Forensic Analysis of Windows Physical Memory

Ali Reza Arasteh

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 2008

© Ali Reza Arasteh, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-42529-9
Our file Notre référence
ISBN: 978-0-494-42529-9

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

Forensic Analysis of Windows Physical Memory

Ali Reza Arasteh

With the ubiquitous application of IT in different industries, digital forensic has become an essential element in IT security for discovering and mitigating the root causes of IT incidents. In this context, forensics memory analysis has recently gained great attention in cyber forensics community. However, most of the proposals in this area have focused on the extraction of important kernel data structures such as executive objects from the memory. This thesis discusses techniques for forensic analysis of Windows physical memory. The state of the art on digital forensic with focus on memory forensic is elaborated in this thesis. Additionally the thesis introduces new techniques for Windows memory forensics. The techniques that are elaborated in this thesis are classified into two categories; physical memory parsing, and execution history analysis. The first category introduces different in-memory structures of Windows operating system that are of forensic value during a digital investigation. The second category proposes an approach to analyze the stack memory of process threads to reveal partial execution histories of processes. The result of applying this technique enables the investigator to discover what actions performed by processes at the time of the incident. An algorithm was developed for this purpose that produces all the possible execution history paths. At the end, the introduced techniques are evaluated and empirical results are provided.

This thesis is dedicated to my wonderful parents who have been with me every step of the way, through good times and bad. Thank you for all of the unconditional love, guidance, and support that you have always given me, helping me to succeed.

Acknowledgments

This dissertation could not have been written without the help and support of Pr. Mourad Debbabi who not only served as my supervisor but also encouraged and supported me throughout my academic program. He has been a great teacher and manager during my Masters degree and I thank him for his endless patience, enlightening guidelines and unconditional support. Pr. Debbabi has been like a father to me during my studies and I am truly grateful to him. I convey special acknowledgement to the Ministry and Faculty for providing the financial means and laboratory facilities. It is a pleasure to express my gratitude wholeheartedly to Mr. Farzad Kohantorabi, Pooria Ansari and Mehdi Hedjazi for their help and support during this period. They have been true friends to me and their constructive comments have been of great help throughout my Masters. Finally, I take this opportunity to express my profound gratitude to my beloved parents for their moral support and patience during my study at Concordia University.

ALI REZA ARASTEH

Concordia University

June 2008

Contents

Chapter 1 Introduction **1**

Chapter 2 Forensics Analysis of Windows Memory **8**

2.1 Windows Operating System Architecture 9

2.2 Objects 11

2.2.1 Memory pools 15

2.2.2 Handle table 18

2.3 Objects Internal Structure 23

2.3.1 _EJOB Structure 25

2.3.2 _EPROCESS Structure 25

2.3.3 Process Environment Block 29

2.3.4 _rtl_user_process_parameters Structure 31

2.3.5 _PEB_LDR_DATA Structure 33

2.3.6 _KPROCESS Structure 35

2.3.7 _ETHREAD Structure 37

2.3.8 _KTHREAD Structure 38

Chapter 3 Memory, Security and Cache Management **46**

3.1 Memory Manager 46

3.1.1 Virtual Memory 47
3.1.2 Page Frame Database 54
3.2 File Extraction 65
3.2.1 Cache Manager 85
3.3 Security Manager 92

Chapter 4 Digital Investigation and Memory Forensics **112**
4.1 Digital forensics 112
4.2 Forensic analysis of physical memory 113

Chapter 5 Stack Trace Analysis **124**
5.1 Approach 124
5.2 Modeling the process and the stack traces 126
5.2.1 Control Flow Graph 129
5.2.2 Stack trace verification 136
5.2.3 Variable length stack frames 145
5.3 Design and Implementation 147
5.4 Experimental Result 154

Chapter 6 Conclusion **164**

Bibliography **167**

Appendices **173**

Chapter 1

Introduction

With the advent of computer technology and ubiquitous use of computers in numerous aspects of human life, cyber security has drawn great attention in the past few years. Having interconnected networks of computers as an integral part of every industry has made our lives exceedingly dependent on sound and reliable operation of the underlying information technology infrastructures. These reliability requirements very often stem from the nature of the industry or field in which IT is being used. Health industry, financial companies, governmental agencies, telecommunication, and customer service units are among many industries that are becoming more and more relying on flawless operation of their underlying IT infrastructure. The service level agreements that are guarantied by service providers can destroy the whole company's prospect and credibility in the event of service unavailability due to IT malfunctioning.

In this context, security professionals have strived to devise new techniques, approaches and solutions to improve the overall security of the IT infrastructure. Various solutions have been invented to automatically detect, stop and prevent the attacks against IT infrastructures. Many governmental and private funds have been allocated to

research and development in cyber security. A variety of security software and hardware products has been introduced to the market whose main goal is to protect confidentiality, integrity and availability of systems and services.

Despite this increased awareness of IT industry of security concerns and the recognition of the need for a secure and reliable infrastructure that is immune to different cyber attacks, current security practices have not been able to provide IT with a secure, practical and effective basis on which different industries can build their required functionalities. Many security products are either too costly to be adaptable by mid-sized and small businesses or have strong resource requirements that renders them inapplicable in resource-intense environment. Intrusion detection systems [22] have proved to be ineffective in correct detection of malicious activities and are notorious for flooding their stake holders with a plethora of false positives unless a great amount of effort is put into tuning them. Even after excessive tuning, they can not keep up with the traffic in large networks and can either play the role of a choke point in the network (i.e. when used in in-line mode) or miss some malicious traffic in well-designed attacks [30]. Firewalls are only able to detect certain obvious attacks and are not usually aware of the application layer flows where the majority of the attacks and vulnerabilities exists. Many companies underestimate the importance and destructive effects of insider threat and have limited or no visibility to their internal network activities in the event of internal security incident.

These factors and the borderless nature of cyber attacks have let many criminals/offenders walk away due to the lack of supporting evidence for conviction. In this context, *cyber forensics* [21] plays a major role by providing scientifically proven methods to gather, process, interpret, and use digital evidence to elaborate a conclusive description of cyber crime activities, stop the ongoing criminal activities against

or using IT infrastructures and provide recommendation to prevent future attacks. A forensics investigation is initiated in the sequel of a *security incident* notification. A security incident [35] is defined as any event that have potentially lead to or includes the breach of the *security policy* of the company. A security policy is a statement of management strategy as regards security. A digital forensic process could also be initiated for purposes other than security such as retrieval of lost information, determining the root cause of a failure, etc.

The digital forensics process [7] is defined as a sequence of steps that are followed in a predefined order during a digital forensic investigation. A comprehensive forensic investigation process usually includes the following steps:

- Notification and escalation of the incident to the proper investigation authority (i.e. Incident Handling and Response Team, Law Enforcement Authorities, Computer Emergency Response Teams (Certs)).
- Verification of the incident to assure the existence of the incident and determine the extent of the damage.
- Planning for containment, eradication and root cause investigation.
- Acquiring the authorization to collect the evidence that is used during the investigation.
- Collecting evidence in a forensically sound manner to prevent any change to the evidence.
- Analysis of the evidence to determine the root-cause of the incident.
- Preparation of the investigation report or other required means of presenting the result of the investigation to the requesting authority.

- Follow-ups that include the required actions to prevent future incidents, lesson learnt, and the disposition of the evidence.

In a commercial software market flooded by security products, the development of forensics IT solutions for law enforcement has been limited. Though outstanding results have been achieved for forensically sound acquisition of evidence, little has been done on the analysis of the acquired evidence. This is particularly evident for volatile evidence such as physical memory and system cache, which is mainly due to the volatile and unstable nature of data that resides on these types of storage. However, if not damaged, the information that is acquired from such sources is one of the most pertinent and definitive evidence and therefore should be analyzed during the initial phases of the investigation.

Physical memory as forensic evidence has recently received some attention in forensic community. However, most of the work on physical memory analysis is limited to forensically sound acquisition of physical memory using different software and hardware solution and the extraction of forensically important kernel data structures such as structures that represent processes and files from the acquired image of the physical memory. This is while, knowing which processes were executing or which files have been opened during the incident do not allow to answer many questions that involve the order of the activities performed by the attacker or what a process has done during its course of execution.

This thesis contributes to the state of the art research on forensic analysis of physical memory in several areas as follows:

- Provides details on physical memory layout of Windows operating system and discusses the inner details of Windows kernel components. This information has been acquired through reverse engineering of Windows operating system code as

well as from different books, forums and papers.

- Identifies Windows kernel components and structures that entail forensically valuable information.
- Develops a process for forensic analysis of Windows physical memory.
- Proposes an approach for performing stack trace analysis on the retrieved information to extract a partial execution path of the process.

In this thesis, the author introduces new techniques for forensics analysis of Windows physical memory. These techniques are classified into two categories: evidence extraction and execution reconstruction. The first category of these techniques discusses different pieces of evidence that are useful during a digital investigation and are retrievable from an image acquired from physical memory. These pieces of evidence are mainly information that is stored by the operating system in different data structures about various operations and processes that are executing. In introducing these techniques, several structures that are maintained by different components of Windows operating system such as memory manager, process manager, etc. are addressed and the relevancy of the information that they store is discussed from a forensics standpoint. The majority of these techniques have been presented and known to digital forensics community. However, in this study, the author tries to bring together all of these techniques and fill some of the existing gaps with his own findings. These findings mainly consist of several previously undocumented Windows operating system structures that are relevant to forensic analysis and approaches for more effective analysis of these structures.

The second category of Windows physical memory analysis techniques discusses an approach to reconstruct the execution of processes that were executing at the time the image was taken from the physical memory. These techniques consist of two steps.

The first step is to model the execution of a process by analyzing the process executable. The second step is to find all of the execution paths in the process execution model that generate an execution trace that matches the existing traces in the physical memory image. In this study, we only focus on the execution traces that exist on the process stack. Therefore, the execution paths that are detected by this technique are in the form of a chain of function calls and returns. More accurate results could be achieved by including other execution traces such as process heaps into the analysis. It is important to notice that this research is mainly focused on the forensic analysis of the physical memory of Windows operating system.

Due to the fact that Windows is a closed source operating system, little documentation and tools are available on the forensic analysis of this operating system. Nevertheless, Windows is one of the most prevalent operating systems that is being used in almost any environment and therefore is involved as a source of evidence in many investigations. However, it is important to emphasize the fact that many of the techniques that are discussed in this thesis can be applied to the forensic analysis of physical memory of other operating systems while some details might differ. The first category of the introduced techniques are dependent on the internal structures of Windows operating system and are therefore only applicable to this operating system. Similar approaches exist for Unix based operating systems that are out of the scope of this thesis. The second category of techniques for forensic analysis of physical memory, however, is not exclusive to any operating system since it only depends on the application of stack mechanism for implementing function calls and returns, which is the mechanism used by most of the existing modern operating systems.

The rest of this thesis is structured as follows: Chapter two starts with a background on Windows operating system, different components of this operating system

and the interaction of these different components with each other and the external environment. The chapter continues by introducing Windows object management and forensically valuable information that could be acquired by investigating these objects. Chapter three discusses other valuable information that could be extracted from Windows security manager, cache manager and memory manager. These two chapters describe the first category of Windows physical memory forensic analysis techniques. In discussing each technique, relevant Windows structures are detailed. Chapter four discusses the state of the art in digital investigation and forensic analysis of Windows physical memory in detail. In this chapter, the author frequently refers to Windows operating system structures that are introduced in Chapters two and three. Chapter five elaborates on our approach in forensic analysis of Windows physical memory. This chapter discusses the second category of forensic analysis techniques that are detailed in this thesis. The chapter continues with a discussion on the implementation details of the system developed for forensic analysis of Windows physical memory using the techniques introduced in this thesis. This chapter ends by providing some empirical analysis results from using the developed system to analyze images taken from several systems. Chapter six, concludes this discussion and proposes some possible future research directions.

Chapter 2

Forensics Analysis of Windows Memory

This chapter and chapter three introduce the preliminary techniques for the forensic analysis of Windows operating system physical memory. The information provided in these chapters constitutes the primary knowledge required in forensic analysis of Windows physical memory. The more advanced analysis techniques that are discussed in later chapters are built on the background information that is provided in these chapters. This chapter focuses on Windows operating system structures, object management and process management. The next chapter describes other related components of Windows including security management, memory management and cache management.

Please notice that since the exact details of some of Windows operating system structures are different based on the Windows version and even from one service pack to another, due to space limitations, in this thesis only Windows XP service pack 2 is discussed. For other versions of Windows, many concepts are the same as presented here with slight differences in some structure fields and offsets. This chapter starts with an

introduction to the overall architecture of Windows operating system. This introduction is followed by a detailed discussion on different Windows operating system components that are of relevance during forensic investigation of a memory image.

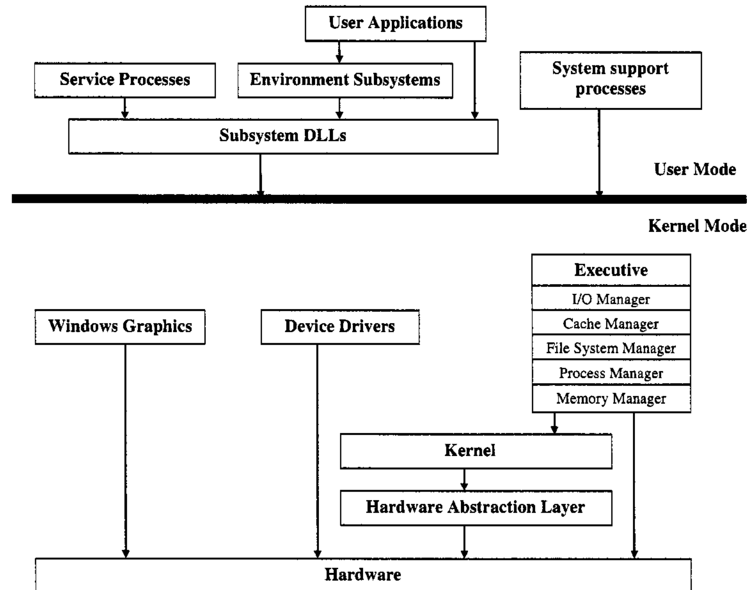
2.1 Windows Operating System Architecture

Figure 2.1 shows the overall architecture of Windows operating system [29]. Windows executes in two modes: User mode and kernel mode. Components that execute in the user mode are:

- System support processes, such as the winlogon process. These processes are not started by the service control manager of Windows.
- Windows service processes such as Task Scheduler and Spooler services.
- User applications, that host the ordinary processes running under user logons. These processes can be of six types: Windows 32-bit, Windows 64-bit, Windows 3.1 16-bit, MS-DOS 16-bit, POSIX 32-bit, or OS/2 32-bit.
- Environment subsystem server processes, that implement part of the support for different operating system environments. Each process might use one of these server processes based on its type and required services.
- Subsystem DLLs act as a wrapper for kernel services and are used by user applications and environment subsystem server processes.

In Windows, user applications don't call the native Windows operating system services directly; rather, they go through one or more subsystem dynamic-link libraries (DLLs). The role of the subsystem DLLs is to translate a documented function into the

Figure 2.1: Windows operating system overall architecture



appropriate internal (and generally undocumented) Windows system service calls. The kernel-mode components of Windows include the following operational units:

- Windows executive that includes memory manager, process and thread manager, security manager, I/O manager, networking, and interprocess communication manager.
- Windows kernel that handles the low-level operating system functionalities including scheduling, interrupt and exception dispatching.
- Device drivers including both hardware device drivers that handle the I/O operations for each hardware and the drivers for file system and network communication.
- Hardware abstraction layer (HAL) that is an abstraction layer implemented in software that isolates the kernel from the hardware platforms differences.

- Windowing and graphics system that provides for the graphical user interface (GUI) functions such as handling Windows, user interface controls, and drawing.

2.2 Objects

Windows object manager is a component of Windows Executive that provides for a unique interface for creation and handling of objects. In Windows, Each object represents an entity that is created during the operating system operation. Windows uses two sets of objects: The kernel objects and the executive objects. Executive objects are objects created by different Windows executive components such as memory manager, and process manager. Process, threads and section objects are examples of the executive objects. Kernel objects are not accessible to user mode applications. Examples of kernel objects include mutant object that are used for synchronization.

From the forensics stand point, executive objects contain most useful information that can be extracted about the incidents from memory. This information includes the processes, threads, files and registry keys accessed by the process, etc. Each object has an object header and an object body. The object header is used by object manager to manage the objects and the object body is controlled by the component that creates the object. An object header points to the list of processes that has access to that object. Below is the structure of an object header. The structure layout is produced using the `dt` command in Windbg debugger. This command displays type information about different structures in Windows (Figure 2.2).

The object header contains information that helps object manager to maintain object operation. This information contains the number of pointers and handles to an object, quota information, etc. A forensic analyst can use the information contained in this structure to identify the type of an object, the name of it, who has access to

```
kd> dt _object_header nt!_OBJECT_HEADER
   +0x000 PointerCount      : Int4B
   +0x004 HandleCount       : Int4B
   +0x004 NextToFree        : Ptr32 Void
   +0x008 Type              : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset    : UChar
   +0x00d HandleInfoOffset  : UChar
   +0x00e QuotaInfoOffset   : UChar
   +0x00f Flags             : UChar
   +0x010 ObjectCreateInfo  : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body              : _QUAD
```

Figure 2.2: Using `dt` command to view the details of `_object_header` structure.

an object and verify if the object is deleted. Below are the description of forensically important fields:

- Field *PointerCount* is the number of pointers from the kernel components to the object.

- Field *HandleCount* is the number of handles that processes has opened to this object. An object can be accessed both by user-land processes and kernel components. User-land processes can only access an object through a handle. Kernel components can also have pointers to the object. A pointer is the memory address at which the object is located. A handle is an index into the handle table as discussed later in this section. A forensic analyst can verify if an object is deleted or is still in use by checking the number of handles and pointers to the extracted object. If both of these counters (number of handles and pointers) are zero, then the object is no longer used by any component.

- Field *ObjectType* contains information about all objects of a specific object type

and links together all those objects. This field can be used to identify the type of an object. Moreover, in order to find all of the objects of the same type (such as all of the processes), an analyst can use the information contained in this structure to locate the beginning of the list that links all of these objects together. Below is the structure of _object_type:

```
kd> dt _object_type ntdll!_OBJECT_TYPE
   +0x000 Mutex                : _ERESOURCE
   +0x038 TypeList             : _LIST_ENTRY
   +0x040 Name                 : _UNICODE_STRING
   +0x048 DefaultObject        : Ptr32 Void
   +0x04c Index                : Uint4B
   +0x050 TotalNumberOfObjects : Uint4B
   +0x054 TotalNumberOfHandles : Uint4B
   +0x058 HighWaterNumberOfObjects : Uint4B
   +0x05c HighWaterNumberOfHandles : Uint4B
   +0x060 TypeInfo             : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key                  : Uint4B
   +0x0b0 ObjectLocks          : [4] _ERESOURCE
```

Structure _LIST_ENTRY is a doubly linked list structure. This structure is used in Windows kernel whenever there is a need for a doubly linked list. Using this structure, a type object for a process links all the object of a specific type together.

```
kd> dt _LIST_ENTRY
   +0x000 Flink                : Ptr32 _LIST_ENTRY
   +0x004 Blink                : Ptr32 _LIST_ENTRY
```

Flink is the forward link pointing to the next structure and Blink is the Backward link pointing to the previous structure.

- Field *NameOffset* specifies the offset of the structure _OBJECT_HEADER_NAME_INFO from the beginning of structure _OBJECT_HEADER. This structure contains naming information for the object. However, this offset should be subtracted from the address of structure _OBJECT_HEADER meaning that, if present, structure _OBJECT_HEADER_NAME_INFO is before structure _OBJECT_HEADER. Typical values are 0, 10 or 20, depending on the presence of a _OBJECT_CREATOR_INFORMATION header part. This structure is detailed below:

```
kd> dt nt!_OBJECT_HEADER_NAME_INFO
   +0x000 Directory         : Ptr32 _OBJECT_DIRECTORY
   +0x004 Name              : _UNICODE_STRING
   +0x00c QueryReferences   : Uint4B
```

  Field Name contains the name of the object. However not all object use this structure to store the name of the object. The name field is of type _UNICODE_STRING, which is a structure used in Windows to store strings. As shown below, field *Buffer* is a pointer to the beginning of the string and field *Length* specifies the length of the string.

- Field *HandleDBOffset* is the offset value that should be subtracted from the base address of the header to obtain the address of structure OBJECT_HANDLE_DB. This structure contains the list of processes that have handles to the object. In [31] this structure is defined as below:

```
struct _OBJECT_HANDLE_DB {
```

```
 union {
  struct _EPROCESS  *Process;
  struct _OBJECT_HANDLE_DB_LIST *HandleDBList;
 }
 DWORD HandleCount;
}

struct _OBJECT_HANDLE_DB_LIST {
  DWORD Count;
  OBJECT_HANDLE_DB Entries [];
}
```

If only one process has opened a handle to an object, then flag OB_FLAG_SINGLE_PROCESS in field ObjectFlags is set and field Process points to a valid process block. If flag OB_FLAG_SINGLE_PROCESS is cleared, then field HandleDBList points to a list of type OBJECT_HANDLE_DB_LIST that contains an array of structures of type OBJECT_HANDLE_DB and field HandleCount contains the number of handles to the object.

The rest of the fields in this structure are for quota and security management which are not relevant to our discussion.

2.2.1 Memory pools

Windows objects are allocated in memory storages called pools. Windows has two types of pools: paged pools and non-paged pools. The former is the memory that can be paged out to the page file while the latter is always resident in the physical memory and is never paged out. Memory pools are created by the kernel at the start time of the system

and depending on the allocation requirements of the system could be expanded or freed later on. Pool allocations are used for the allocation request that are smaller than the page size. Each allocated unit starts with a pool header structure of type _POOL_HEADER. This structure is detailed below:

```
kd> dt _POOL_HEADER
   +0x000 PreviousSize      : Pos 0, 9 Bits
   +0x000 PoolIndex         : Pos 9, 7 Bits
   +0x002 BlockSize         : Pos 0, 9 Bits
   +0x002 PoolType          : Pos 9, 7 Bits
   +0x000 Ulong1            : Uint4B
   +0x004 ProcessBilled     : Ptr32 _EPROCESS
   +0x004 PoolTag           : Uint4B
   +0x004 AllocatorBackTraceIndex : Uint2B
   +0x006 PoolTagHash       : Uint2B
```

Considering the fact that Windows objects are allocated in memory pools, it is possible to retrieve allocated or deleted object by searching in system pools and analyzing every allocated pool entity. Structure _POOL_HEADER can be used to locate these allocation units. The following is the description of the important fields of this structure:

- Field *PreviousSize* is the size of the previous pool block in eight-byte units.
- Field *BlockSize* is the size of the described pool block in eight-byte units.
- Field *PoolType* is the type of the pool. Using this field we can identify if the pool is paged or non-paged.

- Field *Ulong1*, if valid, points to the `_EPROCESS` block of the process whose allocation is charged for the allocation of this block.

- Field *PoolTag* is the label of the pool unit which identifies the content of the pool. However, it is important to notice that there is no authorization process for the use of a specific tag. Therefore, a process can use any tag for this field. These tag values are defined in `c:/Program Files/Debugging Tools for Windows/Triage/Pooltag.txt`. Below are some important tags with their description.

```
CM   - nt!cm        - Configuration Manager (registry)
Cc   - nt!cc        - Cache Manager allocations (catch-all)
File - <unknown>    - File objects
Proc - nt!ps        - Process objects
Thre - nt!ps        - Thread objects
Devi - <unknown>    - Device objects
Driv - <unknown>    - Driver objects
Key  - <unknown>    - Key objects
Sect - <unknown>    - Section objects
Symb - <unknown>    - Symbolic link objects
Toke - nt!se        - Token objects
NDPt - ndis.sys     - TCPIP
```

By scanning for these tag values in the physical memory, Windows executive objects can be identified and extracted. This technique is discussed in more details in chapter two.

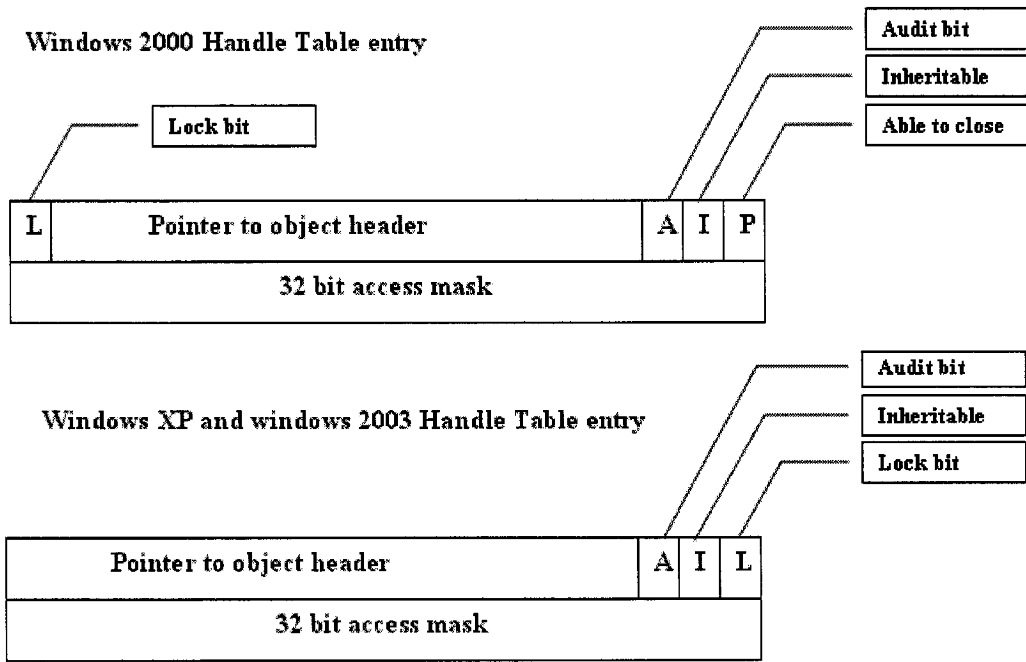
2.2.2 Handle table

In order for a process to use an object it must acquire a handle to it. A handle is an index to the process handle table. The process handle table is pointed by its _EPROCESS block and contains pointers to the objects that the process has a handle to. The process handle table is implemented using a three level scheme. The first level contains pointers to the middle level tables. The middle level tables contain arrays of pointers to sub-handle tables. The sub-handle tables contain the address of the objects.

In Windows 2000, at the time of the process creation, all of the tables in the three levels are allocated. The low 24 bits of the object handles is divided into three 8 bit fields each being an index to the relative handle table. In Windows XP and Windows 2003, the tables are created as needed and only the lowest level table is created at the process creation time. Each table consists of 8 byte entries and the size of the table is the number of entries that fit into a page minus one. The subtracted entry is for auditing purposes. Therefore, in Windows XP and Windows 2003, depending on the size of the table, the addressing scheme deffers.

Figure 2.2.2 shows the structure of a handle table entry. On 32 bit system each handle is 4 bytes long. The pointer to the object header or handle table is 24 bits in Windows 2000 and in Windows XP and Windows 2003 it is 31 bits. Since the entries are in the system address space, the first bit is always one and therefore this bit can be used for locking purpose. This way, the object manager locks the entire process handle table only when the process is creating a new handle or closing an existing handle. The rest of the times, the object manager locks the entry only and lets other threads to use other entries in the handle table.

