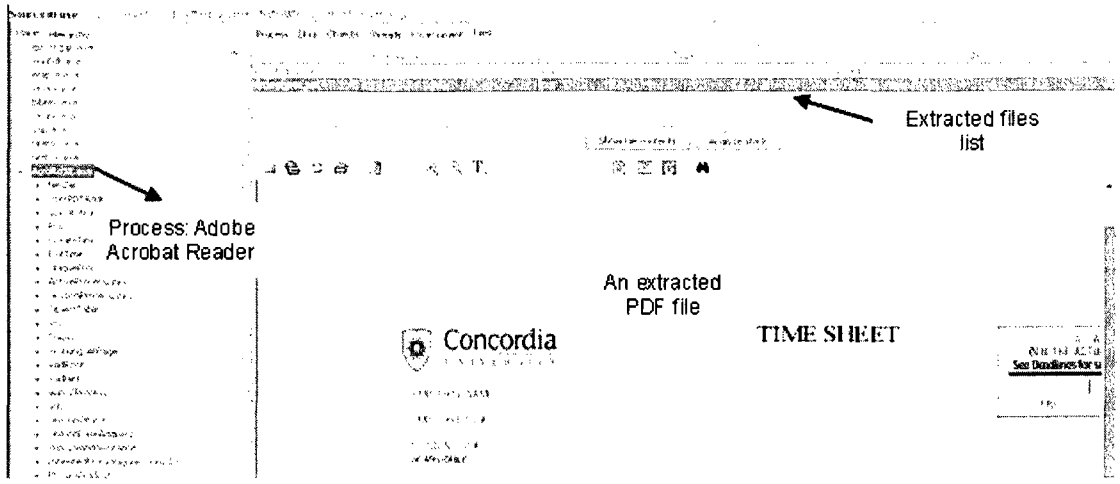

Figure 5: Developed Toolkit - An Extracted Data File

the `DataSectionObject`, we will follow the `ImageSectionObject`. Remember that the `DataSectionObject` is used for data files while `ImageSectionObject` is used for executable images.

There is another approach for extracting executable files out of the physical memory image that we are going to describe:

Remembering the `EPROCESS` structure, we notice a pointer to the Process Environment Block (PEB) assigned to each `EPROCESS`. The `ImageBaseAddress` field of the `PEB` contains the address at which the memory-mapped executable starts. We will read a stream of data from this address. So, the extraction of the rest of the file begins.

To be able to understand the procedure of executable extraction, it is important at this point to clarify the format and layout of an executable file in Windows. This format is called PE (Portable Executable) format.

PE File Format:

Portable Executable (PE) file format history goes back to the earlier Common Object File Format (COFF) that was used on VAX/VMS. However, this specific format has been introduced by Microsoft as a part of the *Win32* specifications. The term *"portable"* shows the intent of this format: Being usable on all Windows versions, regardless of the CPU architecture.

A good aspect of the PE format is its layout in memory: Data structures of a PE file on disk are almost the same as those mapped into the memory. This helps us be sure that what we find in memory is going to be almost identical to what has been on the disk. This would turn to a key point: If you know how to find something in memory (inside a PE image) you will get the same thing that you expect. PE files are not mapped into the memory like other *mapped-file*s, but the operating system decides what it wants to bring into the memory. Furthermore, it is guaranteed that higher addresses of a PE file will be mapped into higher offsets in the memory. A PE file consists of the following main parts:

- DOS Header (`IMAGE_DOS_Header`)
- PE Signature (`IMAGE_NT_HEADERS`)
- File Header (`IMAGE_FILE_HEADER`)
- Optional File Header (`IMAGE_OPTIONAL_HEADER`)
- Section Table
- Sections

Every PE file begins with a small MS-DOS executable, containing the familiar `MZ` mark that introduces a `DOS` executable for us. The only important value in *DOS Header* is `e_lfanew`. It contains the offset of the PE header. The optional header can have a variable size and the file header contains the size of the optional header. The offset of the file header from the image base is stored in the dos header. Therefore in order to find the offset of the section table, we have to read both the dos header and the file header.

Following the `IMAGE_NT_HEADERS` is the section table. The section table is an array of `IMAGE_SECTION_HEADER`s structures. An `IMAGE_SECTION_HEADER` provides information about its associated section, including location, length, and characteristics. Finally we will end up with sections. There can be different sections in a PE file, with names that usually starts with "." such as *.data*, *.text*, or *.rsrc*.

PE files contain data or code in different usage modes like: read or write program data (such as global variables). Besides this, there are other types of data in sections such as API import and export tables, resources, and relocations. When Windows loader loads a PE section into the memory, sections always start at the beginning of a page. That is, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page (on x-86 CUPs a page is 4KB).
Figure 6 shows different parts of a PE file and the way they are mapped into the physical memory.

Now that we know the PE file format, we can extract and reconstruct executable

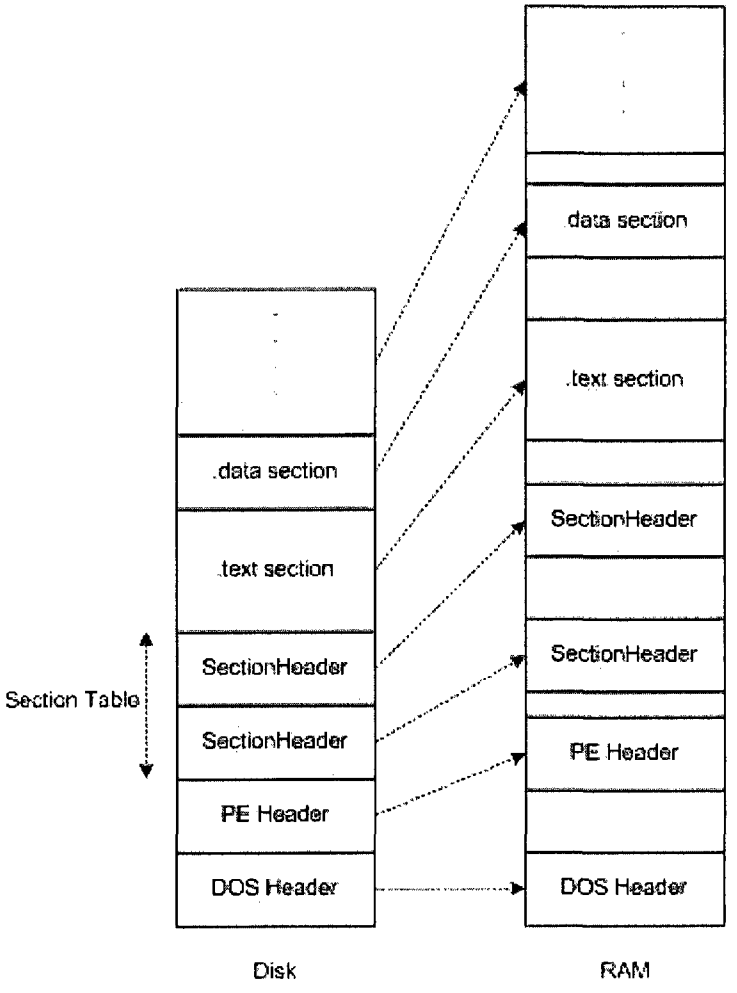

Figure 6: PE Files Layouts in Disk and Memory

files in this format by following the links between parts of the PE file in the memory, extracting them, and putting them together. In order to achieve this goal, we need the first link of the chain that is the `ImageBaseAddress` field of each process. The `ImageBaseAddress` field contains the starting address of the process image in memory.

The Process Environment Block (PEB) of a process, which is pointed to from the `EPROCESS` structure, has an attribute named `ImageBaseAddress`. This attribute contains the starting address of the process image in memory. Now we have the starting

address at which the executable file is mapped. According to the PE file format described earlier, in order to find the *section table*, which contains `ImageSectionHeaders`, we need to go through the headers of the PE file. First we will copy the contents of the memory from virtual address: `ImageBaseAddress` and with the size of an `ImageDosHeader`. Inside this structure, the `e_lfanew` field points to the PE signature (`ImageNTheaders` structure).

Now we have the address of `ImageNTheaders` and will copy the contents of the memory from this address, with the size of an `ImageNTHeaders` structure. The `ImageNTHeaders` structure is also pointing to structures of type `ImageFileHeader` and `ImageOptionalHeader`. We resume by copying memory bytes for these two structures according to their addresses, found in `ImageNTHeaders`. The important point to be noticed is that the size of `ImageOptionalHeader` varies from process to process. This value can be found as a field named `SizeOfOptionalHeader` in the `ImageFileHeader`.

Until now, we have found, copied and written all the headers of the PE file into a new file that we will call it **extracted file**. What we should do in order to get the main parts of an executable file, is to find and go trough section headers that contain information about different sections of the file. `ImageFileHeader` has an attribute called `NumberOfSections`. This value shows how many sections are present in this executable file. Before starting to write actual sections of the PE file, we should copy and write `ImageSectionHeaders`. This structure has a specific size and we will repeat reading and writing according to the number of sections.

Inside the `ImageSectionHeader` structure, there are attributes that help us find the following:

- *Name*: Name of the section that is intended to show the use of the section, for example *.rdata* stands for *read-only data*. Although this value is usually by convention the same for similar sections in different PE files, some compilers such as visual C++ allow programmers to assign customized names to sections at compile time of the PE file [49].

- *VirtualAddress*: This is the offset starting point of the section when loaded into the memory relative to the starting address of the file. This address is where we should start copying memory contents for this section. By adding this virtual address to the `ImageBaseAddress` of the mapped file, we obtain the virtual address at which the section is mapped to the memory.

- *SizeOfRawData*: This is the size of the section in the PE file on the disk.

- *VirtualSize*: This value shows the size of the section when loaded into the memory.

By finding the virtual address of a section and its size, it is possible to read and copy the contents of memory starting from virtual address with the length equal to the size. It is important to mention that `SizeOfRawData` could be equal to, less than or greater than the `VirtualSize` depending on the alignment requirements. However, if it is less than the `VirtualSize`, the remainder of the section will be filled out with zeros and is not of forensic importance.

Once a PE file is loaded into the memory, its sections are not mapped continuously. This means that each page of the section is mapped to a different virtual address. Thus, we have to find the virtual address of each page, translate it to the physical address and copy the page contents into the extracted file. The pseudo code of the algorithm we used for extraction of executable files is presented in Appendix B.

Figure 7 presents a screen shot of our toolkit that demonstrate the success of our method in finding and extracting an executable file (eclipse.exe). We were also able to disassemble this executable (in the "File Contents" panel). The executable file has been disassembled using the "disasm" tool [50] and the result code is shown in the "File Contents" panel of the GUI.

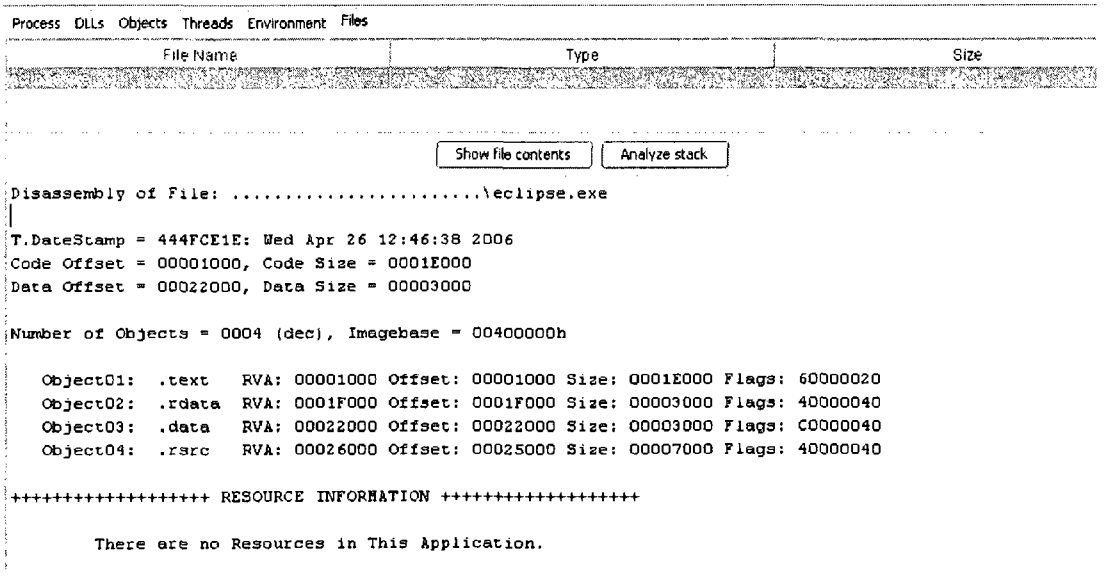

Figure 7: Developed Toolkit - An Extracted Executable File

4.4 Conclusion

In this chapter, we introduced our method for finding and extracting *Windows Kernel Objects* from a physical memory image. Also, we introduced forensically important attributes of these objects that are presented to the investigator in our developed toolkit. Knowing values of these attributes such as process start and end time, number of threads of each process, or loaded DLL files, can shed light on the execution history of the suspect machine.

Furthermore, we presented our method for extraction of executable (EXE) and other data file such as PDF, DLL, and TXT files. The extraction of these files can help the investigation in many ways such as rerunning the executables of the processes that have been active at the time of incident and studying their behavior. Extraction of data files, can reveal images or documents that have not been found on other storage media or have been tampered during the incident.

The research threads explained in this chapter, being fruitful by presenting new methods for extraction of files as a digital evidence, demonstrate valuable contributions to the digital forensics community.

Chapter 5

Sensitive Information Investigation

Most of the research carried out in the digital forensics science focuses on enumerating processes and threads by accessing memory resident objects, while ignoring the value of pieces of sensitive information that can be collected from inside memory contents. Given this, the present chapter summarizes results of our research on extraction of forensically important data from physical memory of machines running under Windows XP (SP1, SP2) operating system (the issue of Windows versions is discussed in Section 1.1). This chapter demonstrates two main approaches: 1) Analysis of fingerprints that applications and protocols leave in memory and 2) Analysis of call stacks.

5.1 Introduction

Since physical memory may contain numerous evidence that even might not be found in any other source, digital forensics community feels the urge to rapidly develop tools and techniques. This way, investigators will be able to capture and analyze the memory in order to facilitate the investigation and to come to more reliable conclusions.

This chapter explains the importance of the information that exists in memory for forensic investigators and introduces new approaches for the extraction of this information. We can describe the new approaches that constitute our contribution as: a) Presenting the *Stack Function Call Analysis* method, b) Systematically finding fingerprints of applications and protocols and using them to extract sensitive data, and c) Presenting some of the most common patterns for fingerprint analysis.

This chapter, using the facts about memory layout and management in Windows that have been explained in Chapter 2, explains how memory may hold sensitive information. In this chapter, we present the categorization of sensitive information in memory, its possible correlation, as well as existing and new methods to extract it.

When performing the investigation and analysis, finding possible correlation between gathered evidence can add to the credibility of conclusions or even refute an assumption. Thus, detailed and precise information obtained from memory can be very important when analyzing the correlation between evidence. For instance, a deleted record in a log file can be acquired from memory as a part of an extracted

log file. Likewise, metadata acquired from process information can be used to reject the validity of a manipulated log file. On the other hand, assume that a suspect has entered a username and password in a Secure Socket Layer (SSL) enabled web page. This information can be extracted from memory and afterwards can be used for analysis of other collected evidence.

To find and extract sensitive information from memory we offer two main approaches that can be used. Forthcoming sections provide details of these methods.

5.2 Application/Protocol Fingerprint Analysis

This section introduces our first approach to extraction of sensitive information: fingerprint analysis. The first subsection explains the old-fashion string search method, which became the origin of our first proposed method, described in the second subsection.

5.2.1 Search for Pre-Known Strings

In many cases, when investigators are looking for data related to a specific subject such as a person's name, address, or friends, they know what they are looking for in memory. For example finding a specific name of address in memory contents, can be accomplished by searching for *ASCII* or *UNICODE* strings in the memory dump using applications such as *WinHex* [44] that facilitates string and binary search in a binary file. Figure 8 shows how an investigator may end up a `Yahoo` account username

when searching for a pre-known last name. The main advantages of this method are easiness and availability of tools and in some cases, accuracy and relativeness of results. There are many freeware or commercial string/binary search tools that can do the job. Although we have incorporated this method in our developed toolkit, it cannot give us the best results. In many cases, there are other names, addresses, user IDs, and other strings present in the memory dump that we are not aware of them while they are of importance.

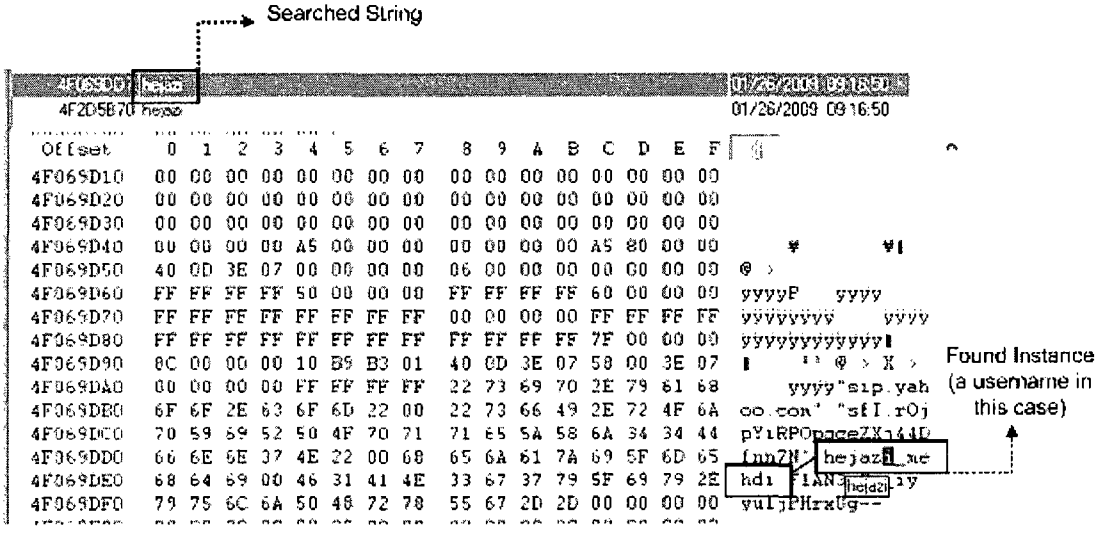

Figure 8: Sensitive Information Found While Searching for Strings in Memory Contents

Existence of unknown sensitive data in the memory is one of the limitations of this method.

5.2.2 Search for Fingerprints

Applications are compiled from source code written in various programming languages. Recalling the concept of functions (a.k.a subroutines) in programming, while processing sensitive input data, they use constants that at some point of time are brought to memory. In this approach, we call those constant variables that precede or succeed sensitive information *"fingerprint"*. Fingerprints, being a string constant or a series of non-string bytes may have constant distances with the sensitive piece of information in memory. In order to shed some light on this concept, consider the piece of code below:

```
if (encrypted)
{
  dlg.SetPasswordMode(true);
  if (dlg.ShowModal() != wxID_OK)
    return false;
  if (!Send(_T("password " +
                dlg.GetValue())))
    return false;
  if (GetReply(reply) != success)
    return false;
}
```

In line 6 of the code, which is a part of *FileZilla* [51] project code (can be found in optionspage_connection_sftp.cpp, filezilla 3.2.0) , shows that the string "password " concatenated with another string value (seems to be a password received from a dialog box) is passed to a function as a parameter. When this value is about to be processed, we can be sure that it exists somewhere in the memory. Assuming that the user has entered the string *"my_secret"* in the dialog box, the string *"password my_secret"* can be found in the memory. Now if another user has entered *"trickword"* in the dialog box, we can expect to find the string *"password trickword"* in the memory. Obviously, the string *''password "* can be considered as a prefix to actual values of passwords presented by this piece of code.

In order to help investigators, we have identified a set of common applications that deal with sensitive information and classified them in several categories. By examining the source code or the compiled code of the application (in case the application is not open source or we do not have access to the source code) we looked for the portions of the code that deal with sensitive information. We also used disassembler applications such as IDA Pro [52] and PE.Explorer [53] in order to analyze the compiled code in case we did not have access to source code. When traced, we examined the code for possible existing strings (or sequence of bytes) that precede or succeed sensitive information. Since there are a limited number of commonly used applications in these categories, we could build a set of these *fingerprints*. This set contains fingerprints used in applications such as FTP clients, SSH/Telnet clients, messengers, and web browsers. Table 2 demonstrates some fingerprints present in our set of patterns.

5.3 Call Stack Analysis

In this section, we will explain our method for investigating memory stack in order to find and extract forensically sensitive information that is present in physical memory. By recalling call stack and stack frames from Section 2.5, which are building blocks of this work, we present our approach.