Figure 2.3: Handle table entry structure



The address of the handle table of a process is stored in the field HandleTable in the _EPROCESS block of the process. This field points to a structure of type _HANDLE_TABLE. This is structure is shown below:

```
kd> dt _HANDLE_TABLE
   +0x000 TableCode           : Uint4B // This is the address of
                                       // the Top level table.
   +0x004 QuotaProcess        : Ptr32 _EPROCESS
   +0x008 UniqueProcessId     : Ptr32 Void // Table owner process ID
   +0x00c HandleTableLock     : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList     : _LIST_ENTRY
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo           : Ptr32 _HANDLE_TRACE_DEBUG_INFO
   +0x02c ExtraInfoPages      : Int4B
   +0x030 FirstFree           : Uint4B
   +0x034 LastFree            : Uint4B
   +0x038 NextHandleNeedingPool : Uint4B
   +0x03c HandleCount         : Int4B Number of handle entries.
   +0x040 Flags               : Uint4B
   +0x040 StrictFIFO          : Pos 0, 1 Bit
```

Field TableCode of this structure points to the top level table. Figure 2.2.2 depicts the mechanism for translating a handle to the virtual address of the object it is referring to.

Figure 2.4: Handle to object address translation and a view of handle table

According to the Figure 2.2.2, the general handle translation formula is as follows:

```
Entry_number = (Handle / 4)
If (Entry_number >= ((page_size) * (page_size) / 32) {
                                        // There are three levels
    Middle_table_address = *(_EPROCESS.ObjectTable
                + [Entry_number/(page_size)*(page_size)/32]);
    Handle_table_address = *( Middle_table_address
                + [Entry_number/(size of a page/8)]);
    Object_header_address = *(Handle_table_address + Entry_number*8)-1;
    Object_address = Object_header_address + 0x18;

} else  if (Entry_number >= size of a page/8){
                                        // There are two levels


    Handle_table_address = *(_EPROCESS.ObjectTable
                +[Entry_number/(size of a page/8)]);
    Object_header_address =  *(Handle_table_address + Entry_number*8)-1;
    Object_address =  Object_header_address + 0x18;

} else{ //There is one level
    Handle_table_address =  (*_EPROCESS.ObjectTable);
    /* The first 8 bytes are for auditing
    * purpose and are not pointing to an object
    * address.
```

```
    */
    Object_header_address =  *(Handle_table_address + Entry_number * 8)-1;
    Object_address =  Object_header_address + 0x18;
}
```

Using the information stored in the handle table of a process, the forensic analyst can locate all of the objects that a process is using or have used previously but is in use by kernel or other processes. Each object, depending on its type, can provide useful information to the analyst. In the following sections, different object types that can contain forensic related information are discussed.

2.3 Objects Internal Structure

Until now, the general structure of Windows objects including object headers and pool headers were described. Figure 2.3 shows important Windows memory manager, object manager and process manager in-memory structures and their inter-relationships. Since each object, depending on its type, has a different body structure, in the following sections, each object type is discussed separately. In discussing each type, we detail on information that it stores and can be of forensic value during a digital investigation.

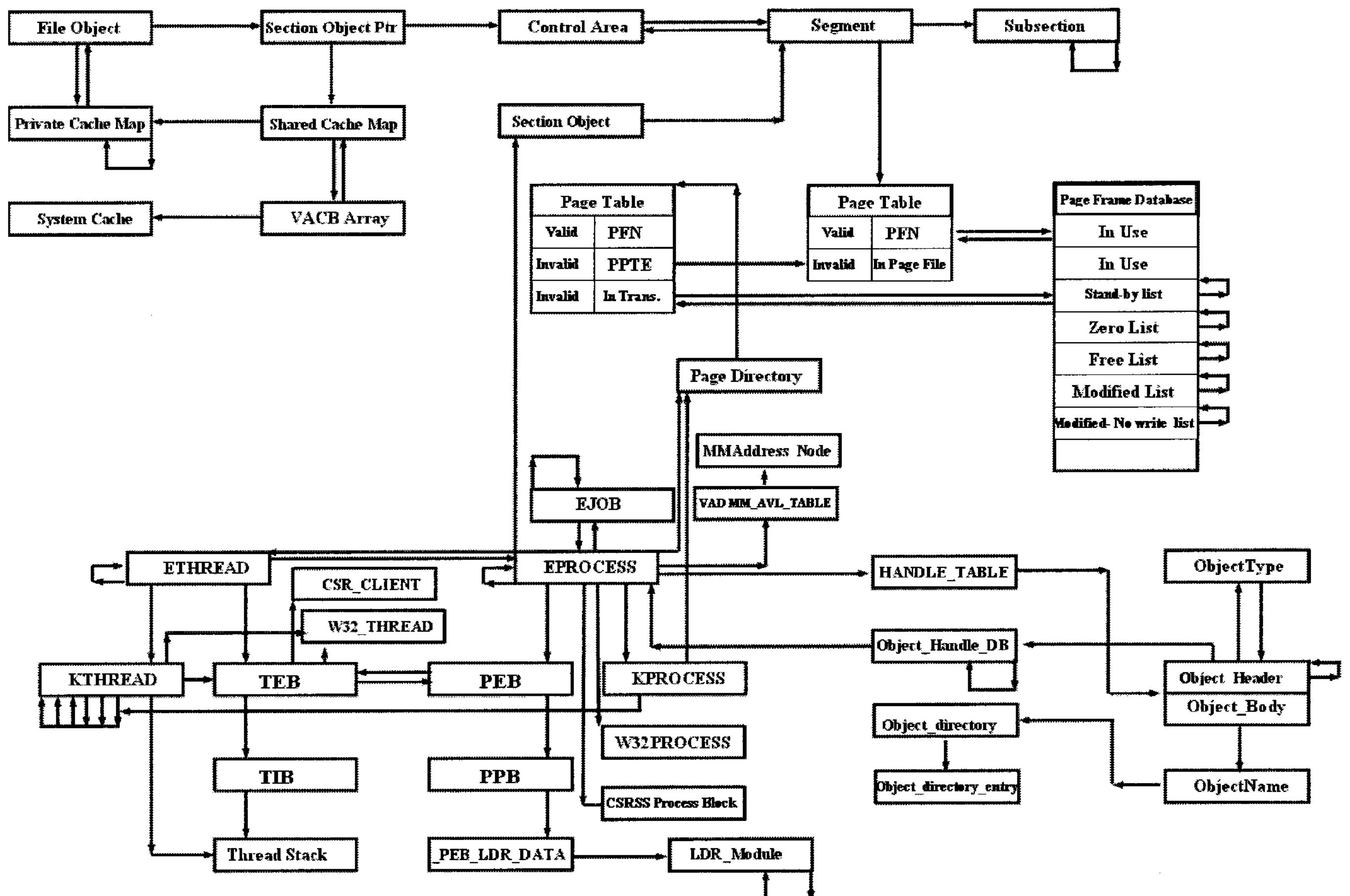


Figure 2.5: windows internal structures and their inter-relationship

2.3.1 _EJOB Structure

A job is a container for a set of processes that enables sand-boxing. The structure describing a job is of type _EJOB. By default, all processes created by a process and their descendants are associated with the same job object. This feature can be utilized to identify all of the process created by a process or its descendants. A well-known technique used by many malwares to survive a system restart is to add an entry to one of the start-up registry keys that points to their executable. Processes involved in the system start-up (i.e smss, csrss, services, lsass, userinit, winlogon and explorer) read these start-up registry keys and run the executables specified by them. Job objects can be used to locate processes created by these start-up processes in order to find suspicious programs. Due to space limitations, the complete listing of this structure is provided in Appendix 1 and only forensic related fields are detailed below:

- Field *JobLinks* contains a doubly linked list of all the executive jobs inside the system. This linked list can be traversed to locate all of the job objects in the memory.
- Field *ProcessListHead* is the head of a doubly linked list of the processes inside the job. This link list can be traversed to locate all of the processes that were created by the head of this list.
- Field *SessionID*: This filed contain the session Id that the job is running under. This field can be later used to correlate job objects to different sessions.

2.3.2 _EPROCESS Structure

_EPROCESS is the main structure that describes a running process. For each process, there exist one *_EPROCESS* structure. All of these structures are linked in a doubly

linked list. This structure is allocated in the kernel land in kernel memory pool with the tag of *Proc*. This structure is the body of the Windows process object. Therefore, right before this structure, there exist an object header of type *Process*. This structure is a good starting point for the analysis of a suspicious process. Many information about the process name, time of the creation, opened files, dlls and threads are either stored or pointed by some fields in this structure. The forensic related fields in this structure are detailed below. For the complete listing of this structure, please refer to Appendix 1.

- Field *Pcb* is a structure that contains kernel related information for the process. This structure is allocated in kernel land and is described later.
- Field *CreateTime* contains the creation time of the process.
- Field *ExitTime* holds the exit time of the process.
- Field *UniqueProcessId* contains the process ID assigned to this process.
- Field *ActiveProcessLinks* is a doubly linked list of processes currently running.
- Field *SessionProcessLinks* is a doubly linked list of the structures of type _EPROCESS structures as before but only for the processes in the session.
- Field *ObjectTable* stores a pointer to the handle table containing the objects used by the process as described before.
- Field *Token* is a security token that contains the control access information for a process. The security manager uses this information to enforce security policies of the system.
- Field *VadRoot* points to the root of the Virtual address descriptor tree. Windows keeps an AVL tree of the virtual address ranges that have been allocated by the

process. Each node in this tree is called a VAD. The VAD tree helps Windows to allocated page table entries only when the region is accessed by the process. The overall data structure of the VAD is shown below:

```
struct vad {
    void *StartingAddress;
    void *EndingAddress;
    struct vad *ParentLink;
    struct vad *LeftLink;
    struct vad *RightLink;
    DWORD Flags;
    Struct _control_area *ca;
}VAD, *PVAD;
```

If field `Flags` is `ViewUnmap`, the VAD is describing a private area, if it is `ViewShare`, the VAD is pointing to a shared area and therefore the control area structure points to a valid object. Field `StartingAddress` is the starting address for the virtual address range that this VAD represents. Field `EndingAddress` is the ending address for the virtual address range that this VAD represents. Fields `ParentLink`, `LeftLink`, and `RightLink` are used to implement a binary three of VAD structures and are parent node, left child node, and right child node of the current VAD respectively. As detailed in the next chapter, this structure can be used to extract the memory used by a process.

- Field *VadHint* points to the last VAD entry that has been allocated.
- Field *Win32Process* points to a structure of type `_W32PROCESS` that exists in win32.sys Windows driver. The details of this structure are not known. How-

ever, here are some facts about this structure that can be of forensics importance. The first four bytes of this structure is a pointer that points back to the _EPROCESS block of the process. At the offset of 0x30, there is a looped pointer that points back to the address of itself. At the offset 0x98, there is a pointer that points to the next _W32PROCESS structure forming a single linked list. These three characteristics can be used for the detection of hidden, lost or partly overwritten _EPROCESS blocks by searching through the memory looking for a structure with the stated properties.

- Field *Job* points to the _EJOB structure that the process is associated with.
- Field *SectionObject* points to a structure of type _SECTION_OBJECT that describes the mapped memory used for loading the image. This field can be used to extract the process image from the memory as discussed in more details later in this chapter.
- Field *SectionBaseAddress* points to the image base of the process. This address is the virtual address of the beginning of the process image inside the memory.
- Field *InheritedFromUniqueProcessId* contains the process ID of the process that has created this process.
- Field *Session* stores the Terminal Services Session ID that is the the ID of the terminal session in which the processes is running.
- Field *ImageFileName* contains the 17 characters of the name of the image file of the process.
- Field *JobLinks* is a list entry that lists all processes that are associated with this process job.

- Field *ThreadListHead* is the head of a doubly linked list of structures of type _ETHREAD for each of the running threads of the process. The _ETHREAD structure is described later in this chapter.
- Field *Peb* points to the process environment block of the process and is detailed later in this chapter.
- Field *ModifiedPageCount* contains the number of pages of memory that have been modified by this process.
- Field *JobStatus* contains the status of the job the process is part of.
- Field *Flags* is a flag that specifies the creation status of a process. The meaning of each bit in the this 32-bit is specified in the structure definition.
- Field *ExitStatus* stores the exit code of the process.

2.3.3 Process Environment Block

This is a high-level user-land structure which contains some of a process properties and attributes. This structure can be used to:

- Determine the base address of the process in memory. This base address can be used for the extraction of the executables as discussed later in this chapter.
- Identify OS version information.
- Find the address to the location that information about dlls used by the process is stored.
- Locate the structure that stores information on the process execution parameters.
- Acquire process heap information.

For the complete listing of this structure, please refer to Appendix 1. The following fields were identified to contain forensic related information:

- Field *ImageBaseAddress* is the base address of the process in memory that is the memory address at which the process executable has been loaded.
- Field *Ldr* is a pointer to the `_PEB_LDR_DATA` structure that contains the dll related information of the process and is discussed later in this chapter.
- Field *ProcessParameters* is a pointer to a `_rtl_user_process_parameters` structure, which also contains loading data such as environment parameters for a running process. This structure is discussed later in this chapter.
- Field *NumberOfProcessors* specifies the number of processors of the system.
- Field *ProcessHeap* is a pointer to the process heap.
- Field *ReadOnlySharedMemoryBase* has a pointer to a system-wide shared memory location. It is usually 0x7f6f0000.
- Field *NumberOfHeaps* contains the number of heaps that has been created by the process.
- Field *ProcessHeaps* is a pointer to a pointer that lists all the heaps the process has.
- Field *OSMajorVersion* stores the major version of the OS.
- Field *OSMinorVersion* stores the minor version of the OS.
- Field *OSBuildNumber* stores the OS build number.
- Field *OSCSDVersion* stores the service pack number.

- Field *OSPlatformId* contains the platform ID of the OS.
- Field *ImageSubsystemMajorVersion* contains the major version of the subsystem.
- Field *ImageSubsystemMinorVersion* stores the minor version of the subsystem.
- Field *CSDVersion* has the service pack name in string format.

2.3.4 _rtl_user_process_parameters Structure

This structure contains the process runtime data such as the command line started the process, window title and run time data. This structure can be used to extract the following information:

- Process environment information.
- The command line instruction that started the process.
- The process executable address on the disk.
- Current directory, window and desktop name of the process.

The structure has a variable length and field *Length* contains the total length of it. For the complete listing of this structure, please refer to Appendix 1. The following fields have been identified as forensically relevant:

- Field *CurrentDirectory* contains the string showing the current directory of the process.
- Field *DllPath* is the list of directory paths that are searched for dlls needed by the process.
- Field *ImagePathName* the complete path to the process executable.

- Field *CommandLine* the command-line which started the process.

- Field *Environment* points to the address in memory that the environment variables for the process are stored. Below is a snippet showing part of the memory that contains the environment variables for a process.

```
kd> dc 0x00010000 1100
00010000  004c0041 0055004c 00450053 00530052  A.L.L.U.S.E.R.S.
00010010  00520050 0046004f 004c0049 003d0045  P.R.O.F.I.L.E.=.
00010020  003a0043 0044005c 0063006f 006d0075  C.:.\.D.o.c.u.m.
00010030  006e0065 00730074 00610020 0064006e  e.n.t.s. .a.n.d.
00010040  00530020 00740065 00690074 0067006e   .S.e.t.t.i.n.g.
00010050  005c0073 006c0041 0020006c 00730055  s.\.A.l.l. .U.s.
00010060  00720065 00000073 004e0041 005f0054  e.r.s...A.N.T._.
00010070  004f0048 0045004d 0043003d 005c003a  H.O.M.E.=.C.:.\.
00010080  0061006a 00610076 0061005c 00610070  j.a.v.a.\.a.p.a.
00010090  00680063 002d0065 006e0061 002d0074  c.h.e.-.a.n.t.-.
000100a0  002e0031 002e0037 002d0030 00690062  1...7...0.-.b.i.
000100b0  005c006e 00700061 00630061 00650068  n.\.a.p.a.c.h.e.
000100c0  0061002d 0074006e 0031002d 0037002e  -.a.n.t.-.1...7.
000100d0  0030002e 00410000 00500050 00410044  ..0...A.P.P.D.A.
000100e0  00410054 0043003d 005c003a 006f0044  T.A.=.C.:.\.D.o.
000100f0  00750063 0065006d 0074006e 00200073  c.u.m.e.n.t.s. .
00010100  006e0061 00200064 00650053 00740074  a.n.d. .S.e.t.t.
00010110  006e0069 00730067 0041005c 006d0064  i.n.g.s.\.A.d.m.
00010120  006e0069 00730069 00720074 00740061  i.n.i.s.t.r.a.t.
```

```
00010130  0072006f 0041005c 00700070 0069006c  o.r.\.A.p.p.l.i.
00010140  00610063 00690074 006e006f 00440020  c.a.t.i.o.n. .D.
kd> du 0x00010000
00010000  "ALLUSERSPROFILE=C:\Documents and"
00010040  " Settings\All Users"
```

- Field *WindowTitle* stores the window title of the running process.
- Field *DesktopInfo* contains the name of the desktop of the process.
- Field *ShellInfo* stores Windows shell information for the process.
- Field *RuntimeData* contains the strings that the process needs during execution.
- Field *CurrentDirectores* contains the dll paths that might be needed in an array of size 32 and type _RTL_DRIVE_LETTER_CURDIR. This structure is shown below:

```
0:001> dt _RTL_DRIVE_LETTER_CURDIR
   +0x000 Flags            : Uint2B
   +0x002 Length           : Uint2B
   +0x004 TimeStamp        : Uint4B
   +0x008 DosPath          : _STRING
```

The _STRING data type has the same structure as _UNICOEDEE_STRING.

2.3.5 _PEB_LDR_DATA Structure

This structure contains the list of the dlls that have been loaded by the the process. This structure has the following fields.

```
dt _PEB_LDR_DATA
   +0x000 Length            : Uint4B
   +0x004 Initialized       : UChar
   +0x008 SsHandle          : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress   : Ptr32 Void
```

The main useful fields in this structure are *InLoadOrderModuleList*, *InMemoryOrderModuleList*, and *InInitializationOrderModuleList* each containing the doubly linked list of the loaded dlls ordered by the initialization order, location in memory and defined loading order respectively. These linked lists, link structures of type `_LDR_DATA_TABLE_ENTRY` which has the following format:

```
kd> dt _LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase           : Ptr32 Void
   +0x01c EntryPoint        : Ptr32 Void
   +0x020 SizeOfImage       : Uint4B
   +0x024 FullDllName       : _UNICODE_STRING
   +0x02c BaseDllName       : _UNICODE_STRING
   +0x034 Flags             : Uint4B
   +0x038 LoadCount         : Uint2B
   +0x03a TlsIndex          : Uint2B
```

```
+0x03c HashLinks          : _LIST_ENTRY
+0x03c SectionPointer     : Ptr32 Void
+0x040 CheckSum           : Uint4B
+0x044 TimeDateStamp      : Uint4B
+0x044 LoadedImports      : Ptr32 Void
+0x048 EntryPointActivationContext : Ptr32 Void
+0x04c PatchInformation : Ptr32 Void
```

The first three fields are the structures of type `LIST_ENTRY`. Field *BaseAddress* contains the based address of where the module is mapped in virtual memory. Field *FullDLLName* and *BaseDLLName* are used for naming the dll file. Field *TimeDateStamp* is the time the dll was loaded in the memory. Field *SectionPointer* is a pointer so the section object representing this dll. Section object structure is discussed in detail later when we discussed Windows caching.

2.3.6 _KPROCESS Structure

`_KPROCESS` or Process Control Block (PCB) is kernel object that contains information about process threads scheduling. This structure can be used to:

- Develop a signature that can be used to locate the process objects.
- Find the page table directory of the process that is required for virtual to physical memory translation.
- Find all of the process threads.

For the complete listing of this structure, please refer to Appendix 1. Field *Header* in this is the dispatcher header that is used for synchronization purposes. This

header exist at the beginning process and thread objects. This structure is detailed below:

```
kd> dt _DISPATCHER_HEADER
   +0x000 Type              : UChar
   +0x001 Absolute          : UChar
   +0x002 Size              : UChar
   +0x003 Inserted          : UChar
   +0x004 SignalState       : Int4B
   +0x008 WaitListHead      : _LIST_ENTRY
```

Field *Type* is the type of object the dispatch header is defined for. For process objects this field is 0x3 and for thread objects this field has the value of 0x6. Field *Size* contains the size of the object in units of four bytes. A. Schuster in his paper titled "Searching for processes and threads in Microsoft Windows memory dumps", uses these information along with the pool header tags to define patterns for process and thread objects [32]. The memory image is then scanned to located these objects by looking for the specified patterns.

Another important field in this structure is *DirectoryTableBase*, which is the address of the beginning of the page table directory of the process. This table is used for virtual address to physical address translation and is detailed in the corresponding section. Fields *ReadyListHead*, *ThreadListHead*, *SwapListEntry* are the linked list of process threads that are in ready state, all the threads of this process and threads whose context are being swaped respectively. Each entity in this list is a thread object. A thread object in Windows is of type _ETHREAD and is discussed in the following section.

2.3.7 _ETHREAD Structure

The `_ETHREAD` structure is the body of the thread object. This means that this structure is preceded by an object header of type **`Thread`**. This structure can be used to find information about the threads that are created with a process. For a complete listing of this structure, refer to Appendix 1. The forensic relevant fields of this structure are described below:

- Field *Tcb* is called Thread Control Block (TCB) and is of type _KTHREAD. This structure is detailed later.
- Field *CreateTime* is the creation time of the thread.
- Field *ExitTime* is the exit time of the thread.
- Field *ExitStatus* is the exit status of the thread.
- Field *Cid* contains the process ID and the thread ID of this thread.
- Field *ImpersonationInfo* contains the thread impersonation information and is discussed in detail in the next chapter.
- Field *ThreadsProcess* points to the _EPROCESS block of the process that created the thread.
- Field *ThreadListEntry* is a linked list that links all the threads of a process together.
- Field *CrossThreadFlags* flag represents the state of the thread.

2.3.8 _KTHREAD Structure

_KTHREAD structure is the kernel thread object. This structure mostly contains information that are needed by kernel to manage the thread scheduling. For a complete listing of this structure, refer to Appendix 1. There are some fields in this structure that are of forensic value. These fields are described below:

- Field *Header*: This is the dispatch header that is used for synchronizing access to the thread object. As discussed before, the constant values in this structure can be used to define a memory layout pattern for this object. To locate the lost or hidden threads inside the memory image, the memory image can be searched looking for pieces of memory that resemble this pattern. This technique is detailed in the next chapter.

- Field *ThreadListEntry, WaitListEntry, SwapListEntry*: These fields are used to create a linked list of all process threads, threads that are in waiting state and threads whose context is being swaped out respectively.

- Field *ServiceTable* This is the beginning address of the System Service Table (SST) for this thread. The System Service Table is a table that has the addresses of Windows kernel services. As discussed before, user-land processes do not directly call the native operating system services. Different subsystems and DLLs wrap these services with functions that are called by user-land processes. One of these DLLs that host the majority of these native services is *ntdll.dll*. The DLL *ntdll.dll* exports two sets of functions that are mostly wrappers for services inside the kernel and start with NT or ZW. Except for the functions that are handled inside *ntdll.dll* such as `NtCurrentTeb(...)`, which performs a purely user-land operation, *ntdll.dll* exported functions are routed to a function with the same name in *ntoskrnl.exe*

[31]. The routing mechanism that is performed by the system consists of switching the CPU from user mode to kernel mode, locating the service inside the kernel, copying the function parameters from user-land stack to kernel-land stack and executing the related service inside the kernel. In order to locate a service inside the kernel, Windows uses the service descriptor table, that is located at the address of symbol `KeServiceDescriptorTable`. This table is a structure consisting of four members of type `_SYSTEM_SERVICE_TABLE` [31]. The first system call table is for *ntoskrnl.exe* services and the second one is for *win32k.exe*. The details of structure `_SYSTEM_SERVICE_TABLE` are shown below:

```
struct _SYSTEM_SERVICE_TABLE
{
    PDWORD ServiceTable;
    PDWORD CounterTable;
    DWORD ServiceLimit ;
    PBYTE ArgumentTable;
)
```

Field *ServiceTable* is the address of an array of the beginning address of each service. Field *ArgumentTable* is an array that stores the number of argument bytes for the corresponding service in the array that is pointed by field *ServiceTable*. For example, the first service in the array that is pointed by field *ServiceTable* for *ntoskrnl.exe* is *NtAcceptConnectPort*, which is at the address of 0x805a3104 and takes six arguments that together take up 44 (0x2c) bytes on the thread stack (Figure 2.6).

- Field *KernelTime* is the amount of time that the thread was executing in kernel-

The ArgumentTable member of the system service table.

```
kd> dd KeServiceDescriptorTable
8055b6e0  80503940 00000000 0000011c 80503db4
8055b6f0  00000000 00000000 00000000 00000000
8055b700  00000000 00000000 00000000 00000000
8055b710  00000000
8055b720  00000002
8055b730  f7b1fa80 f678d9e0 86dbc0f4 806f4040
8055b740  00000000 00000000 fffd9da6 ffffffff
8055b750  603f95e6 01c7c019 00000000 00000000
```

The SYSTEM_SERVICE_STRUCTURE for ntoskrnl.exe

The ServiceTable member of system service table for ntoskrnl.exe. ServiceTable points to the table that has the beginning address of each of the kernel services in memory.

```
kd> dd 80503940
80503940  805a3104
80503950  805a2c06 805ef3e6 805f2c4a 805f2c8e
80503960  80613b9a 806148dc 805ea72e 805ea386
80503970  805d33c2
80503980  806137dc
80503990  80500db4 806148ce 80575974 80537e22
805039a0  8060cde4 805baf64 805f3106 80621cd6
805039b0  805f75f8 805a37f2 80621f2a 805a30a4
```

The beginning address of NtAcceptConnectPort function

```
kd> u 805a3104
nt!NtAcceptConnectPort:
805a3104 689c000000      push    0x9c
805a3109 68309b4d80      push    0x804d9b30
805a310e e8bd7cf9ff      call    nt!_SEH_prolog (8053add0)
805a3113 64a124010000    mov     eax,fs:[00000124]
805a3119 8a8040010000    mov     al,[eax+0x140]
805a311f 884590          mov     [ebp-0x70],al
805a3122 84c0            test    al,al
805a3124 0f84b9010000    je      nt!NtAcceptConnectPort+0x1df
```

```
kd> dd 80503db4
80503db4  2c2c2018
80503dc4  08081810
80503dd4  140c1008 0c102c0c 10201c0c 20141038
80503de4  141c2424 34102010 080c0814 04040404
80503df4  0428080c 1808181c 1808180c 040c080c
80503e04  100c0010 10080828 0c08041c 00081004
80503e14  0c080408 10040828 0c0c0404 28240428
80503e24  0c0c0c30 0c0c0c18 0c10300c 0c0c0c10
```

Number of bytes passed to NtAcceptConnectPort fucntion as arguments on the stack.

Figure 2.6: The service descriptor table contains the address of kernel services.

land.

- Field *UserTime* is the amount of time the thread was executing in user-land.

- Stack Information: One important information that can be found inside this structure is the stack information of the thread. Fields *InitialStack*, *StackLimit* and *KernelStack* contain the stack base, the largest address that the stack can extend to, and the current position of the stack pointer respectively. Field *teb* in this structure points to a structure of type _TEB. This structure is allocated in user-land. Aside from a pointer to the Process Environment Block of the process that created this thread, the only forensically pertinent information that this structure contains is the user-land stack information. The first field in this structure is of type _NT_TIB. Fields *StackBase* and *StackLimit* in this structure contain the stack base and the largest address the stack can extend to respectively. Structure _NT_TIB is shown below:

```
kd> dt _NT_TIB
   +0x000 ExceptionList     : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase         : Ptr32 Void
   +0x008 StackLimit        : Ptr32 Void
   +0x00c SubSystemTib      : Ptr32 Void
   +0x010 FiberData         : Ptr32 Void
   +0x010 Version           : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self              : Ptr32 _NT_TIB
```

Knowing the content of the stack of the thread can be useful to recover some of the activities of the thread during the incident. This technique is the basis of

our approach for discovering the execution path and is detailed in chapter five. In Windows, each user application thread has at least two stacks, one in user-land and one in kernel-land. Accordingly, all functions that are called during the execution of the thread have a stack frame either in kernel land stack or the thread's user land stack depending on the mode in which they are executing. A stack frame contains information that needs to be saved during calling and returning of a function. This information includes, the old value of Base Pointer Register (EBP), the return address, function arguments, local variables, etc. Figure 2.7 shows a stack frame on the stack after function *caller* calls function *callee* at address of n.

As it is shown in Figure 2.7, field `OLD_EBP` on the stack holds the address of the previous frame's `OLD_EBP`. This way, stack frames are chained together and by following this chain each stack frame can be correctly identified. However, some compilers tend to use the EBP pointer within the function as a general purpose register. While this can optimize register utilization, it makes it impossible to trace back the stack by following the EBP chain.

Another technique for identifying the boundaries of a stack frame is to look for return addresses that points to right after a call instruction. In this technique, the stack is traversed word by word testing which address is pointing to an instruction after a call instruction. Using these two techniques, some of the functions called by the program as well as the arguments passed to these functions can be retrieved. Many functions receive as their arguments pointers to data objects that are of forensics importance. For example, consider a program that checks if a string that is entered by the user is the right password by comparing it to the stored password string using the CompareString function. The pointer to the correct password string that is kept in some of the program unknown data structures is

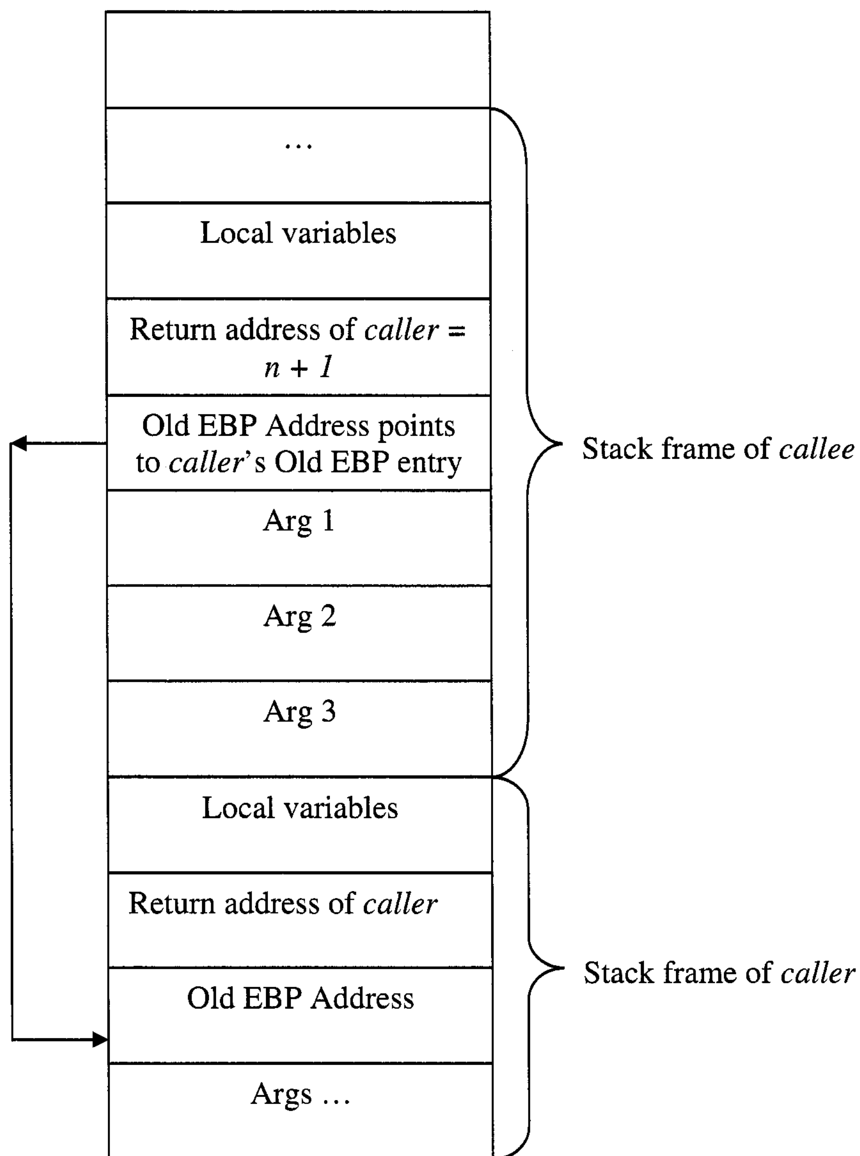


Figure 2.7: The stack frame compositions when function *caller* calls function *callee* at the address of n

passed to this function and therefore is stored on the stack. If the stack frame is not overwritten by the later calls, by tracing this pointer the forensic investigator can retrieve the right password.

Another example is when a program calls one of the services of ntdll.dll. A call to *ntdll.dll* functions loads a service number, which is an index into the arrays kept by the service descriptor table, and executes the *sysenter* command [17]. This command changes the execution mode from user-land to the kernel-land and calls *kiFastSystemCallEntry*. This function locates the service address and jumps to the beginning of it. Figure 2.3.8, shows the thread's stack during the execution of a kernel service.

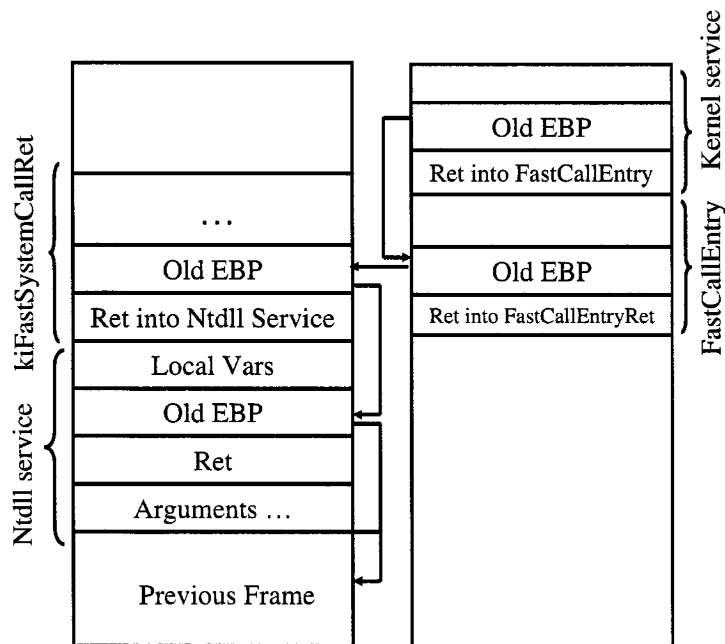


Figure 2.8: Windows thread stacks during the execution of a kernel service

On returning from a kernel service, command *sysexit* is called. This command

switches the execution mode from kernel mode to user mode and jumps to the *KiFastSystemCallRet* that simply returns from the user land *ntdll.dll* function call to the caller. The provided kernel services can also be called by the drivers and other kernel modules. In this case, the calling is performed either in the context of a user application thread that has requested a service from the driver or in the context of system thread if the driver creates its own execution thread by calling *PsCreateSystemThread.*

Now suppose that a rootkit on the system tries to inject a DLL into the memory of an arbitrary process every time that the system starts up by opening a handle to the process using *OpenProcess* and then creating a remote thread inside the process. Function *OpenProcess* receives as one of its arguments the process ID of the process to be opened. Having this information, the investigator will be able to identify the victim process at the time of incident.

In this chapter, we discussed the overall architecture of Windows operating system and detailed on object manager and process manager of Windows as they relate to digital forensics. Different structures and data items that they store and are forensically valuable were detailed. In the next chapter, we follow this discussion by detailing other kernel components that store information that can be forensically valuable. These components include Windows memory manager, security manager and cache manager.

Chapter 3

Memory, Security and Cache Management

In the previous chapters the overall architecture of Windows operating system as it is related to forensic investigation was discussed. Furthermore, Windows object management and possible information that could be acquired from certain Windows objects that are of forensic value were elaborated. This chapter details other components of Windows operating system as they are related to digital investigation. These components are memory manager, cache manager and security manager.

3.1 Memory Manager

Memory manager is part of the Windows executive and therefore exists in the file *Ntoskrnl.exe*. Memory manager is responsible for memory allocation and deallocation operations, managing virtual memory, memory status management, process address space management, disk and memory consistency maintenance and process address space sharing and protection. As a forensic analyst you don't need to know all the

details of how each of these operations are performed by the operating system. However, a complete knowledge about virtual memory and memory page managements and states is necessary in order to extract as much evidence from the memory as possible. In the following sections, first the concept of virtual memory and how it is implemented in Windows is detailed. A discussion will be followed on the manual procedure for virtual address translation and finally as a sample application and an important source of evidence, Windows file management and caching system is described to show the extraction of the content of files that have been copied into memory.

3.1.1 Virtual Memory

Windows uses virtual memory to manage memory operations of processes and operating system components. Physical memory is divided into equal size units called page frames. Each process is assigned a specific amount of virtual memory (4GB in Windows) and all it knows is this virtual address space. This memory is called virtual due to the fact that the addresses are not necessarily mapped to the same address on the physical memory. Moreover, in order to support the 4GB virtual address space, Windows utilizes some part of disk storage to keep the data of the running processes and operating system. This part of the disk is called paging file or swap file and in Windows is represented as *pagefile.sys* on disk. This file is not accessible through the explorer program. However, after taking an image from the disk, it can be analyzed by available tools. It is important to notice that CPU operations can only use data that resides on the Random Access Memory (RAM) and therefore, if the data that is stored in the paging file is required, this data should be brought back into the memory. If there is no free page frame in the physical memory, Windows pages out some pages of memory to the paging file and replaces the page frame's content with the content of the page to be accessed.

Each virtual address is mapped to its corresponding physical address using a procedure known as virtual to physical memory translation. The virtual memory management works at the page granularity. This means that the virtual memory is mapped to physical memory on per page basis. The operating system keeps the records of the mapping between virtual pages to page frames in structures known as page tables. To be more precise, there exist two types of these tables: page directory table and page table. Through the use of these two tables, Windows implements the virtual to physical memory address translation. In the Physical Address Extension(PAE) mode, one more level is added in order to support a bigger virtual address space. This mode of operation is detailed later in this section. The entries in the first page are called Page Directory Entry (PDE). Each PDE points to the beginning of a page table. The page table in turn consists of Page Table Entries (PTE). A PTE has the beginning address of a page frame (physical address). A virtual address is divided into three parts (or four parts in PAE mode). The first part is an index to the page directory table. Using this index, Windows finds the page table that contains the PTE that describes the virtual address. The second part of the virtual address is an index into this table. The PTE that this index points to, contains the page frame number that the desired physical address is part of. After finding which page frame in the physical memory contains the physical address, the third part of the virtual address is used as an offset to this page. The desired virtual address in fact describes the same address in the physical memory as the physical address that is found by adding the beginning of the page frame to the third part of the virtual address.

In the default virtual address management of Windows, each process has 4 GB of virtual memory. This is because the virtual address in Windows is 32 bits. Therefore, 2^{32} different virtual addresses are addressable. In order to support a bigger virtual address

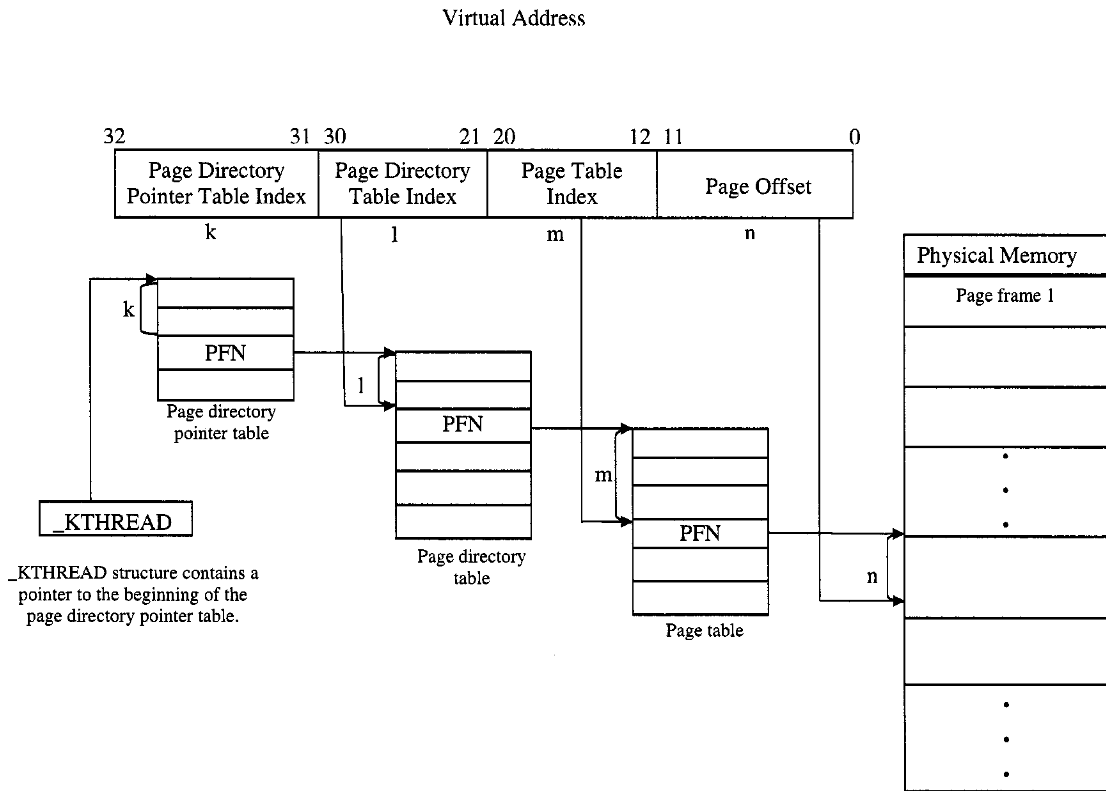
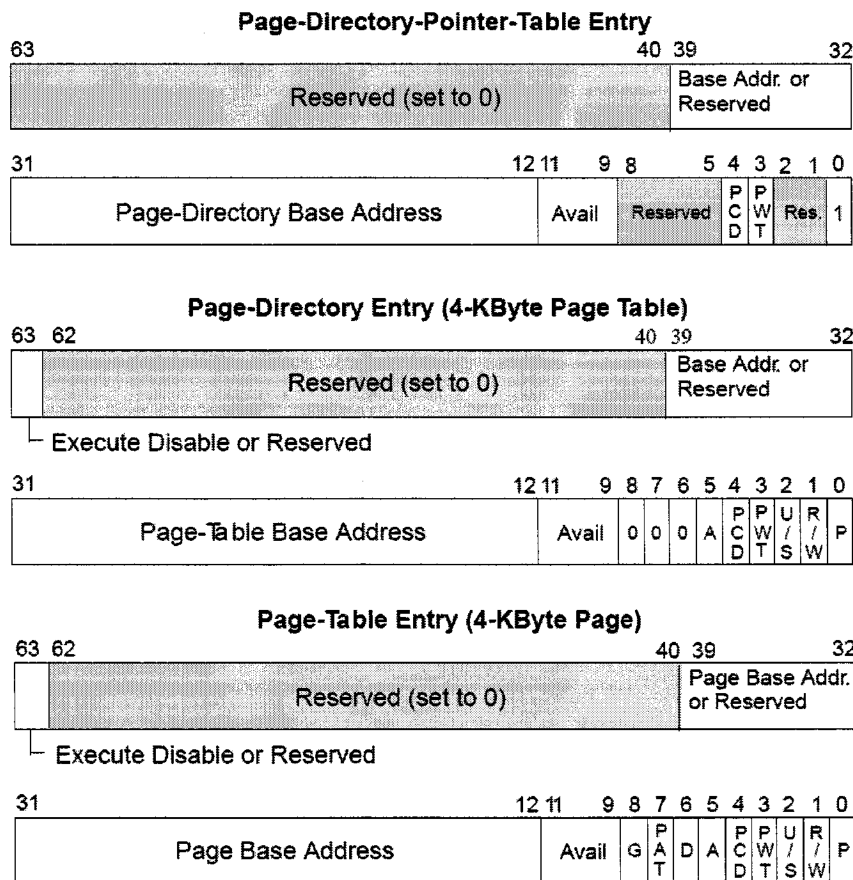


Figure 3.1: Overall address translation in PAE mode. [29]

space, the Intel x86 Pentium Pro processor introduced a memory-mapping mode called Physical Address Extension (PAE). The PAE mode allows accessing of up to 64GB of physical memory on a 32 bit Windows running on current Intel x86 processors. In the PAE mode, the virtual address is divided into four fields instead of three. However, the support for the extended virtual address space rise from the fact that in PAE mode PDEs and PTEs are 64 bits long instead of 32 bits in non-PAE mode of operation. A schematic view of the whole address translation process in PAE mode is shown in Figure 3.1.

The PDE and PTE are 8 bytes long and have the specification shown in figure



h

Figure 3.2: Format of PDE and PTE in PAE mode.

3.2.

In PAE mode, the first level of address translation selects which page directory pointer is pointing to the page directory for the virtual address. This page directory pointers are stored in a table that is pointed by filed *DirectoryTableBase* in `_KPROCESS` structure of the process as discussed in the previous chapter. There are four page directory pointer entries in this table.

To understand how the address translation in PAE mode works, let us follow the

translation procedure by an example. Suppose that we want to translate the virtual address of `0xc2e61940`. For this address, the forth page directory pointer entry will be chosen since the two higher order bits are 11.

As it is shown below, the directory base (*DirBase*) is located at 07600820.

```
kd> !process PROCESS 870eecd8  SessionId: 0  Cid: 05d8    Peb:
7ffd6000  ParentCid: 0c68
    DirBase: 07600820  ObjectTable: e4003948  HandleCount:   0.
    Image: windbg.exe
    ....
```

Therefore, the Page directory pointer entry is located at the physical address of `0x07600820 + 0x18(8*3) = 0x7600838`. Below is the content of the page directory pointer entry:

```
kd> !dd 7600838
# 7600838 0da6b801 00000000 f7b5cc00 00000000
# 7600848 1284f801 00000000 117d0801 00000000
```

The bit 31 - 12 of the first 4 bytes contains the address of the page directory which is `0x0da6b000` and hence, the address of the page directory entry that contains the address of the page table is `PDT_ADDR + PDE_INDEX * SIZEOF_PDE = 0x0da6b000 + 0x17 * 8 = 0x0da6b0b8`.

```
kd> !dd 0da6b0b8
# da6b0b8 073f1963 00000000 073f2963 00000000
# da6b0c8 073f3963 00000000 073f4963 00000000
```

Page Frame Number	Reserved	Reserved	Reserved	Global	Large Page	Dirty	Accessed	Cache Disabled	Write Through	Kernel/ User	Read/ Write	Valid/ Invalid
	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3.3: Field description of PDE and PTE.

Again according to the format of the PDE, the bits 31 - 12 of the first for bytes of the PDE contains the address of the page table which is 0x073f1000 and therefore the address of the page table entry containing the address of the physical page is PTT_ADDR + PTE_INDEX * SIZEOF_PTE = 0x073f1000 + 0x61 * 8 = 0x073f1308.

```
kd> !dd 073f1308
# 73f1308 11df3921 00000000 00000400 e1b11510
# 73f1318 00000400 e1b11518 00000400 e1b11520
```

In the same way the page frame number containing the address would be 0x11df3 and therefore the physical address of 0xc2e61940 will be 0x11df3940. This can be verified as below:

```
kd> dd c2e61940
c2e61940  04e44029 01c6e382 04e44029 01c6e382
...

kd> !dd 11df3940
#11df3940 04e44029 01c6e382 04e44029 01c6e382
...
```

Figure 3.1.1 is the definition of other fields in the PDE and PTE.

- Flag *Accessed*: Page was read before.

- Flag *Cache disabled*: The page should not be cached.

- Flag *Dirty*: The page is dirty meaning that it was written to.

- Flag *Global*: For multi-processors systems.

- Flag *Large page*: The PDE describes a large page. A large page has the size of 4MB in default mode and 2MB in PAE mode. When this flag is enabled, the PDE does not point to a page table anymore and it has the PFN of the large page containing data.

- Flag *Kernel/User*: Whether the page is kernel mode or user mode. If the page is in kernel mode, then it can only accessed from the kernel land.

- Flag *Valid/Invalid*: The page exists in the physical memory. However, if this field is 0, the page might be still in memory. This happens when the page is in transition state or the PTE or PDE points to a prototype PTE. These situations are discussed in more details later.

- Flag *Write through*: The write operation caching is disabled. To improve the disk write operations, Windows caches the writes to a file for a certain amount of time and writes all of the changes to the disk at once. If this bit is set, the writes to a page is flushed to the disk as soon as they are executed.

- Flag *Write*: The page is writable or only read-only.

- Flag *prototype*: The PTE or the PDE points to a prototype PTE. Prototype PTEs are discussed later in this section.

- Flag *Transition*: The page is in transition state. Page states are discussed shortly.

3.1.2 Page Frame Database

Aside from the page tables that keep track of the states of pages, Windows maintains a database of the information regarding the current state of each page frame of the physical memory. This database is called Page Frame Number database and is stored at the address pointed by *MmPfnDatabase* kernel symbol. For each page frame, the status of the page frame can be in one of the following states:

- *Active*: The page is pointed by a PTE. The page is said to be in active or valid state. When a page is in valid state, it can be part of the working set of a process, system, or non-paged part of the kernel. The working set of the process is the part of the process address space that is currently stored in the physical memory rather than the paging file. When a page frame is in active state, the corresponding PTE has its *valid* bit set. In a 32-bit x86 system, the index of the page frame is stored in the PTE in bits 31-12. As mentioned before, this index should be multiplied by the page size to acquire the beginning address of the page frame to which the virtual address is mapped.

- *Transition*: When a page is in transition state, it is not part of any workspace. However, the corresponding PTE of the process that the page has been previously part of its workspace still points to the page frame. A page is in transition state when an I/O operation on the page is still in progress. When a page frame is in transition state, the corresponding PTE has its *transition* bit set and the *prototype* bit unset.

- *Standby*: A page is in standby state when it was previously part of a working set and was removed later. Moreover, the page has not been changed since the last time that it was read from or written to the disk. When a page is in this state,

the corresponding PTE marks the page as invalid but the page frame number in the PTE still points to the right page frame. In this state, the *transition* bit in the PTE is set.

- *Modified*: A page is in modified state when it has been previously part of a working set and it has been modified. However, the updated content of it hasn't been written to the disk. The page frame number in the corresponding PTE points to the right page frame with *dirty* and *transition* bits set and *valid* bit unset.
- *Modified no-write*: A page in this state has been modified previously without its content being written to disk. However, pages in this state will not be written to disk. Drivers can use this page state to defer the writing of the modified pages to the disk to a proper time. For example, NTFS drivers use this state of the page to implement the journaling.
- *Free*: The page frame is free but it contains some data left from the previous allocation of it. For security purposes, a page in this state should be zeroed out before it can be allocated to a process.
- *Zeroed*: The page frame is zeroed out by the operating system and is ready to be allocated to a process.
- *Rom*: The page frame content has been brought to memory from the read-only memory (ROM).
- *Bad*: The page frame is not accessible due to hardware or parity fault.

The corresponding entry of each page frame in the page frame number database describes the status of the page frame. Except for page frames in active states and bad states, the rest of the entries in the page frame number database are part of one linked

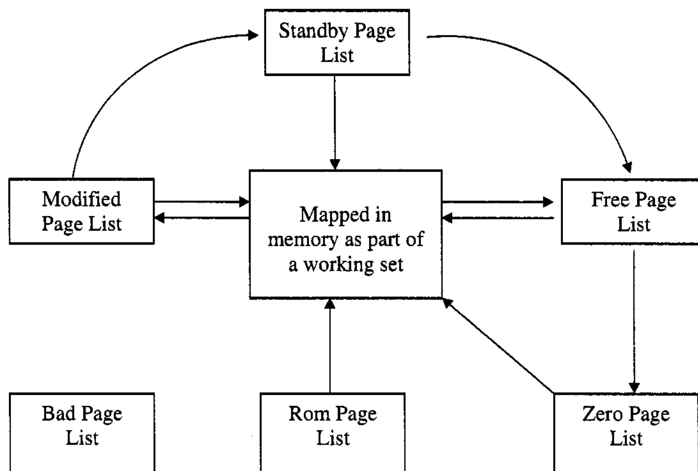


Figure 3.4: The state transition diagram of a page frame in Windows.

list of entries of the same type. Therefore, there exist six linked lists that link entries of the page frame number database together. These linked lists are zeroed, free, standby, modified, rom, and modified no-write. The page frame number entry is linked in one of these lists based on its status as discussed above.

As the system continues operating, the status of the page frames are changed. Figure 3.4 shows the state transition diagram for the states of a page frame. According to the diagram, a page frame is allocated from the zero page list. After the page frame becomes invalid, depending on whether or not it was changed and should be written to disk, its state will change to one of the free, standby, modified, or modified no-write states. After a page is moved from standby state to free, it will stay there until the operating system is in need of more zero pages in which case it will be zeroed and will be placed in the zero list. From forensic stand point, all page frames in states other than zeroed could contain relevant evidence and should be analyzed.