5.3.1 Function Parameter Extraction

The concept of parameter passing using the call stack is the beginning of our interest in stack for forensic analysis of physical memory. When a user interacts with a computer, she/he inputs data and gets some output. These inputs can be usernames, passwords, web site URLs, inputs to fillable forms and many more. However, what happens when the user enters data as inputs? This information will be considered as parameters for some functions of the processes running on the machine and will be placed in physical memory. Therefore, in the course of execution, a specific process stores needed parameters and information in the memory and keeps pointers to memory locations. Functions inside a program process its inputs and many times pass variables between each other in the form of function arguments. Some of the inputs passed to functions in a program are just indexes, or local variables, which are not of forensic interest, but others can indicate names, dates, e-mail addresses, web site addresses, username or passwords. Knowing that this information exist in memory and can be extracted after the incident, makes investigators hopeful. One possible way to

extract sensitive information from memory is string search. This method is based on searching and extracting *UNICODE* or *ASCII* strings from a file (that can be a memory image). Recalling from Section 3.2.2, tools such as "String" or "Windows Grep" can be used to do this search. The disadvantage of theses approaches is that investigators should know what they are looking for and then start to search for it. The other downside of this method is that when the investigators find a string, they do not have enough information about the program that was using this keyword and the way it was used.

In the following subsections, we propose a technique that uses information present on call stack of each process and the executable image of that process to track sensitive function calls and find arguments passed to these functions.

5.3.2 Parameter Extraction: Methodology

In order to extract functions' parameters, we will analyze the stack of a process (or more precisely, a stack of an execution thread) to find out which functions have been called. If the function is an interesting one (that is it contains sensitive data as inputs), we will try to locate and extract the arguments passed to it. These arguments are usually stored in process space and can be accessed using their virtual addresses, which are in turn stored on the stack. For this purpose, we have to observe the signature of designated functions and based on the number and type of arguments they take, look for them (pointers to arguments or the arguments themselves) on the stack.

Windows is based on a layered architecture that prevents user applications from accessing sensitive system components. In this architecture, applications transfer the execution to DLL files in order to be able to communicate with executive services in kernel mode and finally to hardware. There are numerous programs that deal with forensically important and sensitive data such as web browsers, FTP clients, and many more. On the other hand, there are a limited number of DLL files that handle requests of these programs. Therefore, as Figure 9 explains, it is reasonable that instead of digging into all applications, we find those DLL files that handle sensitive requests and look for calls to their functions on the stack.

When an executable or a DLL calls a function in another DLL, a call instruction in the program will be executed. Suppose an FTP client application wants to establish a connection to an FTP server. This program takes the username, password, the address of the FTP server, and other sensitive information as inputs. Then, it stores this information in different locations in process address space and pass them as arguments to functions inside application's code. According to the layered architecture of Windows, application's functions at the end, call some functions in different DLL files and arguments are passed to these newly called functions as well. Therefore, if the function `FTPConnect(user, pass, uri)` in the FTP client application calls the function `connect(s: TSocket; var name: TSockAddr; namelen: Integer)` in `WSOCK32.DLL`, then the structure of the stack for that thread of execution would look like Figure 10.

This way, we can use DLL files as bottlenecks and look for important function

Table 2: Some Fingerprints and Their Corresponding Applications

	Application	Fingerprint
1	Yahoo Web Mail	*passwd*
2	Yahoo Web Mail	*login*
3	Horde Web Mail	*imapuser=*
4	Horde Web Mail	*pass=*
5	WinSCP	*password 00 00 00 08 **
6	Yahoo Messenger	*buddies=(**

* A string followed by hexadecimal values

** List of friends present in Yahoo messenger

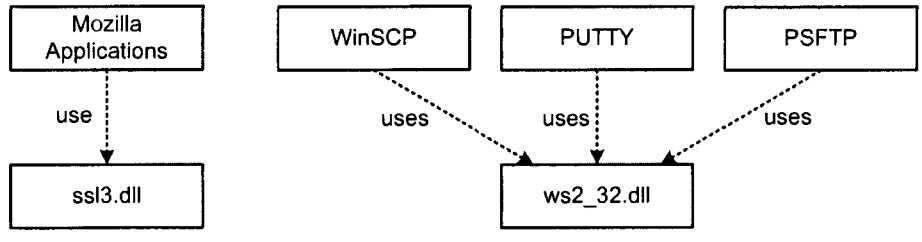

Figure 9: Many Applications Use Common DLLs

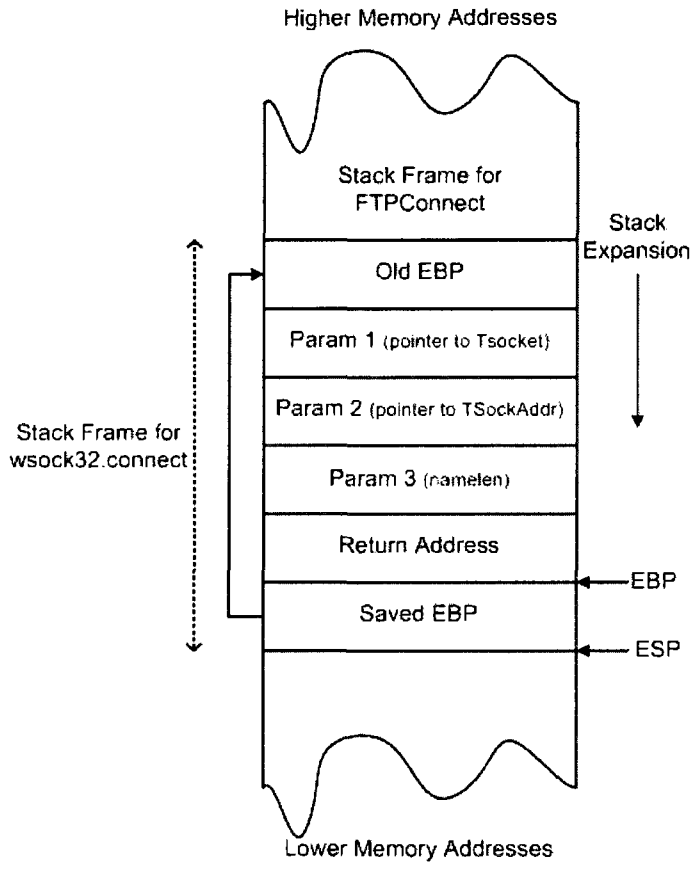

Figure 10: Stack Structure During Function Calls

calls in specific DLLs that can be found on the stack. We can explain this approach in several steps as follows:

1. Locate the stack memory associated with each thread of processes.

2. Distinguish stack frames for each function call on the stack.

3. Understand and describe the function that has been called.

4. Reconstruct the Import Address Table (IAT) of the process image.

5. Compare each called function with the list of forensically sensitive functions.

6. Extract the parameters that exist on the stack, if the function found on the stack corresponds to one the interesting functions in our list.

The rest of this section is dedicated to explaining each of the above steps.

- *Locating the stack for each thread:* As described in Section 2.4, Windows uses internal data structures in order to manage memory operations and objects in the memory. For the purpose of finding stacks of a process, we need to get information about each of its threads. In order to reach threads and their stacks we start with the important `EPROCESS` block, which is the starting point of our access to process information. The `EPROCESS` block, contains a `KPROCESS` structure, also known as `PCB` (Process Control Block), a kernel structure that contains information about scheduling of process threads. The `KPROCESS` block, under the name of `ThreadListHead` maintains the starting address of an array (`LIST_ENTRY`) at the offset of `0x050`. Each entry of this array, in turn, keeps the starting address of `KTHREAD` structures, each of which represents a thread of execution for the current process. `KTHREAD`, is the kernel representation of a thread that contains information about thread scheduling. What is interesting for our method in this structure, is the following information about the thread stack:

  - *StackLimit* at offset `0x01c`,

  - *KernelStack* at offset `0x028` and

– *StackBase* at offset `0x168`

This information provide the maximum size of the stack, current value of the *stack pointer*, and the starting address of the stack expansion. Knowing these values and by translating them to physical addresses, we can reach the area in the memory image that corresponds to each stack. In this approach, we would not limit the area of searching to the space between the *StackBase* and the *Stack Pointer*, because there might be some inactive stack frames above the address that *Stack Pointer* points to (stack residues). These stack frames, represent previous function calls (those functions that have returned). Therefore we will do our experiments for the whole memory space between the *StackBase* and the *StackLimit*. Figure 11 demonstrates stacks of different processes' threads inside the process address space. After locating each stack, we start to parse its contents.

- *Distinguish the stack frame for each function call:* In order to investigate the stack frame of each called function, we have to identify the boundaries of each stack frame. As stated above in this section, when a function is called, the return address for the called function is saved on the stack. This return address is actually the address of the instruction that comes right after the *call* instruction. In other words, if the address of the *call* instruction is n, and the length of the *call* instruction is l, then the return address will be: $retAddress = n + l$. Based on this fact, we will read each 4 bytes on the stack and assume that what is

read, is a return address (*retAddress*). If this assumption is true, then at l bytes before this address (inside the code segment) we should find a *call* instruction, where l is the length of a *call* instruction. Hence, if what we find at address $retAddress - l$ is a *call* instruction, then our first assumption will be true, and what we had read from the stack would be a return address.

Figure 12 demonstrates how stack frames can be associated with function calls inside the code and how they can be distinguished.

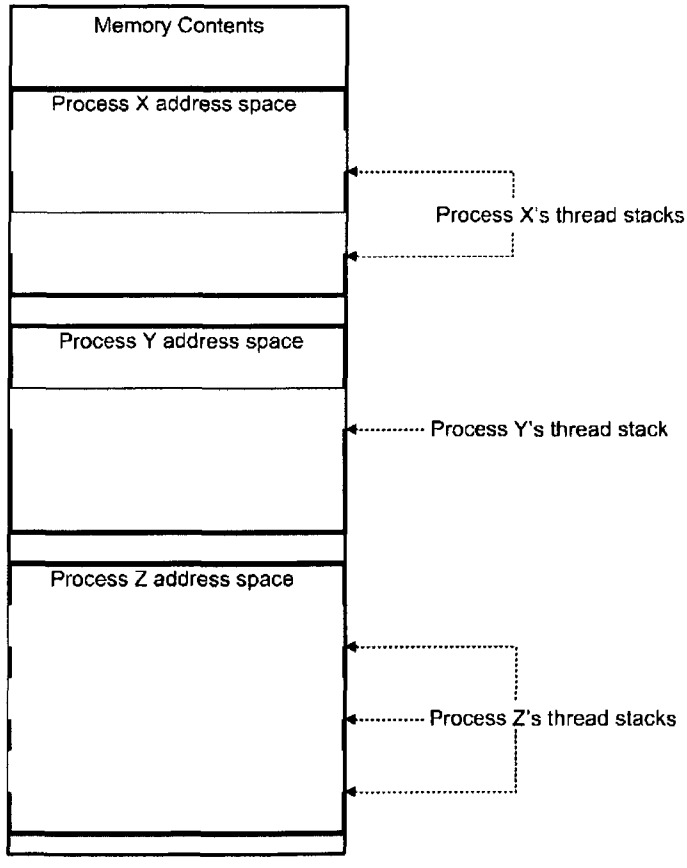

Figure 11: Finding Stack for Each Execution Thread in Process Address Space

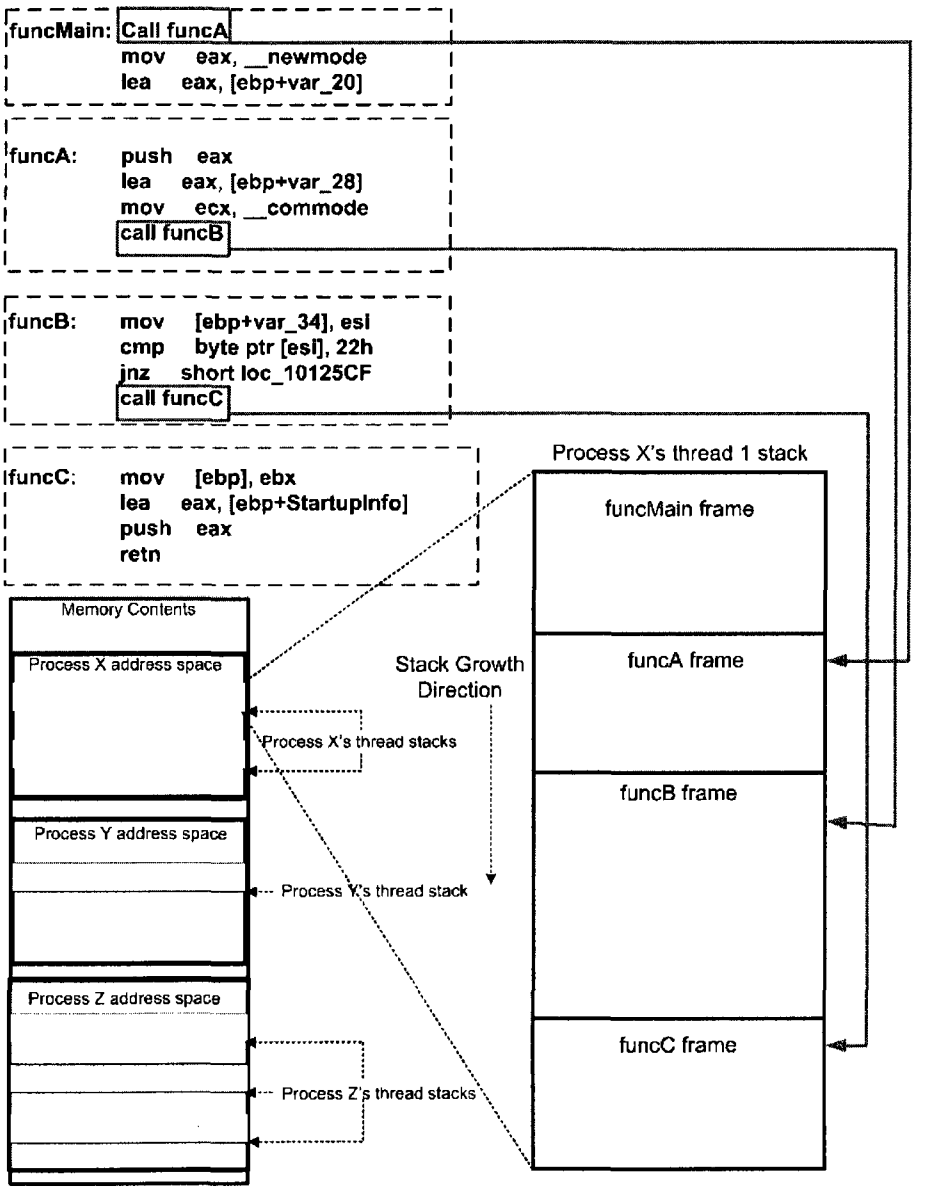

Figure 12: Continuing from Figure 11, Finding Stack Frames

- *Find out which function is the called function:* Since we are trying to find forensically important function calls, we will focus on functions imported from common *Application Programming Interfaces* (API) rather than functions that are specific to an application. For instance, while analyzing the stack of Internet Explorer process, we prefer to look for functions imported and called from Secure Socket Layer (SSL)/Transport Layer Security (TLS) APIs instead of those functions inside Microsoft Internet Explorer that directly implement SSL API. Each imported function (functions that are imported from DLL files and can be used by the process) is mapped to a physical address in the memory. Recalling from previous step, in order to find return addresses, we reached a *call* instruction in the code of the executable. By looking deeper at a call instruction we will see that this instruction is always followed by the address of the called module. This address can be in different modes of addressing depending on the type of the call instruction. This address can be an immediate value, a general-purpose register, or a memory location. Since *Near Call* instructions are calls to procedures within the current code segment, and we are looking for calls to imported procedure (that are definitely not in the current segment) we will only look for *Far Call* instructions. By finding the address of the called procedure (immediate value or memory location) we will locate the function that corresponds to the current stack frame.

- *Reconstruct the Import Address Table (IAT) of the process image* On the other hand, while processing information about each process, we will make a list of

imported functions for each imported DLL file that the process declares. This information can be obtained from Import Address Table (IAT) that is simply a lookup table used when the application calls a Windows API function. IAT stores memory locations of the corresponding library functions [15]. This way, by traversing this table, we can find all the Windows API functions that are imported by this process and can be possibly used during the course of execution of the process.

Figure 13 shows the name of imported DLL file `ADVAPI32.dll` and names of the functions in this DLL file that are imported by a process (ftp.exe) inside a memory image.

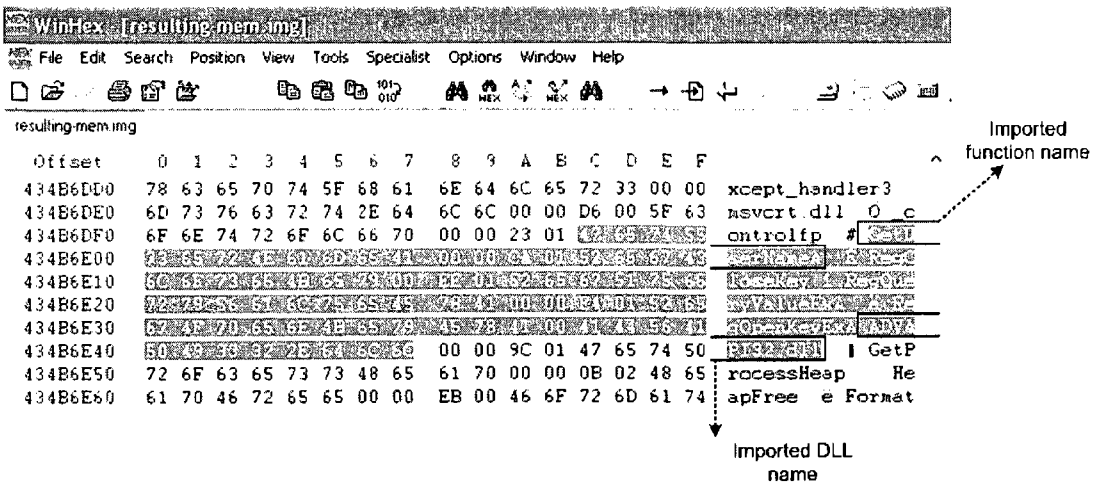

Figure 13: An Imported DLL File and Name of It's Functions in the Memory Image

Now that we have all this information, we can search in the addresses of imported functions for all the function addresses that we found on the stack and

identify which function has been called.

- *Comparison between called and sensitive functions:* As explained earlier in this section, not all of the called functions carry sensitive information. For example, `WSACleanup` function imported from `WS2_32.dll` (Windows Socket 2 API) terminates use of the `WS2_32.dll` [54], a functionality that does not provide us with direct forensically important information. On the contrary, `getaddrinfo` function, from the same DLL file, provides protocol-independent translation from an ANSI host name to an address, which can give us sensitive information such as a host name.

  Thus, in order to filter called functions based on their sensitivity, we have to prepare a list of API functions that are more important from investigator's eyes. To achieve this, we studied common APIs that may process forensically sensitive information such as:

  - OpenSLL, SSL/TLS APIs
  - Network Security Services (NSS) [55]
  - Microsoft Networking and Windows Security APIs (including Windows socket API)
  - Microsoft CryptoAPI (Cryptography application programming interface)

  Table 3 represents examples of functions that we have been looking for during our experiments. We have also traced some functions such as `WriteFile` that writes data to the specified file or input/output (I/O) device and is exported

from `KERNEL32.dll`. Although `KERNEL32.dll` is not a part of security or networking APIs and is widely used by different applications, `WriteFile` is used to write sensitive information to a socket (a specific type of I/O device).

- *Parameter extraction:* Since sensitive information are passed to different functions as parameters, we are interested in extracting those parameters for each found sensitive function. Except very small-size parameters, most of the parameters (including strings) are passed to functions as pointers to memory locations where the actual values of the parameters reside. These pointers to memory locations are stored on the stack (as a part of the call stack frame). The order of the parameters in the stack frame is the same as the order of the parameters passed to the function. Thus, in order to locate a parameter in the memory, we have to extract the pointer to that parameter from the stack frame and follow that pointer to reach the desired memory location. Now we need to know how many bytes to read and how to interpret read bytes. Should we read 4 bytes and interpret them as an integer number or should we read 16 bytes and interpret them as a 16-character string? In order to solve this issue along with the issue of the order of the parameters, we have to know the signature of the function that we are investigating: The number, type, and order of the parameters. This is what the stack analysis module expects from the user.