All the entries in the page frame number database are of the same length. However, depending on the state of a page frame, its corresponding entry has different internal structure. A PFN database entry is of type _MMPFN. This structure is detailed below:

```
kd> dt -r _MMPFN
   +0x000 u1                    : __unnamed
      +0x000 Flink                 : Uint4B
      +0x000 WsIndex               : Uint4B
      +0x000 Event                 : Ptr32 _KEVENT
         +0x000 Header                : _DISPATCHER_HEADER
      +0x000 ReadStatus            : Int4B
      +0x000 NextStackPfn          : _SINGLE_LIST_ENTRY
         +0x000 Next                  : Ptr32 _SINGLE_LIST_ENTRY
   +0x004 PteAddress            : Ptr32 _MMPTE
      +0x000 u                     : __unnamed
         +0x000 Long                  : Uint8B
         +0x000 HighLow               : _MMPTE_HIGHLOW
         +0x000 Hard                  : _MMPTE_HARDWARE
         +0x000 Flush                 : _HARDWARE_PTE
         +0x000 Proto                 : _MMPTE_PROTOTYPE
         +0x000 Soft                  : _MMPTE_SOFTWARE
         +0x000 Trans                 : _MMPTE_TRANSITION
         +0x000 Subsect               : _MMPTE_SUBSECTION
         +0x000 List                  : _MMPTE_LIST
   +0x008 u2                    : __unnamed
      +0x000 Blink                 : Uint4B
```

```
      +0x000 ShareCount       : Uint4B
   +0x00c u3                  : __unnamed
      +0x000 e1               : _MMPFNENTRY
         +0x000 Modified         : Pos 0, 1 Bit
         +0x000 ReadInProgress   : Pos 1, 1 Bit
         +0x000 WriteInProgress  : Pos 2, 1 Bit
         +0x000 PrototypePte     : Pos 3, 1 Bit
         +0x000 PageColor        : Pos 4, 3 Bits
         +0x000 ParityError      : Pos 7, 1 Bit
         +0x000 PageLocation     : Pos 8, 3 Bits
         +0x000 RemovalRequested : Pos 11, 1 Bit
         +0x000 CacheAttribute   : Pos 12, 2 Bits
         +0x000 Rom              : Pos 14, 1 Bit
         +0x000 LockCharged      : Pos 15, 1 Bit
         +0x000 DontUse          : Pos 16, 16 Bits
      +0x000 e2               : __unnamed
         +0x000 ShortFlags       : Uint2B
         +0x002 ReferenceCount   : Uint2B
   +0x010 OriginalPte         : _MMPTE
      +0x000 u                : __unnamed
         +0x000 Long             : Uint8B
         +0x000 HighLow          : _MMPTE_HIGHLOW
         +0x000 Hard             : _MMPTE_HARDWARE
         +0x000 Flush            : _HARDWARE_PTE
         +0x000 Proto            : _MMPTE_PROTOTYPE
```

```
        +0x000 Soft             : _MMPTE_SOFTWARE
        +0x000 Trans            : _MMPTE_TRANSITION
        +0x000 Subsect          : _MMPTE_SUBSECTION
        +0x000 List             : _MMPTE_LIST
  +0x018 u4             : __unnamed
     +0x000 EntireFrame      : Uint4B
     +0x000 PteFrame         : Pos 0, 26 Bits
     +0x000 InPageError      : Pos 26, 1 Bit
     +0x000 VerifierAllocation : Pos 27, 1 Bit
     +0x000 AweAllocation    : Pos 28, 1 Bit
     +0x000 LockCharged      : Pos 29, 1 Bit
     +0x000 KernelStack      : Pos 30, 1 Bit
     +0x000 Reserved         : Pos 31, 1 Bit
```

Notice the use of the `-r` option for `dt` command. This option directs windbg to recursively traverse the structure members and print the details of each structure. As you can see several fields in _MMPTE can have different meanings. Basically, there exist four types of PFN database entry structures. Each of these types are discussed hereafter. The first type is for active frames. This structure is shown below:

```
0x000 WsIndex          : Uint4B
0x004 PteAddress       : Ptr32 _MMPTE
0x008 ShareCount       : Uint4B
0x00c Flags            : Uint2B
         +0x000 Modified         : Pos 0, 1 Bit
         +0x000 ReadInProgress   : Pos 1, 1 Bit
         +0x000 WriteInProgress  : Pos 2, 1 Bit
```

```
      +0x000 PrototypePte      : Pos 3, 1 Bit
      +0x000 PageColor         : Pos 4, 3 Bits
      +0x000 ParityError       : Pos 7, 1 Bit
      +0x000 PageLocation      : Pos 8, 3 Bits
      +0x000 RemovalRequested : Pos 11, 1 Bit
      +0x000 CacheAttribute    : Pos 12, 2 Bits
      +0x000 Rom               : Pos 14, 1 Bit
      +0x000 LockCharged       : Pos 15, 1 Bit
      +0x000 DontUse           : Pos 16, 16 Bits

0x00e ReferenceCount   : Uint2B
0x010 OriginalPte      : _MMPTE
0x018 EntireFrame      : Uint4B
```

Field *PteAddress* in this structure contains the virtual address of the PTE that points to this page frame. Field *ShareCount* is the number of PTEs that refer to this page frame. Field *ReferenceCount* is the number of references to this page frame. When a page frame is first mapped to the working set of a process or system, or a device diver, it is incremented and when it is deallocated, this counter is decremented. The difference between fields *ShareCount* and *ReferenceCount* is that when the *ReferenceCount* is zero, the page can be removed from the active state to one of free, standby, or modified list. However, when *ShareCount* is zero, the page might still stay in active state since there might be other references to it. Moreover, as said before, *ShareCount* is increased every time that a process maps the page frame as part of its working set. However, *ReferenceCount* is increased only the first time that the page frame is mapped to a working set. Therefore, when a page frame has the *ReferenceCount* of one or more,

it is active and when it is 0, then depending on the value of *Flags*, it can be in the zeroed, modified, modified no-write free, bad, ROM or standby state. Field *OriginalPte* contains the content of the PTE that points this page frame. Field *EntireFrame* is the page frame number of the page table that holds the PTE that points to this frame. As you will see later on, using this field, we can find the virtual address of a physical address.

The second type of PFN database Entry is for pages that are in modified or standby state. This structure is shown below:

```
0x000 Flink             : Uint4B
0x004 PteAddress        : Ptr32 _MMPTE
0x008 Blink             : Uint4B
0x00c Flags             : Uint2B
         +0x000 Modified          : Pos 0, 1 Bit
         +0x000 ReadInProgress    : Pos 1, 1 Bit
         +0x000 WriteInProgress   : Pos 2, 1 Bit
         +0x000 PrototypePte      : Pos 3, 1 Bit
         +0x000 PageColor         : Pos 4, 3 Bits
         +0x000 ParityError       : Pos 7, 1 Bit
         +0x000 PageLocation      : Pos 8, 3 Bits
         +0x000 RemovalRequested : Pos 11, 1 Bit
         +0x000 CacheAttribute    : Pos 12, 2 Bits
         +0x000 Rom               : Pos 14, 1 Bit
         +0x000 LockCharged       : Pos 15, 1 Bit
         +0x000 DontUse           : Pos 16, 16 Bits
```

```
0x00e ReferenceCount   : Uint2B
0x010 OriginalPte      : _MMPTE
0x018 EntireFrame      : Uint4B
```

As you can see, the only difference with the previous structure are the addition of the two pointers *Flink* and *Blink*. Using these fields all of the PFN database entries that have the same state are doubly linked together. As said before, for all of the pages in this state field *ReferenceCount* is zero.

The third type of PFN database Entry is used for pages on the zero or free list. The only difference from the previous structure is that instead of backward link, The structure field at the offset of 0x8 stores a value that is used for cache usage optimization. The fourth type of PFN database Entry is used for a page on which an I/O operation is in progress but the page is no longer active. In this structure, the field at the offset of 0x0 contains the address of the event object that will be notified when the I/O operation finishes.

We have previously discussed the manual procedure for virtual to physical address translation. Using the page frame number database we can reverse this procedure to find the virtual address that is mapped to a physical address. If you remember, field *ENitrFrame* in structure `_MMPFN` contains the page frame number of the page table that contains the PTE pointing to this frame. This field can be used to reverse the virtual to physical address translation process. Let us follow this process by an example. Suppose that we want to find the virtual address that is mapped to the physical address of 0x12466000 in a system that operates in PAE mode. For demonstration, we use !pfn command of windbg to show the content of a page frame number. This command receives the page frame number as input. The output of executing !pfn command with the page frame number of this address is shown below:

```
kd> !pfn 12466
    PFN 00012466 at address 812A5B28
    flink 0000031F  blink / share count 00000002  pteaddress C0710FD8
    reference count 0001   Cached     color 0
    restore pte 000000C0  containing page        000A76  Active     M
    Modified
```

containing page shows the value of *EntireFrame* that is 0xA76. This is the page frame number of a page table that has a PTE pointing to this frame. Therefore, if we search through this page table for a PTE that points to the page frame number of 0x12466, that is our page frame, we can find the index of the PTE in the page table and this index divided by 8 (the size of PTE in PAE mode) is the 9 bits in the virtual address that are used as the index in the third level page table. This process is shown below:

```
kd> !dd 0A76000  11000
#  a76000 00aca163 00000000 00ae6143 00000000
.......
#  a76fd0 12431163 00000000 12466163 00000000
#  a76fe0 129a2163 00000000 1252a143 00000000
#  a76ff0 12599143 00000000 000000c0 000091d9
#  a77000 00000000 00000000 00000000 00000000
#  a77010 00000000 00000000 00000000 00000000
.......
```

As you see, the PTE at the address of `a76fd8` has the value of `12466` as its first 20 bits. Therefore the index of the PTE is `(0xa76fd8 - 0xa76000) = fd8` and therefore, the third level offset in the virtual address is `fd8 / 8 = 1fb`. We can apply the same process with the PFN number of `0xa76` to find the second level offset and then the first

level offset in the virtual address.

```
kd> !pfn A76
    PFN 00000A76 at address 810B84E8
    flink 00000000  blink / share count 000001BC  pteaddress C0603880
    reference count 0001   Cached     color 0
    restore pte 00000080  containing page        00032A  Active     M
    Modified

kd> !dd 32A000 11000
.......
#  32a880 00a76063 00000000 0e52f063 00000000
#  32a890 0d6b7063 00000000 08b4f063 00000000
#  32a8a0 0458b063 00000000 04674063 00000000
.......

(32a880 - 32a000) / 8 = 110


kd> !dd 32A000  11000
.......
#  32a010 00329063 00000000 0032a063 00000000
#  32a020 02dd0063 00000000 00000000 00000000
#  32a030 00000000 00000000 03013163 00000000
#  32a040 0301a163 00000000 03015163 00000000
.......
```

```
(32a018 - 32a000) / 8 = 3
```

When we put the three offsets that we found in a 32 bit number we will have the virtual address as:

```
0x1fb = 111111011

0x110 = 100010000

0x3 = 11


virtual address = 11 100010000 111111011 000000000000 = 0xE21FB000


kd> !pte E21FB000

               VA e21fb000

PDE at 00000000C0603880    PTE at 00000000C0710FD8

contains 0000000000A76063  contains 0000000012466163

pfn a76 ---DA--KWEV    pfn 12466 -G-DA--KWEV
```

As you see, the output of executing !pte command with the virtual address we found, has our starting page frame number of `0x12466`. This command shows the details of virtual to physical address translation.

3.2 File Extraction

When a file is created using Windows I/O functions such as *CreateFile*, Windows creates a file object that represents the file in kernel. The file object is of type `_FILE_OBJECT` and has the following structure:

```
kd> dt _FILE_OBJECT
```

```
+0x000 Type              : Int2B
+0x002 Size              : Int2B
+0x004 DeviceObject      : Ptr32 _DEVICE_OBJECT
+0x008 Vpb               : Ptr32 _VPB
+0x00c FsContext         : Ptr32 Void
+0x010 FsContext2        : Ptr32 Void
+0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap   : Ptr32 Void
+0x01c FinalStatus       : Int4B
+0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
+0x024 LockOperation     : UChar
+0x025 DeletePending     : UChar
+0x026 ReadAccess        : UChar
+0x027 WriteAccess       : UChar
+0x028 DeleteAccess      : UChar
+0x029 SharedRead        : UChar
+0x02a SharedWrite       : UChar
+0x02b SharedDelete      : UChar
+0x02c Flags             : Uint4B
+0x030 FileName          : _UNICODE_STRING
+0x038 CurrentByteOffset : _LARGE_INTEGER
+0x040 Waiters           : Uint4B
+0x044 Busy              : Uint4B
+0x048 LastLock          : Ptr32 Void
+0x04c Lock              : _KEVENT
```

```
   +0x05c Event             : _KEVENT
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

In the structure shown above, field *FileName* contains the name of the file. Fields from offset 0x26 to offset 0x2b describe different access modes defined at the time of creating the file. *PrivateCacheMap* points to the private cache map of the process as discussed later. For every file opened by a process, Windows creates a structure of type `_SECTION_OBJECT_POINTERS` that is pointed by field *SectionObjectPointer* in the file object that represents the opened file. This structure is shown below:

```
kd> dt _SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : Ptr32 Void
   +0x004 SharedCacheMap    : Ptr32 Void
   +0x008 ImageSectionObject : Ptr32 Void
```

Fields *DataSectionObject* and *ImageSectionObject* are pointers to structures of type `_CONTROL_AREA`. Field *DataSectionObject* is used when the file is accessed as a data file and field *ImageSectionObject* is used when the file is accessed as an executable. Field *SharedCacheMap* is a pointer to the shared cache map of the file and is discussed later. The `_CONTROL_AREA` structure is shown below:

```
kd> dt _CONTROL_AREA
   +0x000 Segment           : Ptr32 _SEGMENT
   +0x004 DereferenceList   : _LIST_ENTRY
   +0x00c NumberOfSectionReferences : Uint4B
   +0x010 NumberOfPfnReferences : Uint4B
   +0x014 NumberOfMappedViews : Uint4B
   +0x018 NumberOfSubsections : Uint2B
```

```
+0x01a FlushInProgressCount : Uint2B
+0x01c NumberOfUserReferences : Uint4B
+0x020 u                    : __unnamed
+0x024 FilePointer          : Ptr32 _FILE_OBJECT
+0x028 WaitingForDeletion : Ptr32 _EVENT_COUNTER
+0x02c ModifiedWriteCount : Uint2B
+0x02e NumberOfSystemCacheViews : Uint2B
```

This structure stores information about the mapping of the file to the memory. Field *Segment* is a pointer to a segment object that is of type _SEGMENT_OBJECT. This structure is shown below:

```
kd> dt _SEGMENT_OBJECT
   +0x000 BaseAddress       : Ptr32 Void
   +0x004 TotalNumberOfPtes : Uint4B
   +0x008 SizeOfSegment     : _LARGE_INTEGER
   +0x010 NonExtendedPtes   : Uint4B
   +0x014 ImageCommitment   : Uint4B
   +0x018 ControlArea       : Ptr32 _CONTROL_AREA
   +0x01c Subsection        : Ptr32 _SUBSECTION
   +0x020 LargeControlArea : Ptr32 _LARGE_CONTROL_AREA
   +0x024 MmSectionFlags    : Ptr32 _MMSECTION_FLAGS
   +0x028 MmSubSectionFlags : Ptr32 _MMSUBSECTION_FLAGS
```

Field *Subsection* in this structure points to the end of the control area of the file that is the beginning of the first subsection of the file. A subsection is of type _SUBSECTION and describes information about the physical address at which each section of the file is mapped to memory. This structure is detailed below:

```
kd> dt _SUBSECTION
   +0x000 ControlArea      : Ptr32 _CONTROL_AREA
   +0x004 u                : __unnamed
   +0x008 StartingSector   : Uint4B
   +0x00c NumberOfFullSectors : Uint4B
   +0x010 SubsectionBase   : Ptr32 _MMPTE
   +0x014 UnusedPtes       : Uint4B
   +0x018 PtesInSubsection : Uint4B
   +0x01c NextSubsection   : Ptr32 _SUBSECTION
```

Field *StartingSector* is the first sector of the file that this subsection represents. Field *NumberOfFullSectors* is the number of the sectors that this subsection describes. Field *PtesInSubsection* is the number of Page Table Entries (PTE) that the prototype page table of this subsection contains. The prototype page table, that field *Subsection-Base*points to, is a table that consist of prototype page table entries. If you remember one of the flags in a PTE is called *Ptototype* and specifies whether or not the PTE points to a prototype PTE. Prototype page tables are used to enable sharing of a page of memory. The idea is to keep just one copy of a file inside the memory. So if more than one process has mapped the file, then they all have PTEs pointing to the physical addresses to which the pages are mapped. Now suppose that a shared page of memory is swapped out to the paging file and then brought back in to the memory. In such situation, memory manager should keep track of all PTEs that point to this page and update all of them. However, instead of keeping another database for all the frames inside the paging file, memory manager uses the prototype page tables. When the file is first created or a shared page of memory is allocated, memory manager creates a table of prototype page table entries that is pointed by a segment object. When a process

accesses this page for the first time, the corresponding PTE in its page table is filled in from the information in its prototype PTE. If the page is no longer used, the memory manager swaps out the page from the memory and makes all PTEs point to the prototype PTE. When a page is faulted in later, this time memory manager only updates the corresponding prototype page table entry instead of all PTEs since the rest of the PTEs are pointing to the updated prototype PTE. Like regular PTEs, a prototype PTE can be in one of the following states:

- Active
- Transition
- Modified-no-write
- Demand Zero
- Page file
- Mapped file

The description of these states is the same as their corresponding states for page tables and are not repeated. Based on the above discussion, 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. This process is shown below:

```
kd> dt _FILE_OBJECT 0x81F32810
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x82be1738 _DEVICE_OBJECT
   +0x008 Vpb              : 0x82b8d2e8 _VPB
```

```
+0x00c FsContext          : 0xe21487f8
+0x010 FsContext2         : 0xe2148950
+0x014 SectionObjectPointer : 0x8286c22c _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap    : (null)
+0x01c FinalStatus        : 0
+0x020 RelatedFileObject : (null)
+0x024 LockOperation      : 0 ''
+0x025 DeletePending      : 0 ''
+0x026 ReadAccess         : 0x1 ''
+0x027 WriteAccess        : 0x1 ''
+0x028 DeleteAccess       : 0 ''
+0x029 SharedRead         : 0x1 ''
+0x02a SharedWrite        : 0x1 ''
+0x02b SharedDelete       : 0 ''
+0x02c Flags              : 0x140042
+0x030 FileName           : _UNICODE_STRING "\Documents and Settings\
                           bestbuy\Local Settings\Temporary Internet
                           Files\Content.IE5\index.dat"
+0x038 CurrentByteOffset : _LARGE_INTEGER 0x0
+0x040 Waiters            : 0
+0x044 Busy               : 0
+0x048 LastLock           : (null)
+0x04c Lock               : _KEVENT
+0x05c Event              : _KEVENT
+0x06c CompletionContext : (null)
```

```
kd> dt _SECTION_OBJECT_POINTERS 0x8286c22c
   +0x000 DataSectionObject : 0x82b5e898
   +0x004 SharedCacheMap    : (null)
   +0x008 ImageSectionObject : (null)


kd> dt _CONTROL_AREA 0x82b5e898
   +0x000 Segment           : 0xe10963e0 _SEGMENT
   +0x004 DereferenceList   : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x00c NumberOfSectionReferences : 1
   +0x010 NumberOfPfnReferences : 0x3d
   +0x014 NumberOfMappedViews : 8
   +0x018 NumberOfSubsections : 2
   +0x01a FlushInProgressCount : 0
   +0x01c NumberOfUserReferences : 9
   +0x020 u                 : __unnamed
   +0x024 FilePointer       : 0x828b8be8 _FILE_OBJECT
   +0x028 WaitingForDeletion : (null)
   +0x02c ModifiedWriteCount : 0
   +0x02e NumberOfSystemCacheViews : 0


kd> dt _SEGMENT_OBJECT 0xe10963e0
```

```
   +0x000 BaseAddress      : 0x82b5e898
   +0x004 TotalNumberOfPtes : 0x940
   +0x008 SizeOfSegment    : _LARGE_INTEGER 0x9`00000920
   +0x010 NonExtendedPtes  : 0x940000
   +0x014 ImageCommitment  : 0
   +0x018 ControlArea      : 0x000004c0 _CONTROL_AREA
   +0x01c Subsection       : 0x82b5e8c8 _SUBSECTION
   +0x020 LargeControlArea : (null)
   +0x024 MmSectionFlags   : (null)
   +0x028 MmSubSectionFlags : (null)

kd> dt _SUBSECTION 0x82b5e8c8
   +0x000 ControlArea      : 0x82b5e898 _CONTROL_AREA
   +0x004 u                : __unnamed
   +0x008 StartingSector   : 0
   +0x00c NumberOfFullSectors : 0x920
   +0x010 SubsectionBase   : 0xe10a4000 _MMPTE
   +0x014 UnusedPtes       : 0
   +0x018 PtesInSubsection : 0x920
   +0x01c NextSubsection   : 0x81f2c1c8 _SUBSECTION

kd> dd 0xe10a4000
e10a4000  1794f123 00000000 0b07b123 00000000
```

```
e10a4010  1790e123 00000000 06b35123 00000000
e10a4020  0635f123 00000000 10ae4123 00000000
e10a4030  000004c0 82b5e8c8 000004c0 82b5e8c8
e10a4040  000004c0 82b5e8c8 000004c0 82b5e8c8
```

```
kd> !db 1794f000
#1794f000 436c 6965 6e74 2055-726c 4361 6368 6520 Client UrlCache
#1794f010 4d4d 4620 5665 7220-352e 3200 0000 9200 MMF Ver 5.2.....
#1794f020 0050 0000 8023 0100-f626 0000 0000 0000 .P...#...&......
#1794f030 0054 0728 0000 0000-00a0 6c23 0000 0000 .T.(......l#....
#1794f040 0050 a900 0000 0000-1400 0000 4400 0000 .P..........D...
#1794f050 335a 3644 5331 4343-4400 0000 4633 3742 3Z6DS1CCD...F37B
#1794f060 5651 504c 4400 0000-5748 515a 5731 4133 VQPLD...WHQZW1A3
#1794f070 4300 0000 4b44 4d37-4731 5142 4300 0000 C...KDM7G1QBC...
```

To extract process executables, the same technique can be used. However, this time the link to the beginning of the prototype page table entry is stored in a structure of type _SEGMENT, which itself is pointed to by the section object representing the mapped executable file. If you remember, field *SectionObject* in _EPROCESS block of a process points to its section object. The section object is of type _SECTION_OBJECT as shown below:

```
kd> dt _SECTION_OBJECT
   +0x000 StartingVa       : Ptr32 Void
   +0x004 EndingVa         : Ptr32 Void
   +0x008 Parent           : Ptr32 Void
```

```
   +0x00c LeftChild         : Ptr32 Void
   +0x010 RightChild        : Ptr32 Void
   +0x014 Segment           : Ptr32 _SEGMENT_OBJECT
```

Although field *Segment* is declared by windbg to be of type _SEGMENT_OBJECT, it is in fact of type _SEGMENT. This structure is shown below:

```
kd> dt _SEGMENT
   +0x000 ControlArea       : Ptr32 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : Uint4B
   +0x008 NonExtendedPtes   : Uint4B
   +0x00c WritableUserReferences : Uint4B
   +0x010 SizeOfSegment     : Uint8B
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : Uint4B
   +0x024 ExtendInfo        : Ptr32 _MMEXTEND_INFO
   +0x028 SystemImageBase   : Ptr32 Void
   +0x02c BasedAddress      : Ptr32 Void
   +0x030 u1                : __unnamed
   +0x034 u2                : __unnamed
   +0x038 PrototypePte      : Ptr32 _MMPTE
   +0x040 ThePtes           : [1] _MMPTE
```

In this structure, field *PrototypePte* stores the beginning of the prototype page table. As an example, below we have shown this process for extracting an executable file.

```
kd> dt _eprocess f7fac020
ntdll!_EPROCESS
```

```
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER 0x1c7ca32`909a68c4
   +0x078 ExitTime         : _LARGE_INTEGER 0x0
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
    ....
   +0x138 SectionObject    : 0xe1899f90
    ....

kd> dt _SECTION_OBJECT 0xe1899f90
   +0x000 StartingVa       : (null)
   +0x004 EndingVa         : (null)
   +0x008 Parent           : (null)
   +0x00c LeftChild        : (null)
   +0x010 RightChild       : (null)
   +0x014 Segment          : 0xe1896710 _SEGMENT_OBJECT

kd> dt _SEGMENT 0xe1896710
   +0x000 ControlArea      : 0x81f1c008 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : 0x57
   +0x008 NonExtendedPtes  : 0x57
   +0x00c WritableUserReferences : 0
   +0x010 SizeOfSegment    : 0x57000
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : 0
```

```
   +0x024 ExtendInfo       : (null)
   +0x028 SystemImageBase  : (null)
   +0x02c BasedAddress     : 0x01000000
   +0x030 u1               : __unnamed
   +0x034 u2               : __unnamed
   +0x038 PrototypePte     : 0xe1896750 _MMPTE
   +0x040 ThePtes          : [1] _MMPTE

kd> dd 0xe1896750
e1896750  0591b121 80000000 0557c860 00000000
e1896760  16f6d121 00000000 0cd52121 00000000
e1896770  0efb1121 00000000 0558a121 00000000
e1896780  17809121 00000000 09a85121 00000000
e1896790  10114860 00000000 040f9121 00000000
e18967a0  07c32121 00000000 0d161121 00000000
e18967b0  0e344121 00000000 13819121 00000000
e18967c0  0d27e121 00000000 0dd7c121 00000000

kd> !db 0591b000
# 591b000 4d5a 9000 0300 0000-0400 0000 ffff 0000 MZ..............
# 591b010 b800 0000 0000 0000-4000 0000 0000 0000 ........@.......
# 591b020 0000 0000 0000 0000-0000 0000 0000 0000 ................
# 591b030 0000 0000 0000 0000-0000 0000 e800 0000 ................
# 591b040 0e1f ba0e 00b4 09cd-21b8 014c cd21 5468 ........!..L.!Th
# 591b050 6973 2070 726f 6772-616d 2063 616e 6e6f is program canno
```

```
# 591b060 7420 6265 2072 756e-2069 6e20 444f 5320 t be run in DOS
# 591b070 6d6f 6465 2e0d 0d0a-2400 0000 0000 0000 mode....$.......
```

Another easier way for extracting the executable file of a process is to find the the executable header of the file and then parse the header to find the virtual address of each section of the file. If you remember, field *ImageBaseAddress* in structure _PEB of the process contains the virtual address at which the file is mapped. Each executable file that is in Portable Executable format starts with a header that describes different properties about the file including loading requirements of the file. Among these information are the sections of the file. Microsoft defines a section in the PE file as the basic unit of code or data within a PE or COFF file. After an executable is loaded in memory, each section of it is mapped in memory and the virtual addresses at which the sections are mapped are stored in the part of the executable header known as section table. Section table of the executable is in fact an array of entries of type _IMAGE_SECTION_HEADER. This structure is shown below:

```
0:040> dt -r _IMAGE_SECTION_HEADER
   +0x000 Name             : [8] UChar
   +0x008 Misc             : __unnamed
      +0x000 PhysicalAddress  : Uint4B
      +0x000 VirtualSize      : Uint4B
   +0x00c VirtualAddress   : Uint4B
   +0x010 SizeOfRawData    : Uint4B
   +0x014 PointerToRawData : Uint4B
   +0x018 PointerToRelocations : Uint4B
   +0x01c PointerToLinenumbers : Uint4B
   +0x020 NumberOfRelocations : Uint2B
```

```
+0x022 NumberOfLinenumbers : Uint2B
+0x024 Characteristics    : Uint4B
```

Field *Name* in this structure is the name of the section. Usually section names start with a dot(.), although it is not a requirement. This field is intended for describing the content of a section and can have any value of size equal or less than 8 bytes. Typical section names names are as follows:

- `.text/.code/CODE/TEXT`: Section contains executable code (machine instructions).
- `.testbss/TEXTBSS`: Is used whens incremental Linking is enabled.
- `.data/.idata/DATA/IDATA`: Section contains initialized data.
- `.bss/BSS`: Section contains uninitialized data.

Field *VirtualSize* in this structure is the size of the section after it is mapped into memory. We will use this field as the number of bytes that should be copied from the memory to extract the section as it is mapped to memory. Field *SizeOfRawData* is the size of the section in the file on the disk. This value could be equal, less than or greater than the value that is stored in field *VirtualSize* depending on the alignment requirements. However, if it is less than the value of field *VirtualSize*, the remainder of the section will be filled out with zero and is not of forensic importance. Field *VirtualAddress* is the address of the start of the section relative to the image base when the section is loaded into memory. We will add this value to the image base of the executable as we have found in the process environment block to acquire the virtual address at which the section is mapped in the memory. Having the size of the section and the virtual address of it, we can extract the content of the section from memory

by first translating the virtual address of each page of the section and then copying the content of the page until we reach the size of the section.

The section table of the executable is located after four other structures. These structures are dos header, PE signature, file header (or COFF header) and optional file header. The optional header can have a variable size and its size is stored in the file 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.

Dos header is of type `_IMAGE_DOS_HEADER`. The details of this structure is shown below:

```
kd> dt _IMAGE_DOS_HEADER
   +0x000 e_magic          : Uint2B
   +0x002 e_cblp           : Uint2B
   +0x004 e_cp             : Uint2B
   +0x006 e_crlc           : Uint2B
   +0x008 e_cparhdr        : Uint2B
   +0x00a e_minalloc       : Uint2B
   +0x00c e_maxalloc       : Uint2B
   +0x00e e_ss             : Uint2B
   +0x010 e_sp             : Uint2B
   +0x012 e_csum           : Uint2B
   +0x014 e_ip             : Uint2B
   +0x016 e_cs             : Uint2B
   +0x018 e_lfarlc         : Uint2B
   +0x01a e_ovno           : Uint2B
```

```
+0x01c e_res            : [4] Uint2B
+0x024 e_oemid          : Uint2B
+0x026 e_oeminfo        : Uint2B
+0x028 e_res2           : [10] Uint2B
+0x03c e_lfanew         : Int4B
```

Field *e_lfanew* which is the last member of this structure at the offset of 0x3c has the offset of a structure that contains PE signature, file header and optional header. This structure is of type _IMAGE_NT_HEADERS. This structure is shown below:

```
kd> dt _IMAGE_NT_HEADERS
   +0x000 Signature        : Uint4B
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

Field *Signature* is the PE signature with the value of 0x4550 that is the hex value for "PE". Field *FileHeader* is the file header and has a member that stores the length of structure *OptionalHeader*. This structure is shown below:

```
kd> dt _IMAGE_FILE_HEADER
   +0x000 Machine          : Uint2B
   +0x002 NumberOfSections : Uint2B
   +0x004 TimeDateStamp    : Uint4B
   +0x008 PointerToSymbolTable : Uint4B
   +0x00c NumberOfSymbols  : Uint4B
   +0x010 SizeOfOptionalHeader : Uint2B
   +0x012 Characteristics  : Uint2B
```

Field *SizeOfOptionalHeader* stores the size of the optional header. We use this value to find the beginning of the section table. Field *NumberOfSections* is the number of the section that exist in this file and therefore the size of the section table.