Table 3: Example Functions from Different APIs that May Carry Forensically Sensitive Information

	Function	API	Function Description
1	`SSL_CIPHER_description`	*SSL/Ciphers*	Returns a textual description of the cipher used into the buffer *buf* of length *len* provided
2	`SSL_CTX_check_private_key`	*SSL/Protocols*	Verifies that the private key agrees with the corresponding public key in the certificate that is associated with a specific context (CTX) structure
3	`SSL_check_private_key`	*SSL/Connections*	Verifies that the private key agrees with the corresponding public key in the certificate that is associated with the Secure Sockets Layer (SSL) structure
4	`FindFirstUrlCacheEntry`	*WinINet(Windows Internet)*	Begins the enumeration of the Internet cache
5	`FtpCommand`	*WinINet(Windows Internet)*	Sends commands directly to an FTP server
6	`GetAddressByName`	*Windows Sockets 2 (Winsock)*	Queries a namespace, or a set of default namespaces, to retrieve network address information for a specified network service
7	`gethostbyaddr`	*Windows Sockets 2 (Winsock)*	Retrieves the host information corresponding to a network address
8	`send`	*Windows Sockets 2 (Winsock)*	Sends data on a connected socket
9	`BluetoothAuthenticateDevice`	*Wireless Networking/Bluetooth*	Sends an authentication request to a remote Bluetooth device

5.3.3 Case Study

Using the explained method, we have been able to identify and extract sensitive information from a memory image that cannot be accessed using the other analysis method. As an example, we extracted an FTP account username that was used during an FTP session. The FTP client used in this example is Microsoft FTP client, available in machines running Windows operating system. The login credentials used by this program are not cached or saved. However, they are just sent to the server using a connection socket. This is the case where stack analysis method comes to help the investigator. To pinpoint the transmitted sensitive information, we:

1. found process ftp.exe on the list of processes that were running at the time of imaging,
2. enumerated all DLL files imported by this process and all imported functions,
3. located the stack for thread(s) of execution,
4. examined addresses on the stack(s) to find return addresses (to locate the stack frames),
5. checked function calls on the stack against imported functions from DLLs (from step 2),
6. found function `WriteFile` imported from `KERNEL32.dll` (this function writes a buffer of characters to a general file, which is a socket in this case [56]),

7. located addresses of `WriteFile` parameters (according to its signature) and finally,

8. extracted the **username** (second parameter to `WriteFile` function).

Figure 14 displays and summarizes the result of the above steps.

There are two issues that should be mentioned here:

1. All the address values pointed to in the screenshots, are presenting virtual addresses that during the analysis have been translated to physical addresses in order to locate new values in memory image.

2. Due to the fast changes made to stack frames on a stack (because of numerous function calls), we cannot make sure that the stack frame of a certain function call can be found on the stack. This is why in the example above, we have been able to identify an account username, but not the corresponding password.

5.3.4 Limitations

There are some issues that add to the difficulty of this method. In order to explain more and help future extensions to this work, we enumerate some of these limitations:

- Many applications do their internal processing (including encryption password processing, etc) inside the application code and do not directly call the available API. Thus, in many cases, only those low-level and common functions are called from APIs (such as *WriteFile* function, explained in Section 5.3.2). These calls

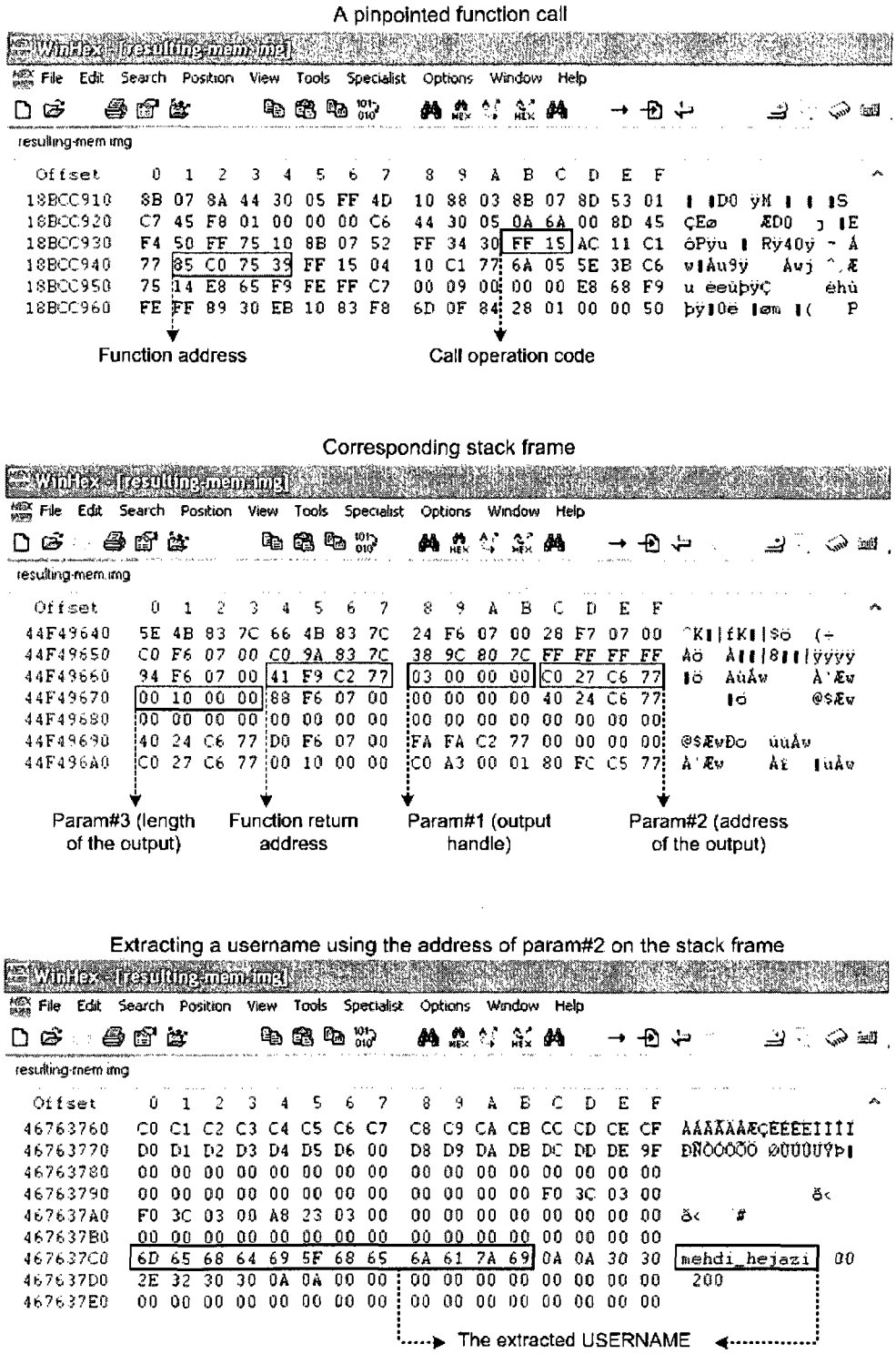

Figure 14: Extracting an Account Username from Memory Using Stack Analysis Method

are repeated many times for various purposes and do not always lead to sensitive information.

- In some applications (such as *PuTTY* [57]) in order to consider security of the users and prevent attacks, sensitive information such as SSH2 client's passwords are completely removed from memory contents right after they are used (by setting memory bytes used to hold the password to zero) [58].

- In cases that we want to analyze closed source applications, to find the flow of sensitive data such as inputs, we need to analyze the assembly code of the program and sometimes use assembly debuggers such as IDA Pro [52]. Consequently, the process of tracing sensitive information path and finding functions that handle these data becomes more cumbersome.

5.4 Conclusion

Due to pertinent and accurate information that can be carved from memory contents, memory analysis has become an important part of forensic analysis. Since evidence found by analyzing physical memory (that in many cases cannot be found in any other sources) can respond to key questions about the incident such as who, when, how, and where, investigators do not want to lose any evidence that may be obtained by memory analysis. In this chapter, we introduced and used new methods for extracting forensically sensitive information such as usernames, passwords, visited URLs, and encryption keys from physical memory. The first method, which is based on locating

fingerprints of applications and/or protocols, uses instances of these fingerprints to identify sensitive information that appear before or after them. However, the second method, tries to find stack frames corresponding calls to specific functions and locate the sensitive parameters passed to these functions. These methods can augment other memory forensics techniques such as finding processes [34, 37] and extracting files and add to their value by correlating results.

Chapter 6

Memory Analysis Plug-in

This chapter provides explanation of our developed toolkit and its capabilities. The sections of this chapter explain how the functionalities and capabilities of our Forensics Investigation Framework and specifically the memory analysis plug-in satisfies the growing demand of the digital forensics community.

6.1 Forensics Investigation Framework

As a part of the research, our team has developed a Forensics Investigation Framework that encapsulates various forensic analysis methods. The developed toolkit is a Java based application that uses JPF (Java Plug-in Framework [59]). It provides an extensible framework through plug-ins environment for forensic investigations and case management. As a parts of this framework, other team members have developed *Network Analysis*, *E-Mail Analysis*, and *Log Analysis* plug-ins in addition to the

Memory Analysis plug-in. We use *Hibernate* and *Object Persistence* [60] in order to keep records of what investigators do in addition to information about the case and involved evidences. This framework allows investigators to add evidence to a case and analyze them.

6.2 Memory Analysis Plug-in

As a part of the forensics analysis framework, we have developed a physical memory analysis plug-in for Windows XP (SP1 and SP2). Our framework provides various capabilities such as listing all the processes found in memory (in both a plain view and in a hierarchial view), their properties and attributes, DLL files used by each process, environment variables associated to each process and kernel threads. The developed plug-in takes a byte-by-byte copy of memory contents (referred to as *image*) and performs file extraction on the image as well.

Once the memory image has been parsed, the investigator can see a list of processes that were running at the time of memory imaging (that can be almost the same as memory status at the time of incident if not many changes are made on the system). By clicking on each process in this list, different tabs containing various information about the process such as DLL files and process properties are shown. In this toolkit, we have provided two different views for processes: *Plain View* and *Hierarchial View.* The plain view gives a plain list of processes while the hierarchial view demonstrates a tree of processes that shows the parent process that created the child process (like

in Process Explorer [61]). One of the process information tabs is "Files" tab on which investigator can see a list of all *extracted files* related to this specific process. In fact, this toolkit generates a folder for each process and names it the same as the process itself and uses a combination of methods stated in Chapter 4 in order to extract executable and data files from memory image. It then generates files with the same name as the file it is extracting from memory and writes what has been extracted to these files. The "Files" tab in the memory plug-in, lists all these files. The "DLLs" tab lists all the DLL files used by the selected process. The "Threads" tab lists all the threads of execution for the selected process and their respective information such as start and end times and thread-Id. As the explanation of the memory plug-in shows, we have developed a self-contained plug-in that encapsulates almost all what an investigator needs to perform analysis on physical memory extracted.

Figure 15 shows the memory analysis plug-in while the investigator wants to select a memory image file for analysis. In the figure you can see that the investigator can select a path in order for the plug-in to dump the extracted files.

6.2.1 Architecture and Technology

One part of the memory analysis is the examination of extracted files, and this may include opening data files or running executables. Since each of these files may contain malicious scripts, this type of analysis can be risky and sometimes harmful to the system on which investigator is working. Thus, it is recommended to use an isolated and protected environment for this purpose. Bem and Huebner [62] suggest use

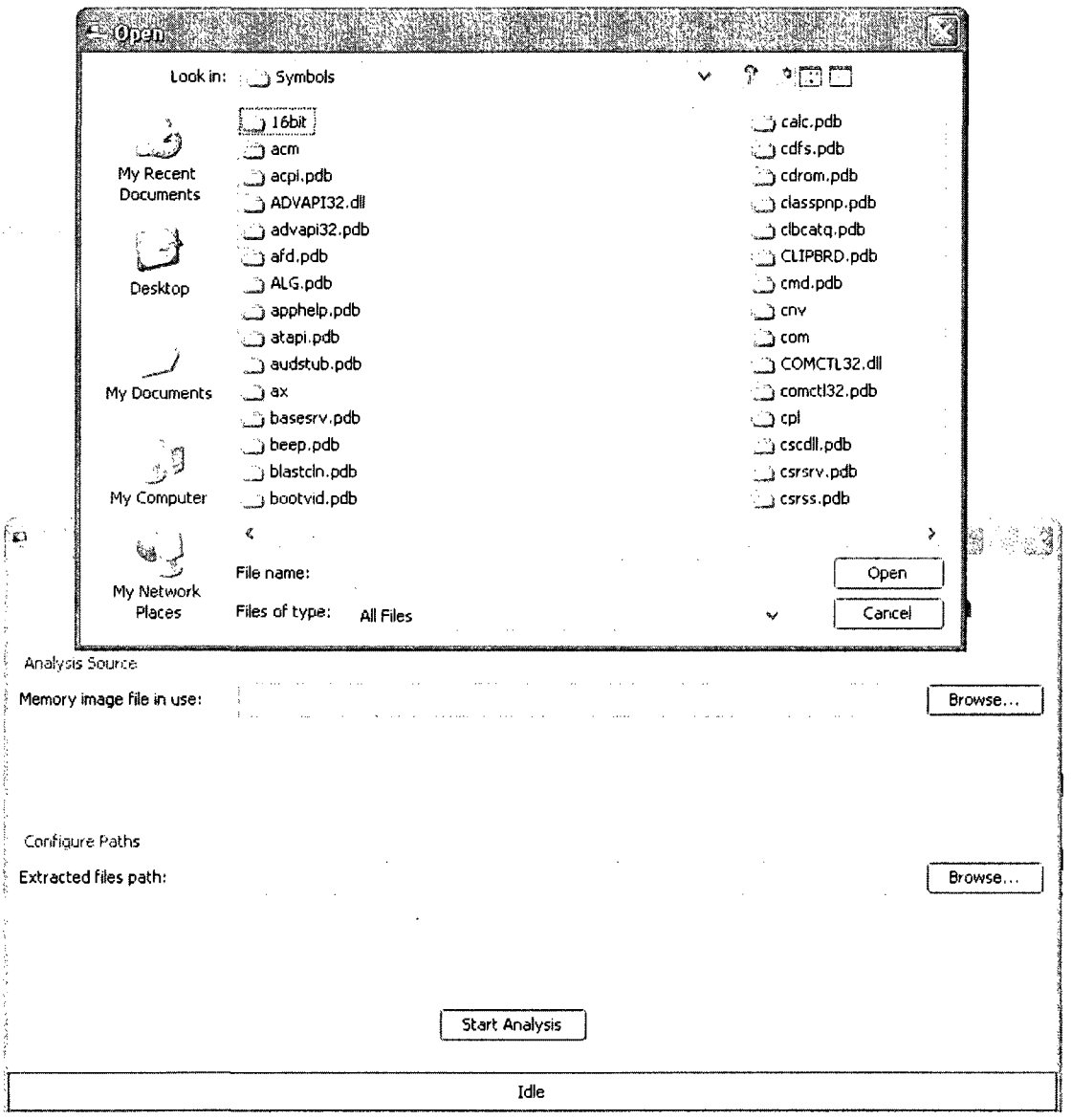

Figure 15: Screenshot of The Developed Framework

of both a conventional analysis environment and a virtual environment concurrently and independently. They have also proved the importance and the efficiency of this approach in their next research that analyzes a USB flash drive in a virtual environment [63]. In order to meet this requirement, we have installed and integrated the framework into a virtual machine. This reduces the actual risks that may arise from memory analysis.

6.2.2 Capabilities and Features

Memory Imaging

Investigation of memory begins with acquisition of the RAM contents. As discussed in Chapter 3, there are different methods of imaging RAM but a method that generates a simple and raw dump file, without extra metadata, still remains the choice for investigators. Since an investigator may find it necessary to analyze the memory image with different tools in order to verify the results, it is better to acquire the image in a general format. The Forensics Physical Memory Analyzer, comes with a built-in RAM dumper that generates a byte-by-byte copy of memory. The user of the tool can select a previously taken image of RAM or to start memory imaging of the present machine and then start to analyze it.

Stack Analysis

In a multitasking operating system, a process initiates one or more execution threads. A fragment of memory space is assigned to each of these threads as it's stack. A

call stack or simply a stack, is a LIFO structure that stores information about the subroutines of the associated thread. Information about each subroutine, is organized in *Stack Frames.* In the most common structure, the stack frame includes local variables of the called function, the return address of the caller function, and the parameter values passed into the called function.

Processes need to store their sensitive inputs (such as IP addresses, usernames, passwords, etc) somewhere in the memory space allocated to them and pass them as parameters to functions. This way, if we know the signature of functions that use these sensitive information, we can search the call stack for them and locate their parameters.

Our memory analysis plug-in is capable of locating the stack region corresponding to each execution thread and then analyzing the stack contents in search of calls to specific sensitive functions. According to the signature of the forensically sensitive function, it then locates the parameters passed to that function and lists these information for investigator's view. Thus, we can provide investigators with significant information about the latest function called during the course of execution of a specified process and more important, the sensitive parameters used by these functions.

File Extraction

Data Files: Having different files extracted from one memory image can help investigators to find non-tampered information about the execution of an application and data files that have been accessed such as text documents, HTML files, web browser

history files and many more files that can be analyzed and correlated with other sources of digital evidence.

The memory analysis plug-in, is capable of listing and extracting data files accessed by active processes. It dumps the extracted files in directories named according the processes. The toolkit also provides a set of modules for common data file types to help investigators read, view or search within the extracted files. Most common modules integrated into our plug-in include:

Text viewer: Using text viewer, our framework can open the text-based files inside the plug-in and view its contents in a text format. Investigators can also benefit a *text search* feature that helps examining the contents of a text file in search of a specific keyword.

PDF viewer: Using *Adobe PDF viewer Java plug-in*, investigators are now able to view PDF files inside our framework. This complete PDF utility is very similar to *Adobe Reader* with most common PDF features, which helps users save time while conducting an investigation. Using such viewer utilities, investigator does not have to open the file with other external and third party applications. This may save their time specially when the framework is used inside a virtual machine (that may not have the required applications installed).

HTML viewer: Our memory framework is able to view extracted files that are in HTML format. Investigators are also able to view the HTML file in a text format, which means the source code for HTML files.

Image viewer: Using our framework, the investigator can view common image files such as JPG, GIF and PNG. Supporting more image formats can be a future extension to this work.

Executable Files: Knowing the format of a *Portable Executable* (PE) file and using the methods explained in Chapter 4, the memory analysis plug-in is able to extract the process images (executable files) corresponding processes that were executing at the time of imaging. Once a PE file is loaded into the memory, its sections are not mapped continuously, that is each page of the section is mapped at a different virtual address. Thus, we have to find the virtual address of each page, translate it to the physical address and copy page content into the extracted file.

We have provided a number of analysis features for extracted executable files including:

Code Disassembling: Using disasm.exe [50], a popular disassembler tool, memory analysis plug-in lets investigator to view the disassembled code of the extracted executable that she/he chooses. This feature is available inside the memory analysis plug-in and the code can be viewed right inside our framework.

Assembly Code Investigation: Using *IDA-Pro* [52] plug-in development features, our framework is capable of lunching *IDA Pro* in order to investigate and explore the assembly code of the extracted executable file. This feature helps investigators to gain more information about the details of the file and perform complex analysis on found evidence. Executable analysis include listing the

functions of the executable, names and strings, debugging the executable and so on.

Executable Run: We have added a functionality to the plug-in that allows the investigator to run an executable that is completely extracted. In many cases, we have been able to extract all parts of an executable. Therefore, we have been able to run it and see what it is able to perform. It is important to mention that running executables or opening and accessing extracted contents from a memory image can be insecure and risky.

File Similarity Check

Memory extracted files (executables or data files) are digital evidence that can be analyzed and serve as a link in the chain of investigation. In order to escalate the admissibility of these evidence, we have to be able to prove their validity. On the other hand, in most of the cases as a result of process execution and its changes to the files, extracted files are not exactly the same as original files on disk. But how similar are they? If we can show the similarity of an extracted file with the original file by a percentage, this could be helpful in deciding whether to accept it as an evidence or not. It is important to mention that when containing important information, pieces of extracted file, even being very small portions of the actual file on disk, still can be useful and carry forensically important information. In our integrated toolkit, we provide investigators with the capability to use *fuzzy hashing* to compare and evaluate the similarity of extracted files with the original files they provide.

String Search in Memory Image

String search in the memory image is a very common need of investigators. Assume that an investigator knows a suspicious e-mail address and wants to verify if this e-mail address was used by any applications at the time of accident. If so, he may want to examine the contents of memory image with more details. As another example, suppose that the investigator wants to know if a specific file (e.g, contract.doc) was accessed by any process or not. In this case, if the file is not among the memory extracted files, he may use string search. The toolkit will display a range of bytes before and after the searched string (if found) in both *binary* and *ASCII* formats and highlights the searched string and its offset in the image file. If more than one hits, the toolkit will list all of search hits.

Editing Data Files

The ultimate goal of extracting files (especially data files) out of memory is to investigate their contents, either in search of a specific piece of information or in search of all important information. By choosing "Files" tab after a process has been selected, all the extracted files associated with the selected process, will be displayed and user can see their contents. Therefore, it is necessary to assist investigators with string search, replace, copy, paste, and other routine editing functionalities when they open and examine data files. As a real example, our plug-in is able to find and extract *history* files associated with a web browser. This could be of high value especially if the history file has been deleted from disk before investigators reach the compromised

machine. This history file, having a volume of at least 1 MB, contains information about the web sites visited by users. So, being able to search for a web site URL in this file using our toolkit and saving time, completely makes sense.

Virtual Address Translation

For systems with virtual memory, a *Virtual Address* (VA) is a memory location that the intervening hardware and/or software maps to physical memory. As an application runs, the same virtual address may be mapped to many different physical addresses as data and programs are paged out and paged in to new memory locations [64].