The extraction process for an executable from the memory based on the above discussion is shown below:

```
kd> !PEB 7ffdf000 PEB at 7ffdf000
    InheritedAddressSpace:     No
    ReadImageFileExecOptions: No
    BeingDebugged:             Yes
    ImageBaseAddress:          00400000
.....


0:022> dd 00400000 + 0x3c 0040003c  00000108 0eba1f0e cd09b400
4c01b821 .....

0:022> dt -r _IMAGE_NT_HEADERS (0x400000 + 0x108)
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
      +0x000 Machine          : 0x14c
      +0x002 NumberOfSections : 3
      +0x004 TimeDateStamp    : 0x44cc1896
      +0x008 PointerToSymbolTable : 0
      +0x00c NumberOfSymbols  : 0
      +0x010 SizeOfOptionalHeader : 0xe0
```

```
      +0x012 Characteristics  : 0x103
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
      +0x000 Magic            : 0x10b
      +0x002 MajorLinkerVersion : 0x8 ''
.....

0:022> dt -r _IMAGE_SECTION_HEADER (0x400000 + 0x108 + 0x18 + 0xe0)
   +0x000 Name             : [8]  ".text"
   +0x008 Misc             : __unnamed
      +0x000 PhysicalAddress  : 0x470491
      +0x000 VirtualSize      : 0x470491
   +0x00c VirtualAddress   : 0x1000
   +0x010 SizeOfRawData    : 0x470600
   +0x014 PointerToRawData : 0x400
   +0x018 PointerToRelocations : 0
   +0x01c PointerToLinenumbers : 0
   +0x020 NumberOfRelocations : 0
   +0x022 NumberOfLinenumbers : 0
   +0x024 Characteristics  : 0x60000020

0:022> dt -r _IMAGE_SECTION_HEADER (0x400000 + 0x108 + 0x18 + 0xe0 +
0x28)
   +0x000 Name             : [8]  ".data"
```

```
   +0x008 Misc              : __unnamed
      +0x000 PhysicalAddress  : 0x7d014
      +0x000 VirtualSize      : 0x7d014
   +0x00c VirtualAddress   : 0x472000
   +0x010 SizeOfRawData    : 0x7bc00
   +0x014 PointerToRawData : 0x470a00
   +0x018 PointerToRelocations : 0
   +0x01c PointerToLinenumbers : 0
   +0x020 NumberOfRelocations : 0
   +0x022 NumberOfLinenumbers : 0
   +0x024 Characteristics  : 0xc0000040


0:022> dt -r _IMAGE_SECTION_HEADER (0x400000 + 0x108 + 0x18 + 0xe0 +
0x28 + 0x28)
   +0x000 Name              : [8]  ".rsrc"
   +0x008 Misc              : __unnamed
      +0x000 PhysicalAddress  : 0x2d3fc
      +0x000 VirtualSize      : 0x2d3fc
   +0x00c VirtualAddress   : 0x4f0000
   +0x010 SizeOfRawData    : 0x2d400
   +0x014 PointerToRawData : 0x4ec600
   +0x018 PointerToRelocations : 0
   +0x01c PointerToLinenumbers : 0
   +0x020 NumberOfRelocations : 0
```

```
+0x022 NumberOfLinenumbers : 0
+0x024 Characteristics   : 0x40000040
```

As you see above, we first find the offset of structure `_IMAGE_NT_HEADERS` by looking at the offset of 0x3c from the image base that is field *e_lfanew* in the DOS header. This field has the value of 0x108. Having the NT headers offset, we add the offset to the image base and consult fields *NumberOfSections* and *SizeOfOptionalHeader* to find the number of sections (three sections) and the size of the optional header (0xe0 bytes). The size of the optional header is added to the address of the optional header to find the beginning of the first section header. This section header is used to find the value of fields *VirtualAddress* and *VirtualSize* of the section. With these values we can extract the section from the memory by copying *VirtualSize* bytes from memory starting at the virtual address of *VirtualAddress.* This process is repeated for the second and third section by adding the size of the section header structure to the beginning of the previous section to find the next section header.

3.2.1 Cache Manager

Unless the file is created with flag `FILE_FLAG_NO_BUFFERING` set, the cache manager caches some parts of the file in memory in order to improve the I/O operation performance. This is where a lot of useful information can be extracted about the content of the files. The file object contains links to structures that are maintained by cache manger and are used to retrieve the content of the file.

Cache manager is a component of Windows that cooperates with memory manager to provide data caching services to other components of Windows. For a file that is created without specifying flag `FILE_FLAG_NO_BUFFERING`, at the first I/O operation on the file, the cache manager creates a shared cache map and a private cache map.

Moreover, 256KB of the file will be mapped to virtual memory. As other regions of the file is accessed, more and more of the file's content will be mapped to memory. The information about accesses to the file and the location where the file is mapped inside the memory are kept in two data structures: private cache map and shared cache map. If you remember, the file object representing a file has links to these two structures.

Each file object has a private cache map of type `_PRIVATE_CACHE_MAP` that keeps the last two addresses inside the file that are accessed by the process. This information will help cache manager in an operation called read-ahead. In this operation, cache manager uses the information stored in the file object private cache to predict the possible future addresses that will be read by the process and thus bring those portions of the file into memory.

The shared cache map is of type `_SHARED_CACHE_MAP` and is where the locations of the file content in memory are stored. These two structures are shown below:

```
kd> dt _PRIVATE_CACHE_MAP
   +0x000 NodeTypeCode      : Int2B
   +0x000 Flags             : _PRIVATE_CACHE_MAP_FLAGS
   +0x000 UlongFlags        : Uint4B
   +0x004 ReadAheadMask     : Uint4B
   +0x008 FileObject        : Ptr32 _FILE_OBJECT
   +0x010 FileOffset1       : _LARGE_INTEGER
   +0x018 BeyondLastByte1   : _LARGE_INTEGER
   +0x020 FileOffset2       : _LARGE_INTEGER
   +0x028 BeyondLastByte2   : _LARGE_INTEGER
   +0x030 ReadAheadOffset   : [2] _LARGE_INTEGER
   +0x040 ReadAheadLength   : [2] Uint4B
```

```
   +0x048 ReadAheadSpinLock : Uint4B
   +0x04c PrivateLinks     : _LIST_ENTRY

kd> dt _SHARED_CACHE_MAP
   +0x000 NodeTypeCode     : Int2B
   +0x002 NodeByteSize     : Int2B
   +0x004 OpenCount        : Uint4B
   +0x008 FileSize         : _LARGE_INTEGER
   +0x010 BcbList          : _LIST_ENTRY
   +0x018 SectionSize      : _LARGE_INTEGER
   +0x020 ValidDataLength  : _LARGE_INTEGER
   +0x028 ValidDataGoal    : _LARGE_INTEGER
   +0x030 InitialVacbs     : [4] Ptr32 _VACB
   +0x040 Vacbs            : Ptr32 Ptr32 _VACB
   +0x044 FileObject       : Ptr32 _FILE_OBJECT
   +0x048 ActiveVacb       : Ptr32 _VACB
   +0x04c NeedToZero       : Ptr32 Void
   +0x050 ActivePage       : Uint4B
   +0x054 NeedToZeroPage   : Uint4B
   +0x058 ActiveVacbSpinLock : Uint4B
   +0x05c VacbActiveCount  : Uint4B
   +0x060 DirtyPages       : Uint4B
   +0x064 SharedCacheMapLinks : _LIST_ENTRY
   +0x06c Flags            : Uint4B
```

```
+0x070 Status            : Int4B
+0x074 Mbcb              : Ptr32 _MBCB
+0x078 Section           : Ptr32 Void
+0x07c CreateEvent       : Ptr32 _KEVENT
+0x080 WaitOnActiveCount : Ptr32 _KEVENT
+0x084 PagesToWrite      : Uint4B
+0x088 BeyondLastFlush   : Int8B
+0x090 Callbacks         : Ptr32 _CACHE_MANAGER_CALLBACKS
+0x094 LazyWriteContext  : Ptr32 Void
+0x098 PrivateList       : _LIST_ENTRY
+0x0a0 LogHandle         : Ptr32 Void
+0x0a4 FlushToLsnRoutine : Ptr32
+0x0a8 DirtyPageThreshold : Uint4B
+0x0ac LazyWritePassCount : Uint4B
+0x0b0 UninitializeEvent : Ptr32 _CACHE_UNINITIALIZE_EVENT
+0x0b4 NeedToZeroVacb    : Ptr32 _VACB
+0x0b8 BcbSpinLock       : Uint4B
+0x0bc Reserved          : Ptr32 Void
+0x0c0 Event             : _KEVENT
+0x0d0 VacbPushLock      : _EX_PUSH_LOCK
+0x0d8 PrivateCacheMap   : _PRIVATE_CACHE_MAP
```

As said at the beginning of this section, shared cache map structure holds information about the address at which each section of the file is mapped in memory. This information is kept in structures called Virtual Address Control Block(VACB) of type

`_VACB`. Each VACB represents one mapped view of the file that is a section of 256KB or less of file's content. This structure is shown below:

```
kd> dt _VACB
   +0x000 BaseAddress      : Ptr32 Void
   +0x004 SharedCacheMap   : Ptr32 _SHARED_CACHE_MAP
   +0x008 Overlay          : __unnamed
   +0x010 LruList          : _LIST_ENTRY
```

Field *BaseAddress* in this structure points to the base address of the mapped view in the memory. Field *SharedCacheMap* is a pointer to the shared cache map structure for this file. Using field BaseAddress, we can retrieve the section of the file that this view represents from the memory. There is one VACB for each 256KB of a file. These VACBs are kept in an array which is pointed by array *Vacbs* unless the file is 1MB or less in which case they are stored in an array of size four that is pointed by field *InitialVacbs*. A VACB is active when field *BaseAddress* in this structure points to a valid address. If the content of the section of the file that the VACB describes is not mapped to the memory, then field *BaseAddress* is 0. The cache manager brings the file content into memory as the file accesses different sections. Therefore, depending on which section of the file are accessed, the file will be partially mapped to memory. This partial content of the file can be extracted by going through the VACB arrays and copying the memory content from the addresses that active VACBs point to. In the example that follows, the content of a file object at the address of `0x87212028` that represents a JPG file is extracted from the memory cache. You can see the JPG signature at the beginning of the extracted memory chunk:

```
kd> dt _FILE_OBJECT 0x87212028
```

```
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x8735be30 _DEVICE_OBJECT
   +0x008 Vpb              : 0x873cc228 _VPB
   +0x00c FsContext        : 0xe48980d0
   +0x010 FsContext2       : 0xe4898228
   +0x014 SectionObjectPointer : 0x871bd9b4 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : 0x871b10e0
   .....
   +0x030 FileName         : _UNICODE_STRING "\Untitled.jpg"
   .....


kd> dt _SECTION_OBJECT_POINTERS 0x871bd9b4
   +0x000 DataSectionObject : 0x86e12c78
   +0x004 SharedCacheMap   : 0x871b1008
   +0x008 ImageSectionObject : (null)


kd> dt _SHARED_CACHE_MAP 0x871b1008
   +0x000 NodeTypeCode     : 767
   +0x002 NodeByteSize     : 304
   +0x004 OpenCount        : 1
   +0x008 FileSize         : _LARGE_INTEGER 0x532e
   +0x010 BcbList          : _LIST_ENTRY [ 0x871b1018 - 0x871b1018 ]
```

```
  +0x018 SectionSize      : _LARGE_INTEGER 0x40000
  +0x020 ValidDataLength  : _LARGE_INTEGER 0x532e
  +0x028 ValidDataGoal    : _LARGE_INTEGER 0x532e
  +0x030 InitialVacbs     : [4] 0x873a0048 _VACB
  +0x040 Vacbs            : 0x871b1038  -> 0x873a0048 _VACB
  +0x044 FileObject       : 0x87212028 _FILE_OBJECT
  +0x048 ActiveVacb       : 0x873a0048 _VACB
  +0x04c NeedToZero       : (null)
  +0x050 ActivePage       : 0
  +0x054 NeedToZeroPage   : 0
  +0x058 ActiveVacbSpinLock : 0
  +0x05c VacbActiveCount  : 1
  +0x060 DirtyPages       : 0
  +0x064 SharedCacheMapLinks : _LIST_ENTRY [ 0x8714dd8c - 0x86808ca4 ]
  .....

kd> dt _VACB 0x873a0048
  +0x000 BaseAddress      : 0xd5600000
  +0x004 SharedCacheMap   : 0x871b1008 _SHARED_CACHE_MAP
  +0x008 Overlay          : __unnamed
  +0x010 LruList          : _LIST_ENTRY [ 0x873a0dc0 - 0x8739ede0 ]

kd> dc 0xd5600000 l1000
d5600000  e0ffd8ff 464a1000 01004649 48000101  ......JFIF.....H
d5600010  00004800 1600e1ff 66697845 4d4d0000  .H......Exif..MM
```

```
d5600020  00002a00 00000800 00000000 4300dbff  .*.............C
d5600030  04030500 05030404 05040404 07060505  ................
d5600040  0707080c 0b0f0707 110c090b 1112120f  ................
d5600050  1311110f 13171c16 11151a14 18211811  ..............!.
d5600060  1f1d1d1a 17131f1f 1e222422 1f1e1c24  ........"$".$...
......
```

3.3 Security Manager

Windows security manager is a component of Windows that is responsible for ensuring the enforcement of access control policies over objects. When a thread accesses an object, the security manager verifies if thread is authorized to perform the requested type of access on the target object. This process is realized by maintaining the access privileges of each thread as well as possible accesses to each object.

The thread privileges are kept in a structure of type `_TOKEN`. Field *Token* in structure `_EPROCESS` of a process points to the security token of the process that contains the access privileges of the process. Except for thread impersonation, this token is used by the security manager to verify the authorization of an access. Thread impersonation is discussed later in this chapter when we detail the security manager. For the complete listing of structure `_TOKEN` please refer to Appendix 1. This structure has two important arrays:

- Filed *UserAndGroups* points to an array of type `_SID_AND_ATTRIBUTES`. Each member of this array contains a SID and the attributes describing whether or not the entry is mandatory and default enabled. In Windows, each user account is assigned an identifier of variable length which is called SID. The users are identified during the execution of the system based on their SID. The SID consists of a SID

revision number, a 48 bit authority identifier, a variable number of sub-authority identifiers or relative identifier (RID) . The authority value determines the agent that assigned the SID. The sub-authority identifiers describe different components trusted by the authority who issued the SID.

When Windows is installed, a SID will be assigned to the computer and each local account. The local account SIDs are created by appending a RID to the system SID. The RID starts from 1000 and is incremented for each new account. Moreover, when a domain is created, a SID will be assigned to it and the new SID will be used in assigning SIDs to the account that are created in this domain. Windows assigns predefined RIDs to some local accounts and groups. Examples include the administrator account that has the RID of 500 and the guest account that has the RID of 501. Additionally, a set of built-in local and domain SIDs are hard-coded in Windows installer. For each log on session, a SID is generated by `Winlogon` process with value of S-1-5-5-0 and a random RID appended to it.

- Field *Privileges* points to an array of type `_LUID_AND_ATTRIBUTES`. The length of this array is stored in field *PrivilegeCount* in the token. Each member of this array specifies a privilege and wether or not this privilege is enabled. Windows security privileges are listed below as defined in `ntddk.h` file:

```
#define SE_MACHINE_ACCOUNT_PRIVILEGE        (6L)
#define SE_TCB_PRIVILEGE                    (7L)
#define SE_SECURITY_PRIVILEGE               (8L)
#define SE_TAKE_OWNERSHIP_PRIVILEGE         (9L)
#define SE_LOAD_DRIVER_PRIVILEGE            (10L)
#define SE_SYSTEM_PROFILE_PRIVILEGE         (11L)
#define SE_SYSTEMTIME_PRIVILEGE             (12L)
```

```
#define SE_PROF_SINGLE_PROCESS_PRIVILEGE  (13L)
#define SE_INC_BASE_PRIORITY_PRIVILEGE    (14L)
#define SE_CREATE_PAGEFILE_PRIVILEGE      (15L)
#define SE_CREATE_PERMANENT_PRIVILEGE     (16L)
#define SE_BACKUP_PRIVILEGE               (17L)
#define SE_RESTORE_PRIVILEGE              (18L)
#define SE_SHUTDOWN_PRIVILEGE             (19L)
#define SE_DEBUG_PRIVILEGE                (20L)
#define SE_AUDIT_PRIVILEGE                (21L)
#define SE_SYSTEM_ENVIRONMENT_PRIVILEGE   (22L)
#define SE_CHANGE_NOTIFY_PRIVILEGE        (23L)
#define SE_REMOTE_SHUTDOWN_PRIVILEGE      (24L)
#define SE_UNDOCK_PRIVILEGE               (25L)
#define SE_SYNC_AGENT_PRIVILEGE           (26L)
#define SE_ENABLE_DELEGATION_PRIVILEGE    (27L)
#define SE_MANAGE_VOLUME_PRIVILEGE        (28L)
#define SE_IMPERSONATE_PRIVILEGE          (29L)
#define SE_CREATE_GLOBAL_PRIVILEGE        (30L)
#define SE_MAX_WELL_KNOWN_PRIVILEGE       (SE_CREATE_GLOBAL_PRIVILEGE)
```

Rootkits can overwrite or enable these privileges to gain the proper security authorization level to perform their malicious activities.

The attribute field in each member of both of these arrays describes whether the privilege or SID is enabled. According to `ntddk.h`, these attributes can have the following values:

```
#define SE_PRIVILEGE_ENABLED_BY_DEFAULT (0x00000001L)
```

```
#define SE_PRIVILEGE_ENABLED            (0x00000002L)
#define SE_PRIVILEGE_REMOVED            (0X00000004L)
#define SE_PRIVILEGE_USED_FOR_ACCESS    (0x80000000L)
```

Below you can see how the security context and privilege information can be extracted from the token. First, the address of the hh.exe process token is found. The content of the process token is shown afterwards.

```
kd> !process 86734600 PROCESS 86734600  SessionId: 0  Cid: 0734
Peb: 7ffdd000  ParentCid: 0218
    DirBase: 07a00820  ObjectTable: e47baa20  HandleCount: 388.
    Image: hh.exe
    VadRoot 86e913d0 Vads 248 Clone 0 Private 4176. Modified 14233. Locked 0.
    DeviceMap e3294680
    Token                             e23bfcf0
    ElapsedTime                       6 Days 05:17:29.793
    UserTime                          00:01:00.750
    KernelTime                        00:01:52.796
    QuotaPoolUsage[PagedPool]         169364
    QuotaPoolUsage[NonPagedPool]      9920
    Working Set Sizes (now,min,max)  (3614, 50, 345) (14456KB, 200KB, 1380KB)
    PeakWorkingSetSize                8988
    VirtualSize                       145 Mb
    PeakVirtualSize                   166 Mb
    PageFaultCount                    109026
    MemoryPriority                    BACKGROUND
```

```
    BasePriority                     8

    CommitCharge                     5336

kd> dt _TOKEN e23bfcf0
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER 0x7fffffff'ffffffff
   +0x030 TokenLock        : 0x8729b378 _ERESOURCE
   +0x038 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x040 ModifiedId       : _LUID
   +0x048 SessionId        : 0
   +0x04c UserAndGroupCount : 0xb
   +0x050 RestrictedSidCount : 0
   +0x054 PrivilegeCount   : 0x14
   +0x058 VariableLength   : 0x26c
   +0x05c DynamicCharged   : 0x1f4
   +0x060 DynamicAvailable : 0
   +0x064 DefaultOwnerIndex : 0
   +0x068 UserAndGroups    : 0xe23bfe80 _SID_AND_ATTRIBUTES
   +0x06c RestrictedSids   : (null)
   +0x070 PrimaryGroup     : 0xe23d2e10
   +0x074 Privileges       : 0xe23bfd90 _LUID_AND_ATTRIBUTES
   +0x078 DynamicPart      : 0xe23d2e10  -> 0x501
```

```
   +0x07c DefaultDacl        : 0xe23d2e2c _ACL
   +0x080 TokenType          : 1 ( TokenPrimary )
   +0x084 ImpersonationLevel : 0 ( SecurityAnonymous )
   +0x088 TokenFlags         : 0x89
   +0x08c TokenInUse         : 0x1 ''
   +0x090 ProxyData          : (null)
   +0x094 AuditData          : (null)
   +0x098 OriginatingLogonSession : _LUID
   +0x0a0 VariablePart       : 0x17
```

As mentioned before, field *UserAndGroups* points to the beginning of the array that contains the SIDs. The first member of this array is at the address that is pointer by field *UserAndGroups*, which is 0xe2269528. The content of this array member is shown below:

```
kd> dt _SID_AND_ATTRIBUTES 0xe23bfe80
   +0x000 Sid                : 0xe23bfed8
   +0x004 Attributes         : 0
```

Field *Sid* points to the beginning of the location where the corresponding SID is stored. The content of this location is shown below:

```
kd> dd 0xe23bfed8
e23bfed8  00000501 05000000 00000015 4632ec15
e23bfee8  e023eb71 13379b66 000001f4 00000501
e23bfef8  05000000 00000015 4632ec15 e023eb71
```

Command `!sid` from windbg could also be used to view the SID.

```
kd> !sid 0xe23bfed8 SID is:
S-1-5-21-1177742357-3760450417-322411366-500
```

Notice that the output of `!sid` command is in decimal format. For example,`1177742357` equals the hex value of `0x4632ec15`. In the same way, field *Privileges* in the structure `_TOKEN` points to the beginning of the process privilege arrays at the address of `0x0xe23bfd90`. The first member of this array is shown below:

```
kd> dt _LUID_AND_ATTRIBUTES 0xe2269438
   +0x000 Luid             : _LUID
   +0x008 Attributes       : 1
```

The content of field *Luid* is shown below:

```
kd> dd 0xe2269438
e2269438  00000017 00000000 00000001 00000008
e2269448  00000000 00000000 00000011 00000000
```

As you can see the privilege specified in this LUID is 0x17 which is `SE_BACKUP_PRIVILEGE` according to the above definitions. The value of 1 for field *Attributes* in this structure means that the options is `SE_PRIVILEGE_ENABLED_BY_DEFAULT`. We can verify our results using command `!token` of `windbg` as shown below:

```
kd> !token
Thread is not impersonating. Using process token...
_EPROCESS 871528a8, _ETHREAD 86d2c890, _TOKEN e20b0030
TS Session ID: 0
User: S-1-5-21-1177742357-3760450417-322411366-500
Groups:
 00 S-1-5-21-1177742357-3760450417-322411366-513
```

```
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-21-1177742357-3760450417-322411366-1011
    Attributes - Mandatory Default Enabled
 03 S-1-5-21-1177742357-3760450417-322411366-1096
    Attributes - Mandatory Default Enabled
 04 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
 05 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 06 S-1-5-4
    Attributes - Mandatory Default Enabled
 07 S-1-5-11
    Attributes - Mandatory Default Enabled
 08 S-1-5-5-0-82499
    Attributes - Mandatory Default Enabled LogonId
 09 S-1-2-0
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-1177742357-3760450417-322411366-513
Privs:
 00 0x000000017 SeChangeNotifyPrivilege          Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege              Attributes -
 02 0x000000011 SeBackupPrivilege                Attributes -
 03 0x000000012 SeRestorePrivilege               Attributes -
```

```
 04 0x00000000c SeSystemtimePrivilege            Attributes -
 05 0x000000013 SeShutdownPrivilege              Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege        Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege         Attributes -
 08 0x000000014 SeDebugPrivilege                 Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege     Attributes -
 10 0x00000000b SeSystemProfilePrivilege         Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege  Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege  Attributes -
 13 0x00000000a SeLoadDriverPrivilege            Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege        Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege         Attributes -
 16 0x000000019 SeUndockPrivilege                Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege          Attributes -
 18 0x00000001e Unknown Privilege                Attributes - Enabled Default
 19 0x00000001d SeImpersonatePrivilege           Attributes - Enabled Default
Authentication ID:         (0,15c1e)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x89 ( Token in use )
Token ID: 37d821f3         ParentToken ID: 0
Modified ID:               (0, 37d821f5)
RestrictedSidCount: 0      RestrictedSids: 00000000
```

One important thing to note is that the security context of users and processes in Windows are based on SIDs rather than usernames. To acquire the username that cor-

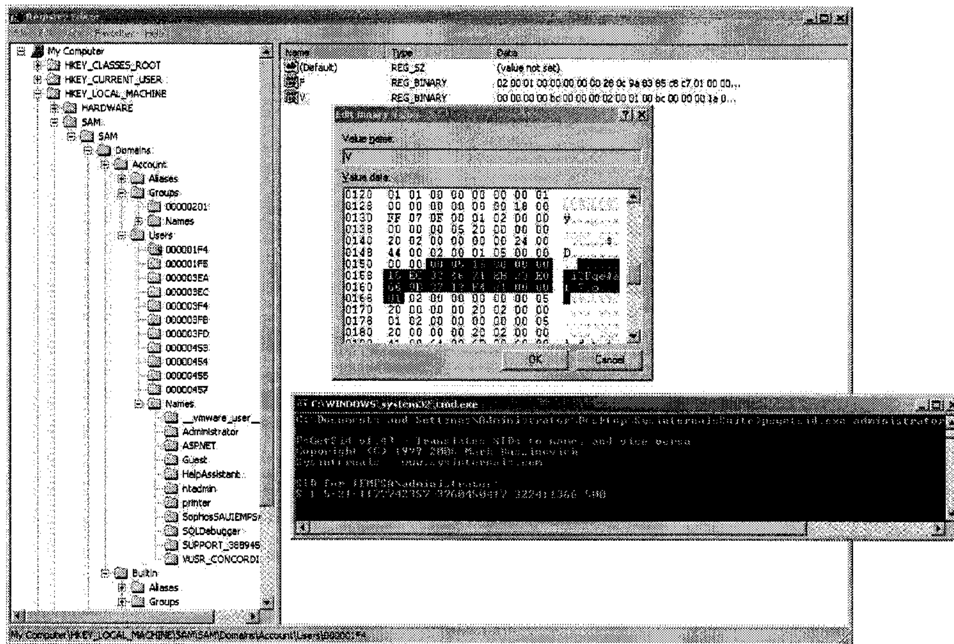


Figure 3.5: Windows registry contains user account information.

responds to an SID, the registry key **HKEY_LOCAL_MACHINE/SAM/SAM/Domains/Account/Users** could be used. This registry key contains the mapping between user and group information and SIDs. The subkey *Names* under this key contains a subkey for each account name. The key for each account name has a value the type of which specifies another subkey under Users key that contains the account information for this user. The SID of the user is stored in this key. Figure 3.5 shows this registry key's content and the account information for *Administrator* account. The command line shows the output of tool *psgetsid*. This tool from Sysinternals can be used to find the SID of and account or viceversa.