Many times, it comes to a situation in which an investigator analyzing physical memory contents, needs to find the physical address of a certain virtual address linked to a certain process. This can help investigator to find the requested memory location and investigate its contents. Our memory analysis plug-in helps investigators to translate a virtual address to its corresponding physical address. Since in 32-bit versions of Windows, the procedure used to translate virtual addresses depends on the mode in which the operating system operates (normal mode or Physical Address Extension mode [14, 11]), the Address Translator module allows users to specify the mode of address translation.

Sensitive Information Extraction

An image of a physical memory may contain numerous pieces of information ranging from MSN network usernames and passwords to "form history" of web browsers containing strings that were inserted in forms of web pages. An investigator may not be aware of everything that exists in a memory dump, but if we can search for common, forensically important information, namely sensitive information, then we can provide the investigator with this information, and of course categorize them in order to make it easier for the user to find useful data. Thus, we have studied commonly used applications under Windows platform to find out their fingerprints in memory. Applications that establish network connections such as FTP, SSL, Telnet, and SSH connections were among the applications that leave sensitive information in memory. After analyzing memory, we categorize information found and show them (if any) as:

- *Connection information (FTP and SSH, SLL/TLS):* This includes usernames, passwords, commands, encryption information and connection destinations.
- *E-mail information:* (MS. Outlook Express and Mozilla Thunderbird)
- *Web browser information* (Firefox, Netscape, and Internet Explorer)

Figure 16 illustrates the architecture of the developed framework and its modules.

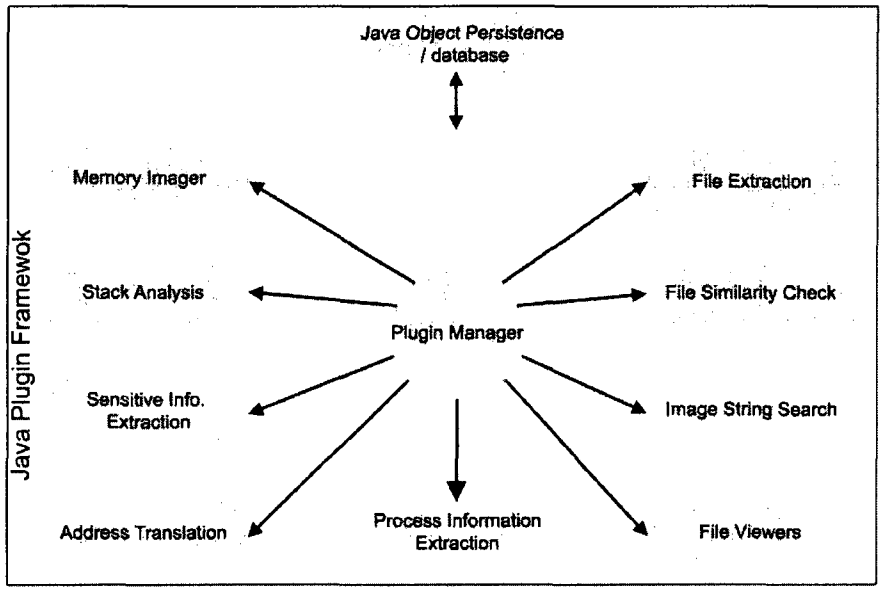

Figure 16: Framework Architecture

Chapter 7

Conclusion

In this thesis, we have focused on acquisition and analysis of physical memory as a reliable source of evidence during a digital forensic investigation. After thorough study of the state-of-the-art in the field of digital forensics and memory forensics, we ascertained that physical memory hosts a great amount of data that cannot be found on any other source of evidence.

Although its size is usually far less and its contents are much more volatile compared to secondary media such as disks, physical memory holds very accurate and pertinent information that can turn to reliable evidence for the court of law. Starting by studying previous works and examining existing free and commercial tools, we identified important areas of this field and started to build our own tools and techniques for the purpose of forensic investigations. In addition to parsing memory resident structures and constructing building blocks for more complex analysis, we have presented two different methods for analyzing physical memory that constitute

our main contributions in this thesis.

The first contribution is file extraction from memory images. Using this method, the investigator can identify, extract, and reconstruct various types of files that are present in a memory image and have been accessed by applications and processes. The extracted files being in HTML, PDF, DOC, TXT, LOG, and even PE (portable executable) format (including EXE and DLL formats), can reveal important information about the machine being investigated. These files might have been deleted from disks after an attack. Thus, extracting them is a very helpful step in an investigative case. Reconstructing an executable file allows the investigator to rerun the file and study its behavior and purpose. This can be referred to as event/scene reconstruction, which is an important phase of digital investigation.

The second contribution is extraction of forensically sensitive information from memory contents. Knowing that in order to be processed, all the sensitive information will appear in physical memory at some point of time, we focused on proposing new methods for pinpointing sensitive information available in memory such as usernames, passwords, URLs, email addresses, and such items. We achieved this objective by using two main approaches: (1) finding application or protocol specific patterns that precede or follow sensitive information and (2) analyzing stack frames for each thread of execution of processes. In the second approach, locating calls to sensitive functions helps the investigator extract sensitive parameters such as passwords.

We also developed a plug-in based *Digital Forensics Framework* for which different teams developed different forensic analysis plug-ins such as Network Analysis plug-in,

Log Analysis plug-in, and Email Analysis plug-in. Our contribution in this framework was the development of the *Memory Analysis plug-in.* The Memory Analysis plug-in is capable of acquiring memory images, parsing images for kernel data structures, finding processes, threads, DLL files, environment variables, and objects associated with each process as well as extracting data/executable file. Moreover, the developed plug-in analyzes the stack of execution threads to build a partial execution path (according to function calls chain) and to extract forensically sensitive information. The developed framework, unlike many other command line analysis tools, is benefiting a rich, user-friendly GUI in addition to its extensibility through plug-in development.

Bibliography

[1] G. Palmer. A Road Map for Digital Forensic Research. Document authored from the collective work of all DFRWS attendees. Technical report, Digital Forensic Research Workshop (DFRWS), August 2001. Retrieved on March 20, 2009, from http://www.dfrws.org/2001/dfrws-rm-final.pdf.

[2] DigitalGuards Terminology Database. Retrieved on May 1, 2008 from http://www.digitalguards.com/glossary.htm.

[3] R. Shirey. Request for Comments: 2828 - Internet Security Glossary, May 2000. Retrieved on May 2, 2008, from http://www.ietf.org/rfc/rfc2828.txt.

[4] End Stalking In America Definitions. Retrieved on May 1, 2008, from http://www.esia.net/Definitions.htm.

[5] B. Carrier and E. Spafford. Defining Event Reconstruction of Digital Crime Scenes. *The Journal of Forensic Sciences*, 49(6), November 2004. Retrieved on May 5, 2008, from http://www.astm.org/JOURNALS/FORENSIC/PAGES/4772.htm.

[6] Debugging Tools for Windows. Microsoft Corporation. Retrieved on March 20, 2009, from http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[7] Sysinternals Process Utilities. Microsoft Corporation. Retrieved on January 8, 2009, from http://technet.microsoft.com/en-us/sysinternals/default.aspx.

[8] Windows Driver Kit: Kernel-Mode Driver Architecture: EPROCESS. Microsoft Developer Network. Retrieved on March 22, 2009, from http://msdn.microsoft.com/en-us/library/bb742898.aspx.

[9] B. Sanderson. RAM, Virtual Memory, Pagefile and All That Stuff. *Microsoft Knowledge Base*, December 2004. Retrieved on March 1, 2009, from http://support.microsoft.com/kb/555223.

[10] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals.* Microsoft Corporation, 4 edition, 2003. Retrieved on March 20, 2009.

[11] Operating Systems and PAE Support. Windows Hardware Developer Central. Retrieved on January 8, 2009, from http://www.microsoft.com/whdc/system/platform/server/PAE/pae_os.mspx.

[12] B. Sanderson. Memory Support and Windows Operating Systems. *Windows Hardware Developer Central*, February 2005. Retrieved on March 1,

2009, from http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx.

[13] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1, July 2008. Retrieved on January 8, 2009, from http://www.intel.com/products/processor/manuals/.

[14] Physical Address Extension - PAE Memory and Windows, July 2006. Windows Hardware Developer Central. Retrieved on January 8, 2009, from http://www.microsoft.com/whdc/system/platform/server/PAE/PAEdrv.mspx.

[15] Microsoft Portable Executable and Common Object File Format Specification, Revision 8.1. Microsoft Corporation, February 2008. Retrieved on March 1, 2009, from http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[16] P. Deitel and H. Deitel. *C++ How to Program.* Prentice Hall Press, Upper Saddle River, NJ, USA, 6 edition, 2007.

[17] Honeynet Challenge of The Month - Scan 24. Recover and Analyze Captured Evidence from a Floppy Disk. Retrieved on March 20, 2009, from http://old.honeynet.org/scans/scan24.

[18] N. L. Beebe and J. G. Clark. Digital Forensic Text String Searching: Improving Information Retrieval Effectiveness by Thematically Clustering Search Results.

Digital Investigation, 4(Supplement 1):49–54, September 2007. Retrieved on January 10, 2009, from http://www.dfrws.org/2007/proceedings/p49-beebe.pdf.

[19] F. Cohen. Challenges to Digital Forensic Evidence. CyberCrime Summit 06, February 2006. Retrieved on January 10, 2009, from http://all.net/Talks/CyberCrimeSummit06.pdf.

[20] DIBS Computer Forensic Equipment, Advanced Forensic Equipment. Retrieved on March 22, 2009, from http://www.dibsusa.com/products/products.asp.

[21] DRAC Forensic Computer Systems. A product of LC Technology International. Retrieved on March 22, 2009, from http://www.lc-tech.com/Hardware/hardware.html.

[22] Brian Carrier. The Sleuthkit and The Autopsy Forensic Browser Digital Forensic Tools. Retrieved on January 15, 2009, from http://www.sleuthkit.org.

[23] Forensics ToolKit 2.0. Access Data. Retrieved on February 5, 2009, from http://www.accessdata.com/forensictoolkit.html.

[24] EnCase Enterprise. Guidance Software. Retrieved on March 17, 2009, from http://www.guidancesoftware.com/products/ee_index.asp.

[25] DOAR Trial Presentation. DOAR Litigation Consulting. Retrieved on February 5, 2009, from http://www.doar.com/litigation-consulting-services/trial-presentation/index.asp.

[26] J.W. Chisum and B. Turvey. Evidence Dynamics: Locard's Exchange Principle & Crime Reconstruction. *Journal of Behavioral Profiling*, 1(1), January 2000. Retrieved on March 25, 2009, from `http://www.profiling.org/journal/vol1_no1/jbp_ed_january2000_1-1.html`.

[27] C. G. Sarmoria and S. J. Chapin. Monitoring Access to Shared Memory-Mapped Files. *Digital Forensic Research Workshop (DFRWS)*, 2005. Retrieved on March 10, 2009, from `http://www.dfrws.org/2005/proceedings/sarmoria_memorymap.pdf`.

[28] B. Schatz. BodySnatcher: Towards Reliable Volatile Memory Acquisition by Software. *Digital Forensic Research Workshop (DFRWS)*, 2007. Retrieved on March 11, 2009, from `http://www.dfrws.org/2007/proceedings/p126-schatz.pdf`.

[29] B. Carrier and J. Grand. A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *The International Journal of Digital Forensics & Incident Response*, 1(1):50–60, February 2004. Retrieved on March 20, 2009, from `http://www.digital-evidence.org/papers/tribble-preprint.pdf`.

[30] A. Martin. FireWire Memory Dump of a Windows XP Computer: A Forensic Approach, 2007. Retrieved on February 11, 2009, from `http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf`.

[31] Memory DD. ManTech International Corporation. Retrieved on January 10, 2009, from http://www.mantech.com/msma/MDD.asp.

[32] M. Suiche. Capture Memory under Win2k3 or Vista with *win32dd*. Retrieved on January 10, 2009, from http://www.msuiche.net/2008/06/14/capture-memory-under-win2k3-or-vista-with-win32dd/.

[33] C. Betz. Memparser Analysis Tool by Chris Betz. DFRWS Forensics Challenge, 2005. Retrieved on January 10, 2009, from http://www.dfrws.org/2005/challenge/memparser.shtml).

[34] G. M. Garner Jr. and R. Mora. KnTTools with KnTList, 2007. Retrieved on March 18, 2009, from http://gmgsystemsinc.com/knttools/).

[35] Digital Forensic Research Workshop (DFRWS). DFRWS Forensics Challenge, 2005. Retrieved on March 20, 2009, from http://www.dfrws.org/2005/challenge).

[36] KnTList Analysis Tool. DFRWS 2005 Forensics Challenge, 2005. Retrieved on March 20, 2009, from http://www.dfrws.org/2005/challenge/kntlist.shtml.

[37] A. Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Forensic Research Workshop (DFRWS)*, 2006. Retrieved on March 1, 2009, from http://www.dfrws.org/2006/proceedings/2-Schuster.pdf.

[38] J. Kornblum. Recovering Executables with Windows Memory Analysis. A Presentation for ManTech International. Retrieved on March 1, 2009, from http://jessekornblum.com/research/presentations/dod-cybercrime-2007-recovering-executables.pdf.

[39] B. Dolan-Gavitt. The VAD Tree: A Process-Eye View of Physical Memory. *International Journal of Digital Forensics and Incident Response*, 4(1):62–64, September 2007. Retrieved on February 5, 2009, from http://www.dfrws.org/2007/proceedings/p62-dolan-gavitt.pdf.

[40] The Volatility Framework. Volatile Systems, LLC, 2006-2008. Retrieved on March 12, 2009, from https://www.volatilesystems.com/default/volatility.

[41] Harlan Carvey and Dave Kleiman. *Windows Forensic Analysis*. Syngress Publishing, 1 edition, July 2007.

[42] W. Alink R.B. Van Baar and A.R. Van Ballegooij. Forensic Memory Analysis: Files Mapped in Memory. *Journal of Digital Investigation*, 5(1):S52–S57, September 2008. Retrieved on March 18, 2009, from http://www.dfrws.org/2008/proceedings/p52-vanBaar.pdf.

[43] M. Russinovich. Strings v2.40, April 2007. Retrieved on January 10, 2009, from http://technet.microsoft.com/en-us/sysinternals/bb897439.aspx.

[44] WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor. X-Ways Software Technology. Retrieved on February 11, 2009, from http://www.x-ways.net/winhex/.

[45] Windows Grep - Advanced Searching for Windows. Retrieved on February 5, 2009, from http://www.wingrep.com/.

[46] Q. Zhao and T. Cao. Collecting Sensitive Information from Windows Physical Memory. *Journal of Computers*, 4(1):3–10, January 2009. Retrieved on March 20, 2009, from http://www.academypublisher.com/jcp/vol04/no01/jcp0401003010.pdf.

[47] Alireza Arasteh. Forensic Analysis of Windows Physical Memory. Master's thesis, Concordia University, Montreal, Canada, June 2008.

[48] J. L. Undercoffer J. Butler and J. Pinkston. Hidden Processes: The Implication for Intrusion Detection. *Proceedings of Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 116–121, June 2003. Retrieved on March 16, 2009.

[49] M. Pietrek. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, February 2002. Retrieved on March 20, 2009, from http://msdn.microsoft.com/en-us/magazine/cc301805.aspx.

[50] S. Cho. Win32 Program Disassembler, February 2006. Retrieved on March 18, 2009, from http://www.geocities.com/~sangcho/disasm.html.

[51] FileZilla, The Free FTP Solution. Retrieved on February 11, 2009, from http://http://filezilla-project.org/.

[52] *IDA Pro Disassembler and Debugger. Hex Rays.* Retrieved on January 8, 2009, from http://www.hex-rays.com/idapro/.

[53] *PE Explorer, EXE File Editor Tool, DLL Reader, Disassembler, Delphi Resource Editing Software. Heaventools Software.* Retrieved on February 2009, from http://www.heaventools.com/.

[54] WSACleanup Function. Microsoft Networking Developer Platform Center, January 2009. Retrieved on February 11, 2009, from http://msdn.microsoft.com/en-us/library/ms741549(VS.85).aspx.

[55] Network Security Services (NSS). Mozilla Developers Center, January 2009. Retrieved on January 10, 2009, from http://www.mozilla.org/projects/security/pki/nss/.

[56] Reference for WriteFile Function. Microsoft Developer Network. Retrieved on February 11, 2009, from http://msdn.microsoft.com/en-us/library/aa365747(VS.85).aspx.

[57] PuTTY: A Free Telnet/SSH Client. Retrieved on March 17, 2009, from http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[58] SSH2 Clients Insecurely Store Passwords. iDEFENSE Security Advisory, Public Advisory 01.28.03. Technical report, iDefense Labs: Security and Vulnerability

Labs, 2003. Retrieved on January 15, 2009, from http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=24.

[59] Java Plug-in Framework (JPF) Project. Retrieved on March 15, 2009, from http://jpf.sourceforge.net/.

[60] C. Bauer and G. King. *Java Persistence with Hibernate.* Manning Publications Co, November 2006. Retrieved on March 12, 2009.

[61] M. Russinovich. Process Explorer v11.33. Technical report, Microsoft Corporation. Retrieved on January 10, 2009, from http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx.

[62] D. Bem and E. Huebner. Computer Forensic Analysis in a Virtual Environment. *International Journal of Digital Evidence*, 6(2), 2007. Retrieved on January 10, 2009, from http://www.utica.edu/academic/institutes/ecii/publications/articles/1C349F35-C73B-DB8A-926F9F46623A1842.pdf.

[63] D. Bem and E. Huebner. Analysis of USB Flash Drives in a Virtual Environment. *Small Scale Digital Device Forensics Journal*, 1(1), 2007. Retrieved on January 10, 2009, from http://www.ssddfj.org/papers/SSDDFJ_V1_1_Bem_Huebner.pdf.

[64] Glossary of DIA SDK Terms. Microsof, MSDN Library. Retrieved on December 18, 2008, from http://msdn.microsoft.com/en-us/library/5e6y0hkw(vs.71).aspx.

Appendix A

Pseudo Code of Data Files Extraction

```
viewSize = 256 * 1024;
VACBList = emptyList();
for(all processes in the process list) {
   for(all FileObject structures related to this EProcess) {
      SOPAddress = FileObject.SectionObjectPointersAddress;
      SOP = Read_SectionObjectPointer_From_Address(SOPAddress);
      SCMAddress = SOP.SharedCacheMapAddress;
      SCM = Read_SharedCacheMap_From_Address(SCMAddress);
      numberOfVACB = SCM.FileSize / viewSize;
      for(i = 0; i < numberOfVACB; i ++) {
         VACBStartAddress = SCM.InitialVACBOffset + (4 * i);
```

```
            tempVACB = fillVACB(VACBStartAddress);
            for(j = 0; j < viewSize / pageSize; j ++) {
               dataAddress = tempVACB.BaseAddress + (pageSize * j);
               physicalDataAddress = translateVirtualToPhysical(
                                                  dataAddress);
               readDataPageByPage(physicalDataAddress);
               writeMemoryContentsToFile();
            }
         }
      }
   }
```

Appendix B

Pseudo Code of Executable Files Extraction

```
for(all processes in the process list) {
   PEB = EProcess.PEB;
   ImageBaseAddress = PEB.ImageBaseAddress;
   ImageDosHeader = Read_ImageDosHeader_From_Address(
                                        ImageBaseAddress);
   ImageNTHeadersOffset = ImageDosHeader.e_lfanew;
   ImageNTHeaders = Read_ImageNTHeaders_From_Address(
      ImageBaseAddress +
      ImageNTHeadersOffset);

   ImageFileHeaderAddress = ImageBaseAddress +
```

```
   ImageDosHeader.e_lfanew +
   ImageNTHeaders.fileHeaderOffset;

ImageFileHeader = Read_ImageFileHeader_From_Address(
                                    ImageFileHeaderAddress);
ImageOptionalHeaderAddress = ImageBaseAddress +
   ImageDosHeader.e_lfanew +
   ImageNTHeaders.optionalHeaderOffset;

ImageOptionalHeader = Read_ImageOptionalHeader_From_Address(
   ImageOptionalHeaderAddress);

CreateEmptyExecutableFile();
WriteToFile(ImageDosHeader);
WriteToFile(ImageFileHeader);
WriteToFile(ImageOptionalHeader);

for(k = 0; k < ImageFileHeader.numberOfSections; k++) {
   ImageSectionHeaderAddress = ImageBaseAddress +
   ImageNTHeadersOffset +
   ImageNTHeaders.optionalHeaderOffset +
   ImageFileHeader.sizeOfOptionalHeader +
```

```
        (k * SizeOfImageSectionHeader);


        ImageSectionHeader = Read_ImageSectionHeader_From_Address(
            ImageSectionHeaderAddress);
                WriteToFile(ImageSectionHeader);
    }


    for(j = 1; j <= ImageFileHeader.numberOfSections; j++) {
        FileSection = Read_Section_From_Address_With_Size(
            ImageBaseAddress +
            ImageSectionHeader[j].VirtualAddress,
            ImageSectionHeader[j].VirtualSize);
            WriteToFilePageByPage(FileSection);
    }
}
```