Knowing the security context and privileges of a thread, Windows security manager maintains security by checking the accesses to objects against the access control policy of each object. Each object has an Access Control List (ACL) that is made up

of Access Control Entries (ACE). There exist two types of ACLs: Discretionary Access Control List (DACL) and System Access Control List (SACL). DACL contains the type of access each SID can have to the object. The access types could be of four types: access allowed, access denied, allowed object, and denied object. Access allowed grants access to the object. Access denied denies any access to the object. Allowed object and denied object have the same meanings with the distinction that these access type qualifiers are used in active directories and they can further specify which objects the ACE is applied to where each object is uniquely identified in the network using a 128-bit GUID. If DACL of an object is null, every user can access it and if it is zero no access from any user is allowed.

The SACL specifies the auditing policy of the system. ACEs in this list contain an SID, type of access and whether it should be logged or not. Based on these ACEs, the object manager generates the proper audits. Each ACE can be of two types: system audit and system object audit. Again system object audit ACEs have the same role as system audit ACEs with the difference that they cover objects in the active directory and specify which objects the ACE applies to.

The DASL and SACL of objects whose security is maintained by object manager is specified in the header of the object. Field *SecurityDescriptor* in structure `_OBJECT_HEADER` points to a structure that contains the following information:

- Header: The header contains information about which structure elements are present, the revision number and how should these security properties be propagated through inheritance.
- Owner SID: The SID of the principal who owns the object.
- Group SID: The SID of the group of the object and is only used by POSIX subsystems.

- SACL and DACL.

The structure of security descriptor is shown below:

```
kd> dt _security_descriptor
nt!_SECURITY_DESCRIPTOR
   +0x000 Revision          : UChar
   +0x001 Sbz1              : UChar
   +0x002 Control           : Uint2B
   +0x004 Owner             : Ptr32 Void
   +0x008 Group             : Ptr32 Void
   +0x00c Sacl              : Ptr32 _ACL
   +0x010 Dacl              : Ptr32 _ACL
```

Fields *Owner* and *Group* are pointers to the SID of the owner and primary group of the object as stated above. Both fields *Sacl* and *Dacl* are of type _ACL. These two fields contain the offset of the SACL and DACL of the object from the beginning of the object security descriptor respectively. The structure _ACL is shown below:

```
kd> dt _ACL
   +0x000 AclRevision       : UChar
   +0x001 Sbz1              : UChar
   +0x002 AclSize           : Uint2B
   +0x004 AceCount          : Uint2B
   +0x006 Sbz2              : Uint2B
```

Field *AclSize* stores the size of the ACL array. Field *AclCount* has the number of ACE entries in this ACL. The reason for including both the size of the ACL and the

number of elements is that depending on the size of the ACE, it can have different sizes. Each ACE starts with a header that identifies its type. The ACE header and different types of ACE structures are shown below as documented by Doxygen project [36].

```
typedef struct _ACE_HEADER {
        BYTE AceType;
        BYTE AceFlags;
        WORD AceSize;
}
typedef struct _ACCESS_ALLOWED_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _ACCESS_DENIED_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _SYSTEM_AUDIT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _SYSTEM_ALARM_ACE {
        ACE_HEADER Header;
```

```
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _ACCESS_ALLOWED_OBJECT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD Flags;
        GUID ObjectType;
        GUID InheritedObjectType;
        DWORD SidStart;
}
typedef struct _ACCESS_DENIED_OBJECT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD Flags;
        GUID ObjectType;
        GUID InheritedObjectType;
        DWORD SidStart;
}
typedef struct _SYSTEM_AUDIT_OBJECT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD Flags;
        GUID ObjectType;
        GUID InheritedObjectType;
```

```
        DWORD SidStart;
}
typedef struct _SYSTEM_ALARM_OBJECT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD Flags;
        GUID ObjectType;
        GUID InheritedObjectType;
        DWORD SidStart;
}
typedef struct _ACCESS_ALLOWED_CALLBACK_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _ACCESS_DENIED_CALLBACK_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
typedef struct _SYSTEM_AUDIT_CALLBACK_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD SidStart;
}
```

```
typedef struct _SYSTEM_ALARM_CALLBACK_ACE {

        ACE_HEADER Header;

        ACCESS_MASK Mask;

        DWORD SidStart;

}

typedef struct _ACCESS_ALLOWED_CALLBACK_OBJECT_ACE {

        ACE_HEADER Header;

        ACCESS_MASK Mask;

        DWORD Flags;f

        GUID ObjectType;

        GUID InheritedObjectType;

        DWORD SidStart;

}

typedef struct _ACCESS_DENIED_CALLBACK_OBJECT_ACE {

        ACE_HEADER Header;

        ACCESS_MASK Mask;

        DWORD Flags;

        GUID ObjectType;

        GUID InheritedObjectType;

        DWORD SidStart;

}

typedef struct _SYSTEM_AUDIT_CALLBACK_OBJECT_ACE {

        ACE_HEADER Header;

        ACCESS_MASK Mask;

        DWORD Flags;
```

```
        GUID ObjectType;
        GUID InheritedObjectType;
        DWORD SidStart;
}
typedef struct _SYSTEM_ALARM_CALLBACK_OBJECT_ACE {
        ACE_HEADER Header;
        ACCESS_MASK Mask;
        DWORD Flags;
        GUID ObjectType;
        GUID InheritedObjectType;
        DWORD SidStart;
}
```

Field *AceType* in _ACE_HEADER structure specifies the type of the ACE and can have one of the following values:

```
#define ACCESS_ALLOWED_ACE_TYPE                  (0x0)
#define ACCESS_DENIED_ACE_TYPE                   (0x1)
#define SYSTEM_AUDIT_ACE_TYPE                    (0x2)
#define SYSTEM_ALARM_ACE_TYPE                    (0x3)
#define ACCESS_ALLOWED_COMPOUND_ACE_TYPE         (0x4)
#define ACCESS_ALLOWED_OBJECT_ACE_TYPE           (0x5)
#define ACCESS_DENIED_OBJECT_ACE_TYPE            (0x6)
#define SYSTEM_AUDIT_OBJECT_ACE_TYPE             (0x7)
#define SYSTEM_ALARM_OBJECT_ACE_TYPE             (0x8)
#define ACCESS_ALLOWED_CALLBACK_ACE_TYPE         (0x9)
```

```
#define ACCESS_DENIED_CALLBACK_ACE_TYPE        (0xA)
#define ACCESS_ALLOWED_CALLBACK_OBJECT_ACE_TYPE (0xB)
#define ACCESS_DENIED_CALLBACK_OBJECT_ACE_TYPE  (0xC)
#define SYSTEM_AUDIT_CALLBACK_ACE_TYPE          (0xD)
#define SYSTEM_ALARM_CALLBACK_ACE_TYPE          (0xE)
#define SYSTEM_AUDIT_CALLBACK_OBJECT_ACE_TYPE   (0xF)
#define SYSTEM_ALARM_CALLBACK_OBJECT_ACE_TYPE   (0x10)
```

Field *Mask* in each ACE is of type `ACCESS_MASK` and defines the type of access as discussed before. It can have one of the following values:

```
#define ACCESS_ALLOWED_ACE_TYPE        (0x0)
#define ACCESS_DENIED_ACE_TYPE         (0x1)
#define SYSTEM_AUDIT_ACE_TYPE          (0x2)
#define SYSTEM_ALARM_ACE_TYPE          (0x3)
```

Using these structure, the investigator is able to retrieve the access control policies of an object. Figure 3.6 shows this process. In this figure, the access control list of a process object is retrieved by first finding the object header of the process object and then traversing through the security descriptor, and then DACL list of the security descriptor. The `!sd` command in windbg does the same thing and its output is shown in Figure 3.7. Please notice that in both figures, in order to find the security descriptor of the object, the last three bits of field *SecurityDescriptor* are zeroed out since these fields are used as flags and the security descriptor is 8-byte aligned.

```
kd> !object 81f8e568
Object: 81f8e568  Type: (82bcbca0) Process
    ObjectHeader: 81f8e550
    HandleCount: 2  PointerCount: 125


kd> dt _OBJECT_HEADER 81f8e550
   +0x000 PointerCount     : 125
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
   +0x008 Type             : 0x82bcbca0 _OBJECT_TYPE
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x828fe810 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x828fe810
   +0x014 SecurityDescriptor : 0xe2071865
   +0x018 Body             : _QUAD

kd> dt _SECURITY_DESCRIPTOR (0xe2071865 & -8)
   +0x000 Revision         : 0x1 ''
   +0x001 Sbz1             : 0 ''
   +0x002 Control          : 0x8004
   +0x004 Owner            : 0x00000054
   +0x008 Group            : 0x00000070
   +0x00c Sacl             : (null)
   +0x010 Dacl             : 0x00000014 _ACL


kd> dt _ACL (0xe2071860 + 0x14)
   +0x000 AclRevision      : 0x2 ''
   +0x001 Sbz1             : 0 ''
   +0x002 AclSize          : 0x40
   +0x004 AceCount         : 2
   +0x006 Sbz2             : 0


kd> dd (0xe2071860 + 0x14 + 0x8)
e207187c  00240000 001f0fff 00000501 05000000
e207188c  00000015 b9ce651a b6d045de 76321bab
e207189c  000003ee 00140000 001f0fff 00000101
e20718ac  05000000 00000012 00000501 05000000
e20718bc  00000015 b9ce651a b6d045de 76321bab
e20718cc  000003ee 00000501 05000000 00000015
e20718dc  b9ce651a b6d045de 76321bab 00000201
e20718ec  00000000 0c0b0415 61564d43 004c0000
```

Figure 3.6: The extraction of access control policies of the object.

```
kd> !sd (0xe2071865 & -8)
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-3117311258-3067102686-1982995371-1006
->Group   : S-1-5-21-3117311258-3067102686-1982995371-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x40
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-3117311258-3067102686-1982995371-1006

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Sacl    :  is NULL
```

Figure 3.7: The output of `!sd` command.

Chapter 4

Digital Investigation and Memory Forensics

This chapter starts with a short discussion on the state of the art in digital investigation. Different proposals on forensic analysis, and digital investigation processes are touched. This discussion is followed by a detailed elaboration of the state of the art research on physical memory forensics. During this discussion, different approaches are discussed in detail. This discussion relies extensively on the structures that were introduced in the previous section.

4.1 Digital forensics

The state of the art on cyber forensic analysis could be classified in the following categories: Baseline analysis, root cause analysis, common vulnerability analysis, timeline analysis, semantic integrity check analysis and memory analysis.

Baseline analysis, proposed in [26], uses an automated tool that checks for the differences between a baseline of the safe state of the system and the state during the

incident. An approach to post-incident root-cause analysis of digital incidents through the separation of the information systems into different security domains and modeling the transactions between these domains is proposed in [34].

The common vulnerability analysis [1], involves searching through a database of common vulnerabilities and investigating the case according to the related past and known vulnerabilities. The timeline analysis approach [18] consists of analyzing logs, and scheduling information to develop a timeline of the events that led to the incident. The semantic integrity checking approach [33] uses a decision engine that is endowed with a tree to detect semantic incongruities. The decision tree reflects pre-determined invariant relationships between redundant digital objects.

In [15], P. Gladyshev proposes a formalization of digital evidence and event reconstruction based on finite state machines. Other research on formalized forensic analysis includes the formalization of event time binding in digital investigation [16, 24], which proposes an approach to constructing formalized forensic procedures. The absence of a satisfactory and general methodology for forensic log analysis has resulted in ad hoc analysis techniques such as log analysis [28] and operating system-specific analysis [19].

4.2 Forensic analysis of physical memory

The DFRWS memory forensics challenge [2] is considered as one of the initiatives for the research on memory analysis. The challenge led to the development of two memory analysis tools: Memparser [10] and Kntlist [20] each capable of traversing the linked list of process structures kept by the operating system to extract information about a running process. In [8], M. Burdach presents an approach to retrieve process and file information from the memory of Unix operating system by following the unbroken links between data structures in the memory.

These tools and approaches retrieve information on the processes that were running at the time of taking the memory image by first locating the process `_EPROCESS` block and then extracting information regarding the threads created by the process, process environment variables, loaded DLLs, owned objects, etc. The main shortcoming of these tools is the fact that anti-forensic techniques exist that can hide an `_EPROCESS` block from these tools and therefore will not be noticed by the forensic analyst. The main anti-forensic approach developed to defeat these tools is a technique called Direct Kernel Object Manipulation [9]. If you remember, all of the processes that are running are doubly linked together through a structure member named *ActiveProcessLinks.* This field is of type `_LIST_ENTRY`. A `_LIST_ENTRY` contains two fields as shown below:

```
kd> dt _LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

Field *Flink* points to the next structure in the list and field *Blink* points to the previous structure in the list. The technique works mainly by removing the `_EPROCESS` block from this doubly linked list and changing the linked list members in a way that it prevents the detection and side effects of this manipulation. This is accomplished by changing field *Blink* in the *ActiveProcessLinks* of the next `_EPROCESS` structure in the list to point to the `_EPROCESS` structure that is before the process to be hidden and changing *Flink* in the *ActiveProcessLinks* of the previous `_EPROCESS` structure to point to the `_EPROCESS` structure that is after the process to be hidden. What makes it possible to hide a process without impacting the execution of it is that thread scheduler of Windows operates on thread basis rather than process basis. Depending on their states, the threads that are executing in the system are linked together in one of several linked lists that system maintains. These lists are *WaitList*, *SwapList*, *ThreadList* and

QueueList. Many malware analysis tools including kntlist and memparser or utilities that gather information about the processes running such as Windows Task Manager either use the API that is provided by kernel or walk though the linked list of _EPROCESS structure in order to acquire information about each process. The main kernel API that is used for this purpose is *ZwQuerySystemInformation*. Using this function, a variety of information about the system such as processes running and loaded modules can be retrieved. However, this API works again by traversing the linked list of different structures maintained by kernel such as _EPROCESS active process linked list. Therefore, removing a _EPROCESS structure from this linked list can effectively hide the process from the eye of these tools.

Another anti-forensic technique that is used by malwares in order to hide their existence is API hooking [37]. API hooking can be performed both at kernel and user land level. This is accomplished by overwriting the memory location that stores the address of a specific piece of code. For example, by overwriting the system service table entry that points to the code that implements *ZwQuerySystemInformation* service to make the calls to this function execute the code that the malware supplies, one can hide a desired process from being detected by above tools.

To defeat these anti-forensic techniques, in his paper titled "Searching for processes and threads in Microsoft Windows memory dumps" [32], A. Schuster proposes an approach to define signatures for executive object structures in the memory and recover the hidden and lost structures by scanning the memory looking for predefined signatures. We have already touched the idea when we talked about kernel pool structure. Now let us discuss this approach in more details. As said before, Windows kernel uses a special memory called pools for the allocation of memory. A pool allocation is preceded by a pool header that contains information about the size of the allocation and the size of

the previous pool allocation. Knowing that many executive objects are allocated from this memory, we can scan through this memory and look for a set of patterns that are defined for important structures such as _EPROCESS or _ETHREAD. The address of this part of the memory is stored by several kernel variables including *MmPagedPoolStart*, *MmPagedPoolEnd*, *MmNonPagedPoolStart*, and *MmNonPagedPoolEnd* for the starting and ending address of paged and non-paged pools respectively. However, an easier way to obtain these addresses are through the structure that is called Kernel Processor Control Region (KPCR). In Windows 2000, XP, and Vista, this structure is always at the hard coded virtual address of `0xffddff000`. Below is the details of this structure:

```
kd> dt _KPCR
   +0x000 NtTib            : _NT_TIB
   +0x01c SelfPcr          : Ptr32 _KPCR
   +0x020 Prcb             : Ptr32 _KPRCB
   +0x024 Irql             : UChar
   +0x028 IRR              : Uint4B
   +0x02c IrrActive        : Uint4B
   +0x030 IDR              : Uint4B
   +0x034 KdVersionBlock   : Ptr32 Void
   +0x038 IDT              : Ptr32 _KIDTENTRY
   +0x03c GDT              : Ptr32 _KGDTENTRY
   +0x040 TSS              : Ptr32 _KTSS
   +0x044 MajorVersion     : Uint2B
   +0x046 MinorVersion     : Uint2B
   +0x048 SetMember        : Uint4B
   +0x04c StallScaleFactor : Uint4B
```

```
+0x050 DebugActive      : UChar
+0x051 Number           : UChar
+0x052 Spare0           : UChar
+0x053 SecondLevelCacheAssociativity : UChar
+0x054 VdmAlert         : Uint4B
+0x058 KernelReserved   : [14] Uint4B
+0x090 SecondLevelCacheSize : Uint4B
+0x094 HalReserved      : [16] Uint4B
+0x0d4 InterruptMode    : Uint4B
+0x0d8 Spare1           : UChar
+0x0dc KernelReserved2  : [17] Uint4B
+0x120 PrcbData         : _KPRCB
```

Field *SelfPcr* points back to address stored in fs register which is the container KPCR structure itself. Field *NtTib* stores state information about the stack of the process. Field *KdVersionBlock* points to a structure of type `_KDDEBUGGER_DATA64`. This structure contains the value of many interesting unexported kernel variables. The complete listing of structure `_KDDEBUGGER_DATA64` is shown appendix 1 as defined in the Debugging Tools For Windows SDK [25] in header file wdbgexts.h. As you can see, this structure has a great deal of information useful in analyzing the physical memory including, the start and end address of the paged and non-paged pools, page frame number database, the head of the `_EPROCEE` linked list of active processes, the head of different page frame database linked list such as zero, free, standby, modified, available, and modified no-write lists, system cache start and system cache end, etc. The fact that the address of this structure is stored in the KPCR structure whose address is known, makes it a useful starting point. Another way to find the pool areas is using a structure of type

_MM_SESSION_SPACE that is pointed by field *Session* in any _EPROCESS structure that is executing in a session. This structure is shown below:

```
kd> dt _MM_SESSION_SPACE
   +0x000 ReferenceCount   : Uint4B
   +0x004 u                : __unnamed
   +0x008 SessionId        : Uint4B
   +0x00c SessionPageDirectoryIndex : Uint4B
   +0x010 GlobalVirtualAddress : Ptr32 _MM_SESSION_SPACE
   +0x014 ProcessList      : _LIST_ENTRY
   +0x01c NonPagedPoolBytes : Uint4B
   +0x020 PagedPoolBytes   : Uint4B
   +0x024 NonPagedPoolAllocations : Uint4B
   +0x028 PagedPoolAllocations : Uint4B
   +0x02c NonPagablePages  : Uint4B
   +0x030 CommittedPages   : Uint4B
   +0x038 LastProcessSwappedOutTime : _LARGE_INTEGER
   +0x040 PageTables       : Ptr32 _MMPTE
   +0x044 PagedPoolMutex   : _FAST_MUTEX
   +0x064 PagedPoolStart   : Ptr32 Void
   +0x068 PagedPoolEnd     : Ptr32 Void
   +0x06c PagedPoolBasePde : Ptr32 _MMPTE
   +0x070 PagedPoolInfo    : _MM_PAGED_POOL_INFO
   +0x094 Color            : Uint4B
   +0x098 ProcessOutSwapCount : Uint4B
   +0x09c ImageList        : _LIST_ENTRY
```

```
+0x0a4 GlobalPteEntry   : Ptr32 _MMPTE
+0x0a8 CopyOnWriteCount : Uint4B
+0x0ac SessionPoolAllocationFailures : [4] Uint4B
+0x0bc AttachCount      : Uint4B
+0x0c0 AttachEvent      : _KEVENT
+0x0d0 LastProcess      : Ptr32 _EPROCESS
+0x0d8 Vm               : _MMSUPPORT
+0x118 Wsle             : Ptr32 _MMWSLE
+0x11c WsLock           : _ERESOURCE
+0x154 WsListEntry      : _LIST_ENTRY
+0x15c Session          : _MMSESSION
+0x198 Win32KDriverObject : _DRIVER_OBJECT
+0x240 WorkingSetLockOwner : Ptr32 _ETHREAD
+0x244 PagedPool        : _POOL_DESCRIPTOR
+0x126c ProcessReferenceToSession : Int4B
+0x1270 LocaleId         : Uint4B
```

As you see, this structure also has the address information on memory pools. In order to verify if a piece of memory constitute an executive object, A. Schuster verifies first verifies if the memory section starts with a pool header, second checks if following the pool header there exist an object header and third verifies if the executive object starts with a dispatcher header. The type of the object can be defined based on the pool tag and the object type pointed by the header of the object. He defines three sets of rules for pool headers, object headers and dispatcher header. The followings are the rules that are used to verify a possible pool allocation unit:

1. There must be enough space preceding the current pool allocation unit to fit in

the previous pool allocation unit.

2. From the start of an assumed pool allocation unit, there must be enough space left in the current page to fit in the pool allocation unit based on the size of the block.

3. The assumed `_POOL_HEADER` structure has to be aligned on a 32 byte (for Windows 2000) or 8 byte boundary (later versions).

4. *PoolType* must be either zero for free blocks, non-paged class with an odd value or paged class with an even value. However, since executive objects are allocated in kernel non-paged pool, for recovering executive objects, even value should not be considered.

5. *PoolTag* with the value of `0xe36f7250` specifies a pool that is allocated with *Proc* tag and as shown before, pool units with this tag can denote that a process _EPROCESS structure is stored in the pool unit.

6. *PoolTag* with the value of `0xe5726854` specifies a pool that is allocated with *Thre* tag and as shown before, pool units with this tag can denote that a thread _ETHREAD structure is stored in the pool unit.

Concerning object headers, the following two rules are defined for process and thread executive objects:

1. For process objects, field *Name* in the object type structure that is pointed by field *Type* in structure `_OBJECT_HEADER` should point to a unicode string that has its *Buffer* pointing to the string "process". Therefore, field *Name.Buffer* should point to a memory with content of `0x636f7250`. Moreover, field *Name.Length* should be equal to 0x0e and field *Name.MaximumLength* should contain the value of 0x10.

2. Based on the same course of reasoning, for a thread object, field *Name.Buffer* should point to a memory with the content of `0x65726854`. Moreover, field *Name.Length* should be equal to `0x0c` and field *Name.MaximumLength* should contain the value of `0x0e`.

3. When an object is closed using function `nt!obpFreeObject`, field *Type* will be set to the value of `0xbad0b0b0`. Therefore, if the thread or process objects are closed, their type will have this value instead of a pointer to a unicode string.

And finally the following rules are defined for the dispatcher header. As discussed before, this header is used by Windows for synchronization of access to synchronizable objects and is the first member of structures `_KPROCESS` and `_KTHREADs` that are at the beginning of structures `_EPROCESS` and `_ETHREAD`. Two fields in this structure, *Size* and *Type*, have constant values during the life time of the object.

1. For process objects in Windows 2000, XP and 2003, field *Type* equals `0x01` and *Size* is `0x1b`.

2. For thread objects, *Type* equals `0x06`. In Windows 2000 the value of field *Size* is `0x6c`. In Windows XP, this value is `0x6c` and for Windows 2003, field *Size* equals `0x72`.

Schuster further specifies some properties of the structure members of structures `_EPROCESS` and `_ETHREAD`. These properties includes the size fields, synchronization events, the page alignment of page directory tables, and the fact that pointers to structures `_EPROCESS` and `_ETHREAD` should have values in the kernel address space and therefore be greater than `0x7fffffff`.

chris@bugcheck.org in his paper titled "GREPEXEC: Grepping Executive Objects from Pool Memory" introduces some more specific internal structure signatures for

grepping executive objects from memory. He also argues that some of these signatures are breakable by simply assigning a field to another value that evades the signature without impacting the operation of the operating system. Theoretically, it is possible to completely change the part of the memory that stores a specific structure and modify the operating system code through hooking or run time patching to continue its operation. Moreover, defining an exact signature that can uniquely identify some important data structures is not achievable. For example consider structure `_SUBSECTION`. As discussed in the previous chapter, this structure contains the mapping address information of a section of a file mapped to memory. However, due to the fact that most of the fields in this structure can store different values, it is not possible to detect the structure by scanning the memory. Lastly, it is possible that some part of a data structure is overwritten by later memory allocations and therefore make it not comply with the signatures while the rest of it that contain valuable information still exists in the memory.

Another research on memory forensics is the work in [27] that presents an extensible framework (FATKit), which provides the analyst with the ability to automatically derive digital object definitions from C source code and extract the underlying objects from memory. In [38], A. Walters and N. L. Petroni present an approach for extracting in-memory cryptographic keying material from disk encryption applications. B. Carrier and J. Grand in [23], discuss a strategy for robust address translation by incorporating invalid pages and paging file to improve the completeness of the analysis. This approach was detailed when we discussed the Windows memory manager in the previous section.

Most of the research on memory forensic analysis is focused on the extraction of relevant data structures from memory. After extracting these data structures, the forensic analyst will be left to analyze the gathered evidence to recover a time line from the events that happened during the incident. However, insufficient research results and

processes are available for the actual analysis of physical memory. In the next chapter, we discuss a technique that can be used to extract a partial execution history of the process from the extracted memory structures.

In this chapter we discussed the state of the art research on forensic analysis of physical memory. Several proposals were discussed and the advantage and limitations of each was discussed. The chapter was concluded with the emphasis on the fact that the previous research works on forensic analysis of physical memory have mainly focused on extraction of forensically valuable data structures and limited results and procedures are available that can help the investigator interpret the extracted data according to the facts of the case. In the next chapter we detail an analysis technique that can help an investigator to reconstruct the events that took place at the time of the execution of a process by analyzing program code and stack.

Chapter 5

Stack Trace Analysis

As discussed in the previous chapter, the research on digital investigation of physical memory has been limited mostly to the extraction of operating system structures that are of potential forensic value during the investigation. However, as we will show in this chapter a more informed analysis of the traces left in the memory from the execution of the programs can lead to better understanding of the chain of events happened during the time of the incident. In this chapter we first provide an overall introduction to our approach to forensic investigation and the motivations behind using this approach. This introduction is followed by a detailed discussion of the theory, algorithm and proofs of our approach.

5.1 Approach

In this section, we lay out the principles underlying our approach to the forensic analysis of stack leftovers. What makes this approach possible is the way the stack operates in the course of program execution. The stack mechanism is used in most of the prevalent operating systems to make structured programming possible. For each function call

made by a process, a stack frame is created and stored on the stack. The stack frame contains the parameters passed to the function, the return address, the previous value of the EBP register and the local variables of the function. These function call traces enclose the history of what a process has done during its course of execution.

After a function returns, the stack pointer is moved down to point to the previous stack frame. However, the returned function stack frame still resides in the memory until another call is made by the process, and the stack grows up enough to overwrite the frame. The depth of the stack at each point of the execution depends on the number of nested function calls that are made by the process as well as the length of each stack frame. Due to the fact that the depth of the stack has arbitrary values during the execution, a large number of previously called function stack frames stay on top of the stack untouched or partially overwritten. Moreover, current software engineering best practices encourage the implementation of a service through long chain of function calls with each component serving some part of the service requested. This fact intuitively reenforces our proposition.

The correlation of the stack with the program source code reveals the execution history of the program in terms of function call chains. We have developed a modeling technique, an algorithm and the system that makes this approach possible. As shown in Figure 5.1, the physical image acquired from the system under analysis is parsed to retrieve the process executable code and thread stacks. The stack frames are extracted by analyzing the thread stacks. The extracted executable is analyzed to produce the Control Flow Graph of each function and all the resulting CFGs are combined to form an abstract model of the program execution. The program model is correlated against the stack residues to produce all the possible execution paths that could be executed by the process and could generate the right stack leftover. For this purpose, we have

developed an algorithm that takes the program execution model and the stack residues and produces a state machine that contains all the solutions to the problem.

Some of the advantages of this technique include:

- The analysis is performed on the assembly code of the process that is extracted from the memory and there is no need for the external provision of the source code or executable. This feature overcomes the anti-forensic techniques that hide the executable in the filesystem by hooking operating system APIs or injecting the code directly into the memory.
- The technique integrates the formal analytical power of state machines and program models to retrieve the execution history of the process. This feature bestows the precision required in most jurisdictions for digital investigations.
- As stated before, the result of the analysis could reveal important facts about what was performed by the process at the time of the incident rather than what exists in the memory. This is of paramount importance to forensic investigation since the final goal of forensics is to discover the activities performed by the suspects with the exact order during the incident.
- The presented verification algorithm is able to retrieve all possible solutions to the problem. This enables the investigator to reach a sound and logical conclusion by considering all the possible execution paths that the program could have taken.

The following sections discuss the details involved in this process.

5.2 Modeling the process and the stack traces

In this section, we elaborate our approach to stack analysis by enumerating each step involved in the process and then discussing the details involved in each phase. Our

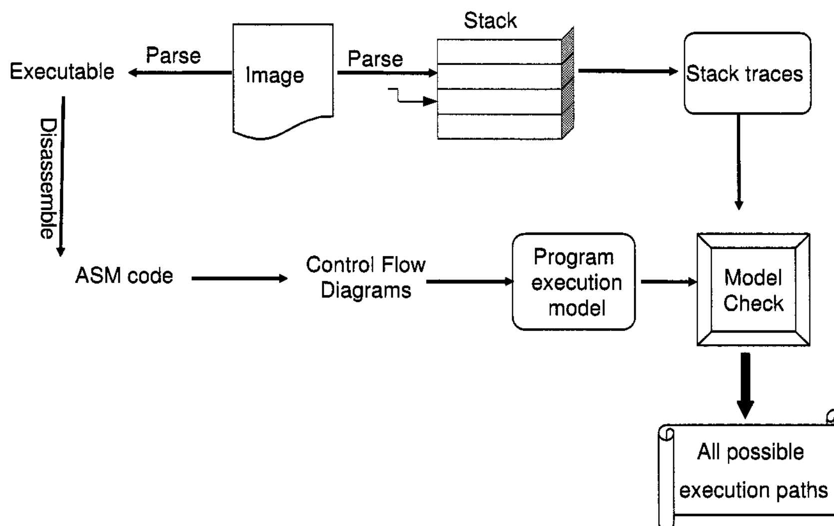


Figure 5.1: Our approach; The program model is correlated with the stack traces.

approach consists of four phases. First, we generate the Control Flow Graph (CFG) of all functions of the program. Second, the CFGs are transformed into finite state machines (FSM). Third, the finite state machines are combined to form a Push Down System (PDS). The resulting PDS models the program execution in terms of function calls and returns made by the program. Fourth, we generate an FSM as the solution to our problem by applying an algorithm that traverses the PDS model of the program, while generating the FSM based on the stack residues. The resulting FSM reflects all the possible execution paths that could be executed by the program and at the same time can generate the leftover found on the stack. In the following sections, we elaborate on each phase. As an example, consider the program shown in Figure 5.2. We use this program to clarify each phase. For simplicity, we chose a program written in c. However, it is important to note that since our approach only deals with function call and returns, exactly the same procedure is applicable to the assembly code.

```
1.  #include <iostream>
2.  void op(int i);
3.  void h(int i, int j) {
4.      return;
5.  }
6.  void g(int i, int j) {
7.      return;
8.  }
9.  void b(int i, int j, int k, int l) {
10.     return;
11.  }
12.  void e(int i, int j, int k, int l) {
13.     op(2);
14. }
15. void a(int i, int j, int k, int l) {
16.     if (i == 49) {
17.         g(i,j);
18.         e(i,j,k,l);
19.         return;
20.     }else{
21.         h(i,j);
22.         return;
23.        }
24. }
25. void c(int i, int j, int k, int l) {
26.     b(i,j,k,l);
27.     return;
28. }
29. void d(int i, int j, int k, int l) {
30.     h(i,j,k,l);
31.     return;
32. }
33. void op(int i) {
34.     char input;
35.     printf("Input a value
                between 1, 2:\n");
36.     fflush(stdin);
37.     scanf("%c", &input);
38.        a(i,0,0,0);
39.     switch (input) {
40.         case '1':
41.             d(0,0,0,0);
42.             break;
43.         case '2':
44.             c(0,0,0,0);
45.             break;
46.     }
47.     return;
48. }
49. void inc(int i) {
50.     if (i < 10) {
51.         inc(i+1);
52.     } else {
53.         op(1);
54.         return;
55.     }
56. }
57. void main() {
58.     inc(0);
59. }
```

Figure 5.2: Sample program to analyze.

5.2.1 Control Flow Graph

A control flow graph (CFG) [8] is a structure that characterizes possible execution paths in a program. Vertices of the graph contain one or more instructions of the program that execute sequentially. Edges in the graph show how control flow transfers between blocks.

Let f be a function in a program P. The control flow graph for f is denoted by $G_f = \langle V_f, E_f \rangle$ where V_f is the set of vertices and $E_f \subseteq V_f \times V_f$ is the set of edges. A vertex in G_f is a basic block.

Each v $\in V_f$ contains a sequential list of instructions in f satisfying the following properties: There is no control-flow transfer into the middle of a basic block nor a transfer out of the middle of a basic block. In defining basic blocks, notice that the call to a function is considered as a transfer of control out of the basic block and therefore, each basic block at most has one function call instruction. An edge $\langle v_j, v_k \rangle \in E_f$ if there exists a possible control flow from v_j to v_k.

The first step of our approach is the generation of a control flow graph of each function called in the program. As an example, the control flow graph of function op of the sample program is shown in Figure 5.3.

Having the CFG of a function, we generate the local automata model of the CFG as discussed in [14]. The local automata model of a CFG is a finite state machine whose states represent nodes of the CFG and its transitions are defined based on the control flows among different nodes of the CFG. Below is the formal definition of the local automata model.

Suppose that F is the set of functions in program P, C is the set of function call sites in P, and $\theta(c)$ denotes the target function of call site c. The local automata model of function f with control flow graph of $G_f = \langle V_f, E_f \rangle$ is defined as follows:

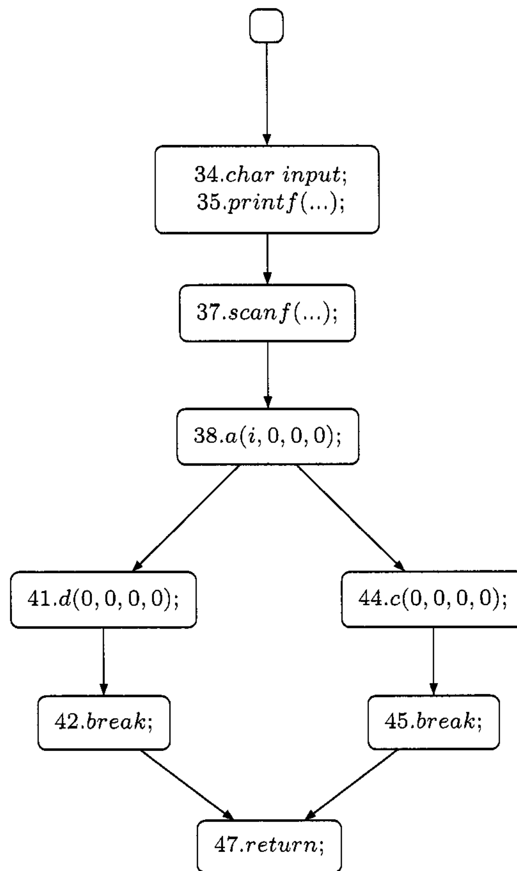


Figure 5.3: The control flow graph of op.

Let $a \lhd v$ indicate that vertex $v \in V_f$ contains call site a. The local model for f is $A_f = \langle Q_f, \Sigma_f, \delta_f, q_f, F_f \rangle$, where:

- $Q_f = V_f$.
- $\Sigma_f = C_f \cup \{\epsilon\}$
- $F_f = C_f$ where $C_f \subseteq C$.
- $q_f \in V_f$ is the CFG entry state.
- $F_f = \{v \in V_f | v\ is\ a\ CFG\ exit\ state\}$
- Function call transition: $\delta_f(p, a) = q$ if $a \lhd p, a \in C_f$, and $\langle p, q \rangle \in E_f$.
- ϵ-transition: $\delta(p, \epsilon) = q$ if $\langle p, q \rangle \in E_f$ and $\forall a \in Cf : \neg(a \lhd p)$

Please notice that we have changed the above model from the original version in [14] by removing the system call transitions. This is due to two facts; Firstly since we are analyzing the kernel stack as well as the user land stack we do not need to restrict our analysis only to the user land system calls. Secondly, depending on the extent of the analysis, a stack trace analysis could expand to only the functions inside the program, the system calls, the library calls or even the low level kernel function calls. Therefore, we have introduced the concept of the end function calls which are a set of function names that are defined by the analyst to limit the depth of the analysis. The CFG of end function calls has only one state which is both an entry and an exit state. Intuitively, the local automata model of a CFG is a finite state machine whose states represent the nodes of the CFG and edges are either the name of a function called from the originating node, or ϵ.

As explained in [14], the ϵ-reduction algorithm is performed on the local models to remove the ϵ transitions. This will increase the performance of the system since the

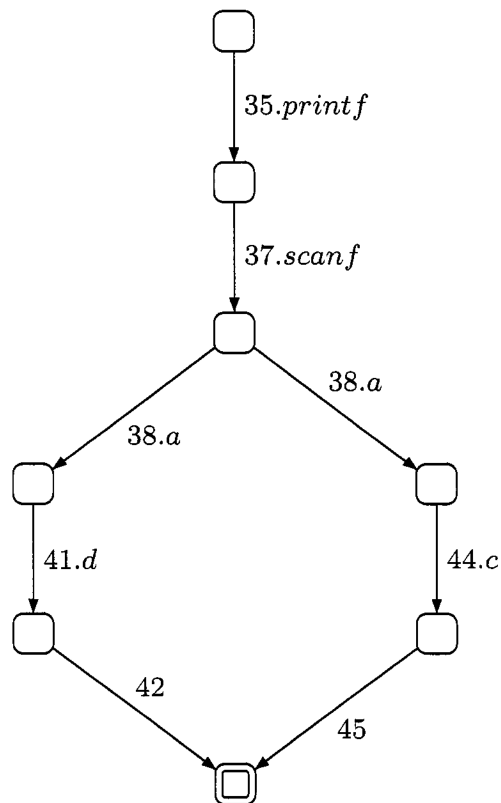


Figure 5.4: The local automata model of op.

ϵ edges are always traversed without consuming any symbol from the input. As an example, Figure 5.4 depicts the local automata model of function op. Notice that in Figure 5.4, we have included the line number in the transition names to differentiate among different calls to the same function. For the same reason, the definition of the local automata model of a CFG contains the concept of the function call site rather than the function name.

Until now, we have modeled the execution of the program as a set of local state machines each representing the execution of a function in the program. However, in

order to analyze the execution of a program as a whole, we have to combine the local state machine models into a global model. The resulting model should encompass all the possible control flows among the basic blocks of the program, while preserving the inter-procedural control flows. We have developed a modeling approach using Push Down Systems (PDS) [11] that accurately models the execution of the program in terms of function calls and returns made by the program. The model maintains the inter-procedural execution flows.

A PDS is a triple $P = (Q, \Gamma, \sigma)$ where Q is the final set of control locations, Γ is the finite set of stack alphabets and $\sigma \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of transition rules. The program execution in terms of the chain of function calls and returns made by the program is modeled using a PDS. We combine the local automata models of individual functions to form the PDS model of the whole program as follows:

> Again suppose that F is the set of functions in program P, C is the set of function call sites in P, $\theta(c)$ denotes the target function of call site c. The combination of the local models of the functions of a program is defined as the PDS $P = (Q, \Gamma, \sigma)$ where:
>
> - $Q = \bigcup Q_f$ for all $f \in F$.
> - $\Gamma = C$ is the set of stack variables.
> - Function call transition: $\sigma(p, \epsilon) = (q, r)$ if $\exists f \in F, c \in C$ such that $\delta_f(p, c) = r, q = q_{\theta(c)}$ where $q_{\theta(c)}$ is the entry state of the local automata model of $\theta(c)$.
> - Function return transition: $\sigma(p, t) = (q, \epsilon)$ if $\exists f \in F, r \in Q$ such that $\delta_f(r, c) = q, p \in F_{\theta(c)}$ where $F_{\theta(c)}$ is the set of final states of the local automata model of function $\theta(c)$.

As an example, Figure 5.5 shows the resulting PDS model of the program in Figure 5.2. For clarity, in the diagram, the stack operations are represented as labels of edges. An edge labeled as a call site represents the push operation and an edge labeled as a bared call site represents the pop operation. Notice that in our analysis we have considered the *scanf* and *printf* functions as end functions. However, a more detailed analysis could involve modeling the function calls inside these functions.

It is important to observe that the stack settings extracted from the memory is actually a configuration of the PDS model of the program if we suppose that the the PDS stack works in a similar way to the operating system stack. This means that popping an element from the PDS stack only brings the stack pointer down and does not remove the stack symbol from the stack. The configuration of a push-down system at any stage in its processing is determined by its current state and the content of its stack. In this case, we also have to include the position of the stack pointer of the PDS in the configuration since the old frames are not removed from the stack, but the stack pointer goes up and down along the stack.

To find the configuration of the program PDS at the time of the image was taken from the memory, we have to specify the state of the push down system. Notice that each line of the program can be mapped to a unique state of the PDS model of the stack. This means that using the value of the Program Counter register (PC), we can precisely specify the state of the PDS. Moreover, the stack of the program PDS can be thought of as a simplified version of the program stack itself. Therefore, the Stack Pointer register (SP) can be used to identify the position of the PDS stack pointer in the stack.

Another important fact to notice here is that a PDS does not have a starting state. However, in our modeling, the starting state of the program, that is the state representing the entry point of function *main* is considered as the entry state of the

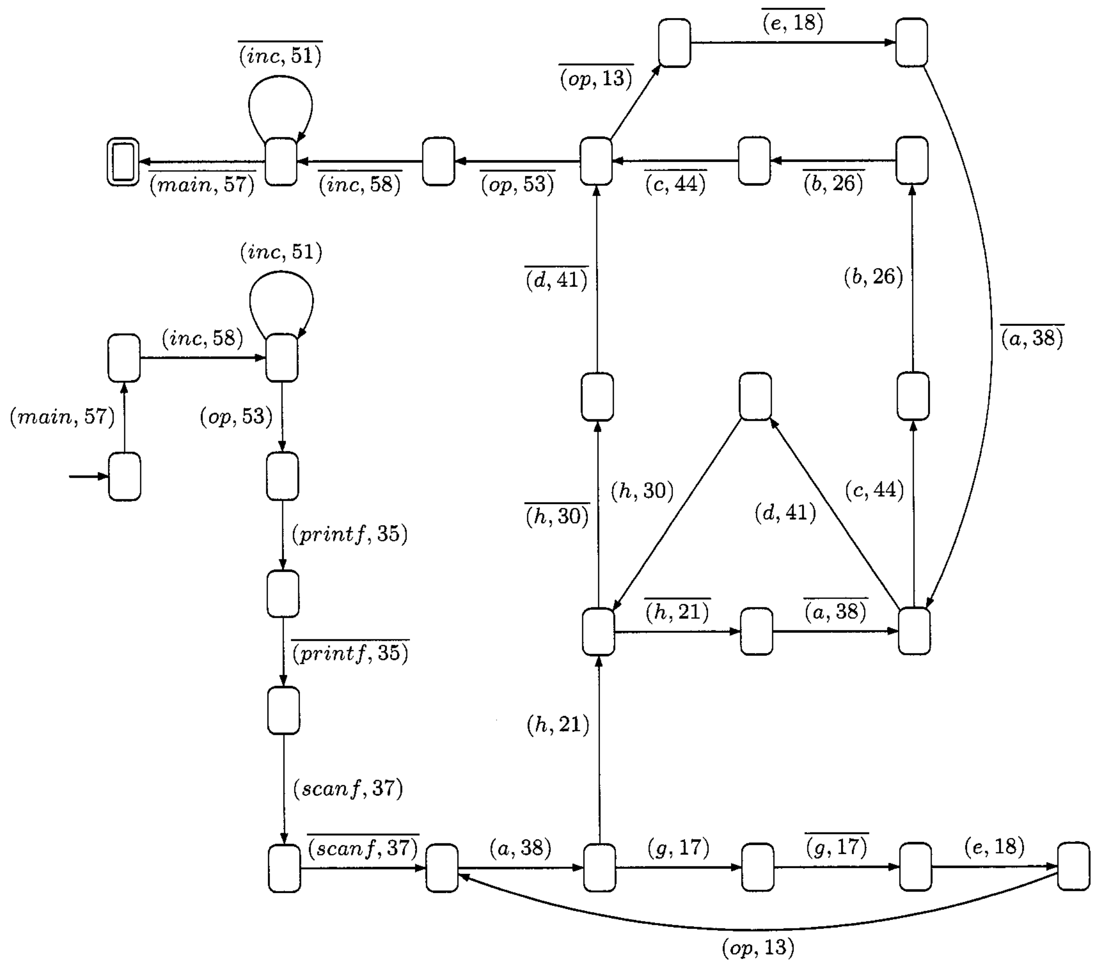


Figure 5.5: The PDS model of the program.

PDS. Moreover, the content of the PDS stack at the beginning of its execution is empty and the stack pointer points to the bottom of the stack.

5.2.2 Stack trace verification

In the previous section, we modeled the program execution. The model is able to capture all the execution paths of the program based on the functions calls and returns made by the program. In this section, we elaborate on our approach to generate all the possible execution paths that could be executed by the program and ,if executed, would generate the right stack residue.

A stack frame contains the address from which the program execution should continue after the function returns. Based on this address, the callee, the caller and the exact address of the call site in the code are identifiable. Consequently, each stack frame in the stack leftover represents a unique call site. Moreover, the stack frame stores the local variables and arguments. Therefore, depending on the number of local variables, arguments and push/pops, the length of the stack frames can be different from each others'. This means that a stack frame could be partially overwritten and therefore, some traces might still remain from it.

Since the PDS model described above captures program flows based on the call sites instead of the function name, each stack frame can be associated with a transition in the PDS model. In addition to the call sites, information regarding the length of the stack frame can be included in the model. Consequently, each function call in the PDS model is modeled as a triple $(site, callee, length)$ where *site* is the call site (line number) of the call made to *callee* and *length* is the length of the frame at that point of the execution. Accordingly, each frame in the stack trace is modeled as a five-tuple $(site, caller, callee, start, end)$ which represents function *callee* being called by function

caller at call site *site* and the stack frame starts at the depth of *start* and ends at the depth of *end*.

The length of the stack frame at each point of execution can be calculated statically by analyzing the assembly code [13]. However, as mentioned before, depending on the comprehensiveness of the analysis, the modeling could include program function calls, DDL calls and even kernel calls. It is important to notice that in calculating the length of the stack frame which is assigned to a function call, we should consider whether or not the function is chosen as an end function. Suppose that in our analysis, we have chosen function A to be an end function while it actually calls another function B in some DDL file. For our approach to work, we have to choose the length of the stack frame representing A in a way that it reflects the changes calling A can make to the stack. Therefore the selected frame length for the stack frame representing calling of A in our model will be the sum of the actual length of the stack frame A and stack frame B.

Depending on which execution path is taken by the process, the stack leftover can be an arbitrary combination of stack frames. However, based on the function call model of the program and the mechanism that stack works, a set of rules could be derived as follows:

> Suppose that $function(A)$ represent the function call which has generated stack frame A on top of the stack.
>
> - If stack frame $A = (c, a, b, -, -)$ is on top of stack frame $B = (f, d, e, -, -)$ and a is not equal to e then $function(A)$ has been called before $function(B)$.
> - If stack frame $A = (c, a, b, -, -)$ is on top of stack frame $B = (f, d, a, -, -)$ and there is no execution path in d's control flow graph that exits without calling any function, then $function(A)$ has been called by

$function(B)$.

Using these properties, it is possible to discover a set of possible execution order for functions representing stack frames. Having an order of the executions of the functions whose frames are on the stack, one might be able to execute the program (or follow the program control flow model) to generate the particular order of function execution. However, the resulting execution path that could generate the function call order still may not produce the same stack leftovers since the frames could be overwritten after creation. Therefore, after finding an execution path that can generate the right order of the function calls, the execution path should be actually executed to see if the resulting stack leftover is the same as the one that has remained on the stack at the time of the incident. Moreover, the execution of the process could involve infinite or long loops, which can make it almost impossible to check all the execution paths. Therefore a solution to the problem should be presented in a way that abstracts all the possible paths that can generate the right stack trace.

On the other hand, the state of the stack during the program execution can be modeled as an FSM. Each state of the resulting FSM is the sequence of frames that exist on the stack combined with the address the stack pointer is pointing to. The state machine state changes as the result of a function call or return made by the program. Having the state machine of the stack during the program execution reduces the problem of execution history extraction to finding all the paths starting from the initial state of the FSM to the state of the stack found at the time of the incident.

The only problem with this approach stems from the fact that the state machine representing the states of the stack should be generated from the program source. Supposing that the stack can fit up to n frames (with an average length) and the program has m different function call sites, then the number of the states of the state machine

is of $o(m^n)$. Moreover, in order to define the transitions between the states, one has to consider all the states, one at the time. This is while most of these states are unreachable from the stack's initial state and are out of consideration. Therefore, we developed an algorithm to generate only the relevant states and transactions. The algorithm eventually generate part of the complete stack finite state machine that is accountable for the stack leftover.

Our algorithm starts from the final state of the stack that is the state of the stack at the time the image is taken from RAM. The algorithm traverses the stack state machine backwards based on the transitions allowed in the program PDS model. This means that the algorithm simultaneously traverses both the program PDS model and the stack state machine backwards. While the algorithm traversing the stack state machine, it tries to create what has remaind on the stack. Since the algorithm is traversing the PDS model backwards, when a return transition is traversed, the frame existing after the location where stack pointer is pointing to should represent the call site corresponding to the return transition taken. Additionally, when a return transition is traversed, the stack pointer is increased to point to the newly created frame on top of the stack. In a similar way, when a call transition is traversed, the frame existing at the offset that the stack pointer is pointing to should represent the call site corresponding to the call transition taken. Moreover, when a call transition is traversed, the stack pointer is moved down and the stack frame that previously was pointed to by the stack pointer is freed.

For clarity, we first define the algorithm with the supposition that all the stack frames are of the same length. We prove the soundness, completeness and finiteness of the simplified version of the algorithm and then improve the algorithm to support variable length stack frames. Before we start the formal definition of the algorithm,

as an example, consider the program shown in Figure 5.2. The content of the stack resulting from the execution of the program with inputs 1, 2 is shown in figure 5.2.2.

Figure 5.2.2 shows the first two steps of the algorithm. Note that the transitions with a bar indicate the return from the call site.

To define the algorithm, we represent the state of the stack at each point of execution as the sequence $S = (f_1, ..., f_m, sp, f_{m+1}, ..., f_n)$ where f_is represent the frames on the stack sorted from the bottom of the stack toward the stack limit and sp represents the location that the stack pointer is pointing to that is the top of the stack frame f_m. Suppose that $S = (\Sigma, Q, q_0, \delta, F)$ is the state machine representing the stack of the program. F is the set of final states and in this case contains only one state, which is the state of the stack at the time of imaging the memory. The state q_0 is the initial state which is $(sp, a_1, ..., a_n)$ where $\forall i, a_i = -$. A frame is with the value of $-$ means that it can be overwritten with any arbitrary frame. Also suppose that $D = (P, \Gamma, \sigma)$ is the PDS model of the program.

Suppose that at some point of the execution of the algorithm, the current stack state is $s = (f_i, ..., f_m, sp, f_{m+1}, ..., f_n)$ and the current PDS state is p. The possible backtracking transition at each state are defined using the following rules:

Rule 1. *If* $\exists q, \sigma(q, f_{m+1}) = (p, \epsilon)$ *then* $\delta(r, f'_{m+1}) = s$ *where* $r = (f_i, ..., f_m, f_{m+1}, sp, ..., f_n)$.

Rule 2. $\delta(r, f_m) = s$ *where* $r = (f_i, ..., sp, -, f_{m+1}, ..., f_n)$.

Notice that in the second rule, we are replacing the frame f_m with $-$ which is a way to mark the region as a free region which can be overwritten by any arbitrary value. In the first rule, free regions can match with any frame. After the possible transitions are identified, the same step is performed on the newly entered state if the state has not been expanded before.

```
stack_trace!main + 0x4
  call   stack_trace!ILT+0(?incYAXHZ) (00401050)
stack_trace!inc+0x93
  call     stack_trace!_chkesp        (00408560)
stack_trace!inc+0x31
  call   stack_trace!ILT+0(?incYAXHZ) (00401043)
stack_trace!inc+0x93
  call     stack_trace!_chkesp        (00408560)
                         ...
stack_trace!inc+0x31
  call   stack_trace!ILT+0(?opYAXHZ)  (00401005)
stack_trace!op+0x93
  call     stack_trace!_chkesp        (00408560)
stack_trace!op+0x71
  call     stack_trace!ILT+55(?dYAXZ) (0040103c)
stack_trace!d+0x50
  call     stack_trace!_chkesp (00408560)
stack_trace!d+0x40
  call     stack_trace!ILT+35(?hYAXZ) (00401028)
stack_trace!h+0x2a
  call     stack_trace!_chkesp (00408560)
stack_trace!a+0x1a
  call   stack_trace!ILT+0(?opYAXHZ)  (00401005)
stack_trace!op+0x93
  call     stack_trace!_chkesp (00408560)
stack_trace!op+0x83
  call     stack_trace!ILT+50(?cYAXH) (00401037)
stack_trace!c+0x38
  call     stack_trace!_chkesp (00408560)
stack_trace!c+0x28
  call     stack_trace!ILT+45(?bYAXH) (00401032)
```

Figure 5.6: Extracted stack traces

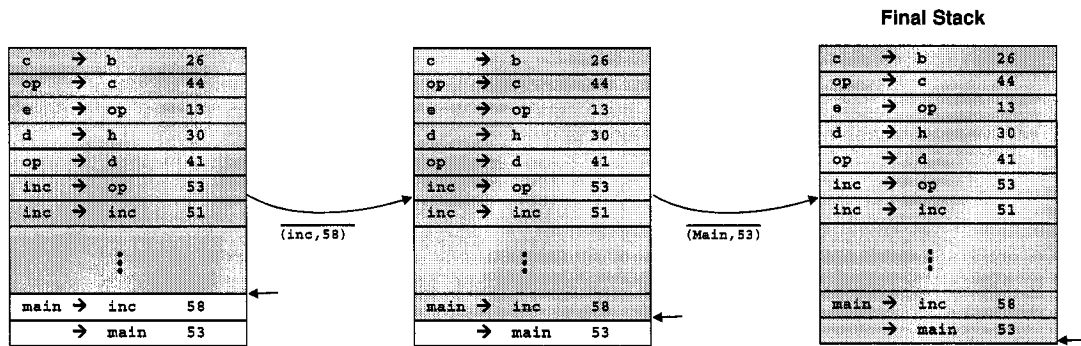


Figure 5.7: Part of the execution of the stack FSM generation algorithm.

The algorithm marks all the transitions from an FSM state that are traversed so that later if the same FSM state is being processed, those transitions are not analyzed again. This is due to the fact depending on which PDS state we are at when we are analyzing an FSM state, different transitions could be possible. The algorithm stops if there is no more state left to be expanded. Figure 5.2.2 shows the pseudo code for the algorithm. The algorithm has a queue that stores the states to be expanded. At each step, after finding the transitions, the originating states (the states to be expanded) are added to the beginning of the queue and then the next state from the queue is processed.

In the following discussion, we prove that the algorithm is sound, complete and finite. To prove the soundness of the algorithm, we show that all function call chains that are accepted by the resulting stack FSM, if executed by the program, will produce the same trace that exist on the snapshot of the stack. To prove the completeness, we show that the algorithm produces all the possible solutions to the problem. To prove that the algorithm is finite, we show that the algorithm eventually stops.

Suppose that C is the set of all functions called by the program, $S = (\Sigma, Q, q_0, \delta, F)$

```
Input: D:The PDS model of the program, t: The final state of the stack FSM., p: The
       current state of D.
Output: S:The FSM that abstracts all the program execution paths that can generate
        the stack residue.
Queue Q;
Q.add(⟨t,p⟩);
while Q is not empty do
    ⟨r,q⟩ = Q.next();
    Use rules 1,2 to get the set of all the possible transitions δ(s,c) → r.
    foreach s do
        if (¬isMarked(s,c)) then
            mark(s,c);
            Q.add(⟨s,u⟩), where u is the originating state of the transition in D that
            changes S from s to r;
        end
    end
end
```

Figure 5.8: The pseudo code of the stack FSM generation algorithm with stack frames of the same size.

is the resulting stack FSM of the execution of the algorithm on the PDS model of the program $D = (P, \Gamma, \sigma)$, the stack leftover at the time of the incident is the sequence $T = (a_1, ..., a_n)$ where $\forall i, a_i \in C \cup \{sp\}$, and $L = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}$ is the language that S accepts.

Theorem 1. *If $t \in L$, then the execution of P with t as input, eventually produces T on the stack.*

To prove this theorem, we define function *trace* as below:

Definition. Suppose that $D = (P, \Gamma, \sigma)$ is a PDS, $\Delta = \Gamma \cup \Upsilon$ where $\Upsilon = \{c' | c \in \Gamma\}$ and $R = \{(r0, ..., rn) | r_i \in P\}$. Function $trace : PDS \times R \rightarrow \Delta^*$ receives as input a PDS D and a sequence of states $U \in R$ and returns sequence S which is produced as below:

for each transition of D from two subsequent states $U_i, U_{i+1} \in P$

if $\sigma(U_i, \epsilon) = (U_{i+1}, f)$ then $S = S.f$.

if $\sigma(U_i, f) = (U_{i+1}, \epsilon)$ then $S = S.f'$.

Proof. According to the algorithm proposed, the configuration of the stack in the initial state of the PDS model corresponds to the initial state of S. Moreover, the only final state of S is the state that represents the configuration of D's stack at the time of the incident. Therefore, if a function call chain is accepted by S, then the execution of D with the function chain as the input, generates the right stack configuration. □

Theorem 2. *If $t \in P^*$ is an execution of D that produces T on the stack, then $trace(D,t) \in L$.*

Proof. We prove this theorem by contradiction. Suppose that $\exists t \in P^*$ such that the execution of D according to t produces the right stack residue but S does not accept $E = trace(D,t)$. This means that at some point during the execution of the D and S, D can make a transition but S cannot make a transition to the stack state which results from the execution of D's transition on its stack. Suppose that this stage happens when D is transiting from state p_i to p_{i+1}. Further suppose that the state of S that it can not make the desired transition from is s_i. Since D is able to make a transition from p_i to p_{i+1} and there are only two types of transitions in D, we have

$$\sigma(p_i, f) = (p_{i+1}, \epsilon) \text{ or } \sigma(p_i, \epsilon) = (p_{i+1}, f').$$

In both cases, according to the production rules, S can make a transition from s_i to the resulting PDS stack state which is a contradiction. □

Theorem 3. *The algorithm to produce the stack FSM from the stack leftover and the program PDS model is finite.*

Proof. To find the FSM that produces function call chains that if executed, will produce the right stack residue, the algorithm traverses the stack states based on the transitions allowed by the program (PDS model). Since the algorithm is marking the states it has

visited before to prevent the reprocessing of a state, in the worst case, the algorithm will traverse all the possible configurations of the stack. Since the length of the stack and the function call sites of the program are both finite, therefore the number of states of the stack is finite and consequently, the algorithm eventually stops. □

5.2.3 Variable length stack frames

Until now, we have supposed that stack frames are of the same length. This is while, depending on the number of arguments and local variables, stack frames can have different sizes. In order to consider variable-length stack frames, we have to first locate the frame boundaries. As it is explained in the next section, our algorithm for finding the stack frame boundaries searches through the stack and identifies the return addresses. Each return address represents a stack frame. Based on the number and types of local variables and arguments of a function, we can specify the offset of the saved return address from the frame boundaries as well as the size of the stack frames. For simplicity, we do not consider other fields of a stack frame in our analysis. Therefore, the stack residues can not be partitioned into clear-cut stack frames. Instead, we think of the stack residues at the beginning of the algorithm as a vector of bytes. The value of byte at some offsets in the vector are known which are the return addresses. As the algorithm backwards through the PDS model and stack FSM model of the program, the stack frames with predefined sizes are created and removed from the stack based on the defined rules. The frame creations and removal rules are defined in a way that the algorithm generates program execution paths whose execution will generate the right values at offsets with known values.

Each state of the stack again is modeled as a sequence $(a_1, ..., a_m, sp, a_{m+1}, ..., a_n)$. However, this time a_is represent bytes of the stack instead of the stack frames. In order

to specify the frame creation and removal rules, we define match relation as follows:

Definition. A stack frame $f = (f_1, ... f_n)$ where f_is represents bytes of the frame, matches the byte array $b = (b_1, ..., b_n)$ if $\forall i, (b_i = -) \vee (b_i = f_i)$ and we write $match(f, b)$.

The *match* relation is used to verify if part of the stack matches the content of a stack frame. In the following, we define the transition generation rules that describe the possible stack state transitions during the execution of the algorithm. To define the substitution rules, suppose that $S = (\Sigma, Q, q_0, \delta, F)$ is the state machine representing the stack and $D = (P, \Gamma, \sigma)$ is the PDS model of the program. Suppose that at some point of execution of the algorithm, the current stack state is $s = (s_1, ..., s_{m-1}, sp, s_m, ..., s_n)$ and the current PDS state is p. The possible state transitions are defined by the following two rules.

Rule 3. *If* $\exists q, f = (f_1, ..., f_k), \sigma(q, f) = (p, \epsilon)$ *and* $match(f, (s_m, ..., s_{m+k-1}))$*, then* $\delta(r, f') = s$ *where* $r = (s_1, ..., s_{m-1}, f_1, ..., f_k, sp, s_{m+k}, ..., s_n)$.

Rule 4. *If* $\exists q, f = (f_1, ..., f_k), \sigma(q, \epsilon) = (p, f)$ *and* $match(f, (s_{m-k}, ..., s_{m-1}))$*, then* $\delta(r, f) = s$ *where* $r = (s_1, ..., s_{m-k-1}, s_p, \underbrace{-, ..., -}_{k}, s_m, ..., s_n)$.

Except for the state transition generation rules, the algorithm for building the partial stack FSM stays the same. Therefore, proof of completeness, finiteness, and soundness are as before.

Function Pointers

In the above discussion we implicitly supposed that the exact destination of a function call can be identified at the compile time. However, the application of the function

pointers can invalidate this supposition by allowing a process to specify the target of a function call dynamically. To overcome this problem, when generating the PDS model, for each function pointer call, we mark the destination as a free state and create a free call and return transition to that state. A call transition that is designated as free means that it can match any function call. Similarly, a free return transition means that a return instruction to any function can match this transition. During the execution of the second phase (finding the executed path), when we see a free return transition (remember that we are traversing the PDS backward), we bind every return statement in the program to this free return once and try to find the possible execution paths by following the execution of the algorithm. It is important to note that this technique can cause a performance degradation to the analysis. However, our analysis showed that most of the function pointer calls are traceable at the compilation time and are determined automatically by IDA Pro [12] as discussed in the next chapter. Moreover, most of the bindings for function pointer calls that can not be determined at compile time prove to be wrong binding in the first and second step of the execution of the algorithm after the assignment. As it is discussed in the future research direction, more optimized approach can be achieved by considering other elements such as the values of CPU registers in the analysis.

5.3 Design and Implementation

In this section, we discuss the design and implementation of the system that was developed based on the elaborated techniques in the previous sections. Simulated examples are provided to demonstrate the effectiveness and short-comings of these techniques.

The Windows physical memory analyzer was developed as a plug-in for a digital investigation framework. This framework was designed as an integrated framework that

consists of a set of forensic analysis plug-ins. The framework was developed to meet the following requirements:

- **Product functionality:** The framework should provide the investigator with necessary analysis required during a digital investigation. This includes but not limited to memory analysis, disk analysis, log analysis and email analysis. The framework should be focused on analysis rather than acquisition of evidence. This is due to the fact that many commercial and open-source solutions for acquisition of evidence exist while the required analysis functionality are non-existing in the existing solution.

- **User characteristics:** The target users of this software are mainly forensic agents responsible for investigating through the incidence of an intrusion, finding enough evidence, analyzing the evidences and generating judicially approved reports. The framework should provide a simple to use user interface that does not require users to have an indept knowledge of the automatic analysis performed in the framework in order to use the result of the analysis.

- **Operating environment:** The software is aimed to be portable both on Windows and Unix based operating systems.

- **Design and Implementation Constraints:** The software should be written in Java, whenever required, uses native objects to get advantage from other programming languages capabilities that lack in java such as certain system calls. The design should be extendable both in terms of evidence resources and analysis techniques meaning that the user should be able to introduce new resources for evidence gathering phase and new analyzer plug-ins for the analysis phase.

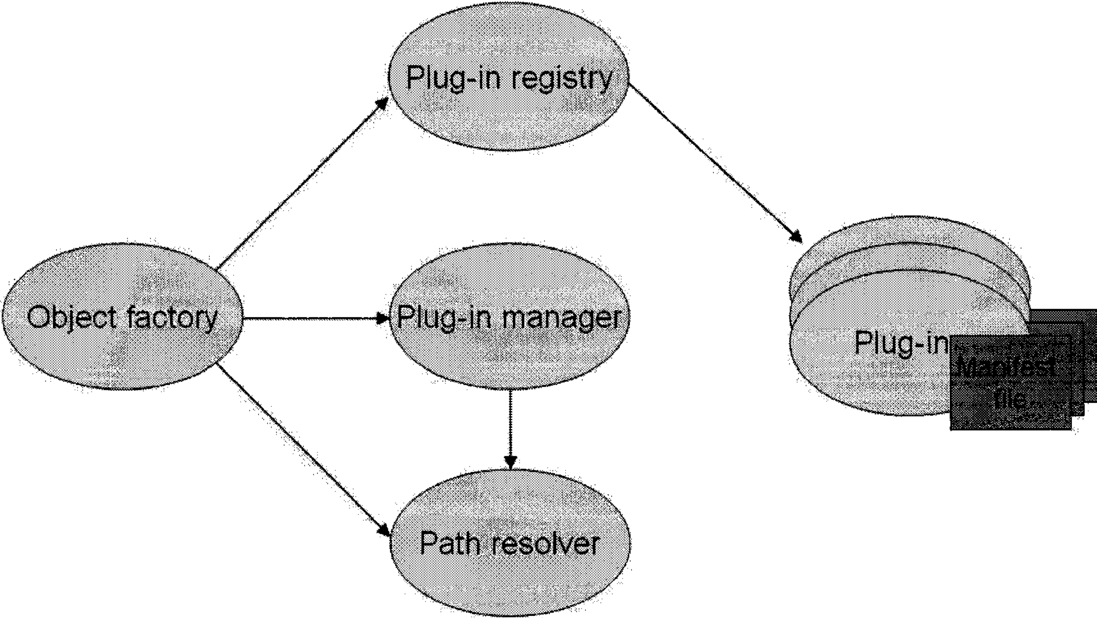
Based on the above requirement, Java Server Framework (JPF) [3] was chosen as the

implementation framework for our forensic toolkit. JPF provides a runtime engine that dynamically discovers and loads the plug-ins. A plug-in is a structured component that contributes code and resources to the system and describes them in a structured way. These plug-ins can further define extension points, well-defined method hooks that can be extended by other plug-ins. JPF maintains a registry of available plug-ins and the functions they provide. Figure 5.9 shows the high-level architecture of this framework. The main components of JPF are described below:

- A plug-in is a structured component that describes itself to JPF using a plug-in manifest. The plug-in manifest is an XML file which contains all the information needed by the JPF framework about each plug-in.
- PluginRegistry is a set of interfaces that abstract meta-information about plug-ins and plug-in fragments.
- PluginManager is the runtime system of the Framework. The main responsibility of the manager is to activate (load plug-in code and call the plug-in initializer class) plug-ins upon client code requests and manage inter plug-in dependencies. It is also possible to deactivate plug-ins during the life of the application. This feature may help to reduce application resources requirements.
- The ObjectFactory class allows application developers to easily create base JPF objects: PluginRegistry, PluginManager and PathResolver.

The Windows physical memory plug-in was developed as one of the plug-ins in our digital forensic investigation framework. This plug-in consists of five main components: Structure manager, Windows in-memory structure classes, file extraction, ThreadStackParser, and stack trace analyzer. Most of the in-memory structure classes were introduced in

Figure 5.9: JAVA Plug-in Framework (JPF) overall architecture.



chapter 2. The java implementation of these classes includes necessary information to correctly fill and use these structures.

Each in-memory structure inherits from `WinStructure` class that contains the required functions for filling structures from raw data. These functions includes `fillListEntry`, `fillSingleListEntry`, `fillListEntryHead`, `fillSingleListEntryHead`, `readStructFromByteData`, `readObjectFromByteData`, etc. The name of these functions describes their functionality. These function receive the class object of the structure that should be extracted and call a function named `extractUniqueStruct` passing it the class object and the beginning address of the structure in the raw image. Using java reflection, this function acquires the `fillStruct` function that is defined by each Windows in-memory structure class. Function `fillStruct` fills all of the fields of the structure from raw data and returns the object.

One important task of `extractUniqueStruct` function as its name suggests is to verify if the structure has previously been extracted and if it has, it retrieves the extracted object rather than creating a new object. This is due to the fact that most of Windows in-memory structure instances are pointed by several structures and therefore the analyzer could have previously extracted them from memory. All `WinStructure` objects share a static table called `ExtractedObjects`. This table is essentially contains the mapping between the starting physical address of the structure and the object representing the structure. Every time that `extractUniqueStruct` is called with the starting address of a structure to be extracted, it calls the `searchObject` function that uses `ExtractedObjects` table to search for a structure that has the same starting address and type as the one to be extracted. It is important to notice that in addition to the starting physical address, `searchObject` verifies the type of the stored object and the structure to be extracted. This is because some structures could start at the

same physical address but be of different types. An example that this situation exists is when filling _KThread and _EThread. If you remember, _KThread is the first member of _EThread and therefore both of these structures start at the same physical address in memory. By comparing the type of the stored object with the type of the structure to be filled in, the right object can be retrieved.

The third component of Windows memory analysis plug-in is the file extractor. File extraction functionality is implemented as part of the functionality of EProcess and FileObject classes. Two techniques are used to extract files. For executable files, the COFF header is used to extract the executable from the memory. For other types of files, the SharedCacheMaps were used. These two techniques were discussed in detail in chapter two. It is important to note that the content of the file in memory exists in the form of a file when the file is opened as a memory mapped file by a process and this is the only time that it can be extracted from the memory as it exists on disk.

The fourth component of the Windows memory analysis plug-in is the ThreadStackParser. This class is responsible for parsing the thread user-land and kernel-land stacks. It performs the parsing based on the techniques discussed in chapter two. After parsing the stack, it exports the stack to a file that can be used with the stack trace analysis component to discover the executed paths. Since in the current proposed approach for stack trace analysis, only function calls are considered and other values on the stack are not included in the analysis, The exported information includes the starting offset of each stack frame and the return address that is stored in the frame. This information is enough for the fifth component, the stack trace analyzer, to perform the analysis as described in chapter four.

The stack trace analyzer is implemented as a plug-in for IDA Pro [12]. IDA Pro is one of the most popular disassemblers that provides an SDK for developing further

analysis on executables that are disassembled in this environment. Please notice that this plug-in is executed in IDA Pro environment and therefore, requires the information that is extracted from the physical memory image to be imported into it. This information consists of two data: the extracted executable, and the stack information. This plug-in is written in c. As discussed in chapter four, the analysis is performed in three steps. In the first step, the analyzer creates the control flow graphs (CFG) from the executables. In this step, the stack trace analyzer scans each function from the first instruction to its last instruction looking for jump, call and return instructions. When one of these instruction is seen, a new state is created and a transition is added from the current analysis state to the newly created state and the analysis is continued.

After the CFGs of all of the functions of the executable are created, the next step is to combine these CFGs into a PDS. This task is performed by `generate_pds_from_local_models` function declared in `pds_model.cpp`. For each CFG, this function examines all of the outgoing transitions and if the instruction assigned to the transition is a call instruction, a new transition is created from the state under examination to the starting state of the CFG of the called function. In addition to this, a new transition is created from each of the final states (return states) of the called function to the destination state of the old transition. Next, the old transition is removed from the state under examination.

After the PDS model is created, the last step of the analysis is performed according to algorithm 5.2.2 discussed in chapter four. This algorithm is implemented in `process_fsm_state` function. In order the for the function to perform correctly, it needs to know the starting state for the analysis. This state is the state at which the program was executing when the image was taken from memory and is detectable using the instruction pointer register. This is possible because each instruction in the executable is

mapped uniquely to a single state in the created PDS model. As the algorithm traverses the PDS model, it creates the final solution in the form of a state machine which is implemented as a linked list of structures of type `fsm_state`. As an output, the program creates a graphical view of this final state machine. The graphical view is created in udg format that is viewable by uDraw [4].

5.4 Experimental Result

As part of the integrated forensic investigation framework, the Windows physical memory forensic analyzer plug-in was developed with the required portability and usability functionalities in mind. As shown in Figure 5.10, the Windows physical memory forensic analyzer consists of two sub-panels. The left sub-panel contains the detailed values of in-memory structures grouped by process name in the form of a jtree. The right sub-panel contains six tabs. Each tab contains specific category of information. The **process** tab contains information about the process such as process executable, name, user land and kernel land time, etc. Information about loaded dlls are listed in the **dll** tab. This information consists of the name, path, base address, entry point, and size of the image. The **thread** tab contains information about the threads of the process selected in the left tab. this tab lists all of the threads of the process and by selecting each thread from the list, the investigator is able to see various information about the thread. Moreover, the export stack button, allows the user to export the stacks of the selected thread to a file that can be later used as the input to the stack trace analyzer plug-in developed in IDA Pro. The **object** tab, contains the parsed handle table of the process that contains information about all of the objects created by the process. This information differs based on the type of the object and can be viewed by double clicking on the object entity in the table. **Environment** tab contains the environment

Table 5.1: The stack analysis result for the different processes.

Program	Funcs#	PDS states#	FSM states#	Recovery%
dcfldd.exe if=test	200	3101	98	25
notepad.exe	88	2179	50	55
nc.exe -l -p 80	168	2913	55	100
nc.exe localhost 80	168	2913	89	100
psexec.exe -s cmd	395	4321	200	83

information for each process. The last tab is the extracted files information that contains the list of the files opened by the process. The user is able to export or view these files by double clicking on them. Double clicking on the executables opens them with disasm.exe program [5]. Figures 5.10 to 5.15 show the described plug-in tabs.

In order to verify the effectiveness of our algorithm in recovering the execution history of executables, several softwares were executed using IDAPro debugger while the function call tracing option was enabled. In executing each software, several actions were performed and then an image was taken from the memory. This image was analyzed using our Windows memory analyzer framework and the program executable, its stack(s) and the instruction pointer was extracted and handed to our stack analyzer. The stack analyzer output was compared with the trace that was acquired from executing the software using IDApro with function call tracing option enabled. The result of this analysis is shown in table 5.1.

As it can be deduced from table 5.1, the percentage of the execution path that could be retrieved in each scenario varies. This is due to the fact that the effectiveness of our technique depends on several factors including:

- The number of functions that are called by the application.
- The number of different execution paths that exist in the application.

- The amount of time the application was running before the memory image was acquired.
- The distribution of calls to system calls in application.
- The number of iterative operations.

Figure 5.16 shows the output of the stack analyzer for the execution of nc.exe.

In this chapter, we presented an approach to analyze the extracted information from Windows physical memory in order to discover the execution path that was executed by each thread. In this approach, we model the execution of a process as a PDS model using static analysis techniques and correlate this global execution model with the stack frames that are left on the stack after the execution of the process. The result of this correlation is the partial execution path that could have been executed by the process and if executed would leave the right stack trace on the stack. This result is modeled with a finite state machine. The algorithm that we proposed for correlation of stack traces and the process model is able to find all of the possible answers. We also proved that this algorithm terminates after processing infinite number of states. Empirical results were provided with emphasis on the fact that the effectiveness of our approach depends on the implementation details of the program as well as the amount of traces that are left on the stack.

Figure 5.10: Windows physical memory analyzer - general tab.

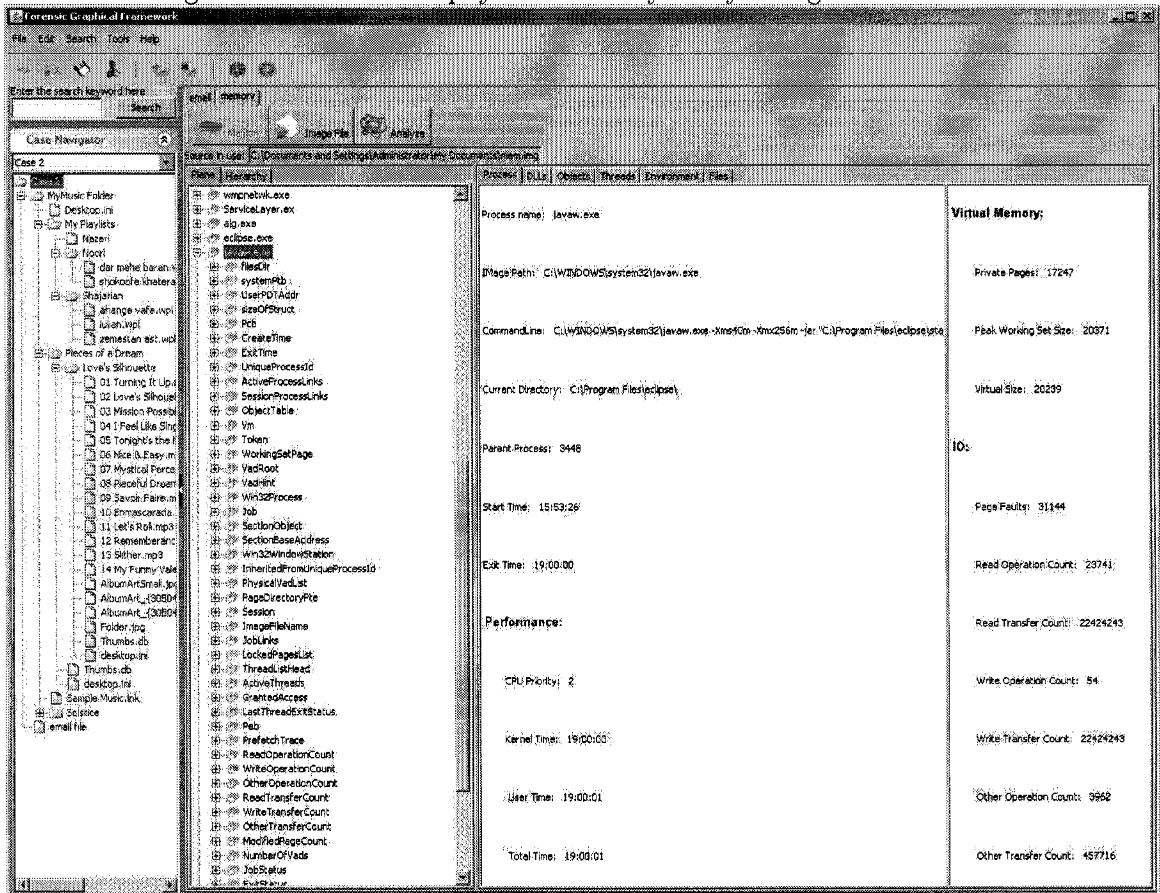


Figure 5.11: Windows physical memory analyzer - thread tab.

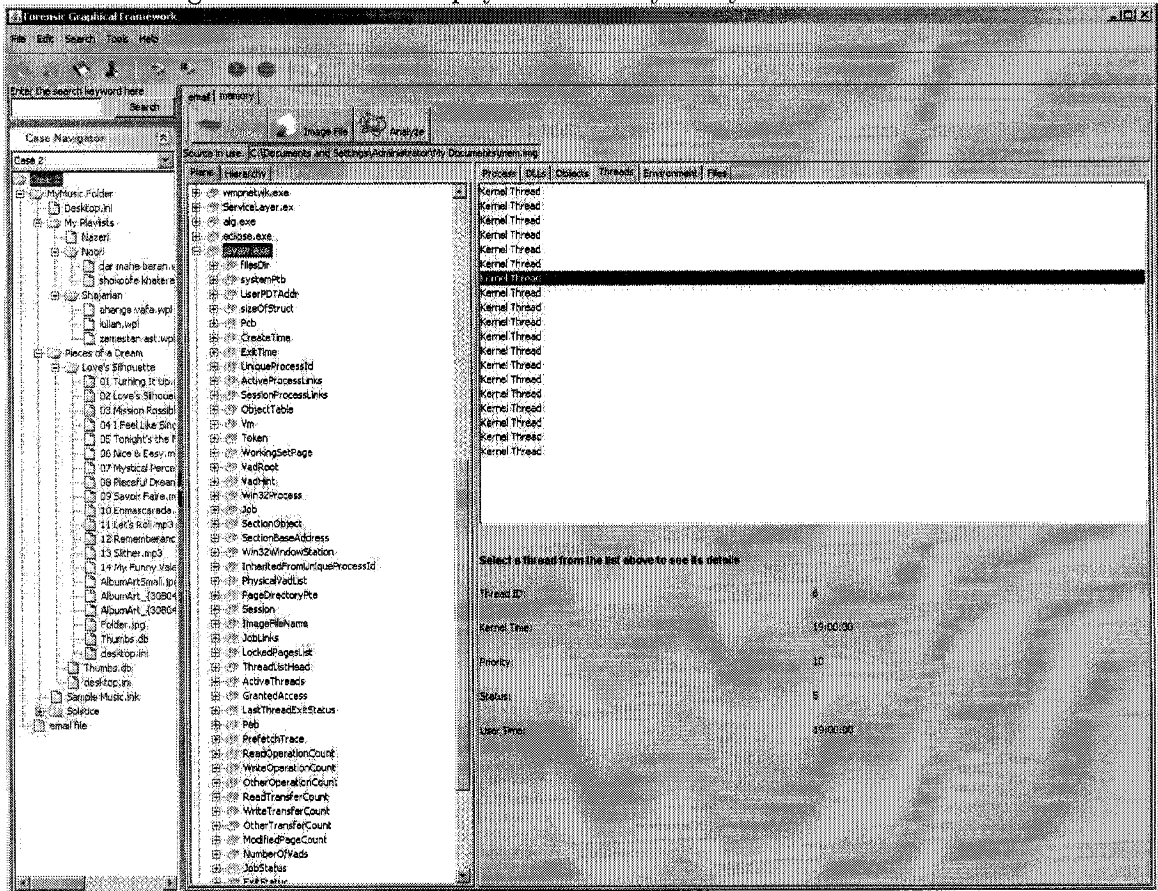


Figure 5.12: Windows physical memory analyzer - object tab.

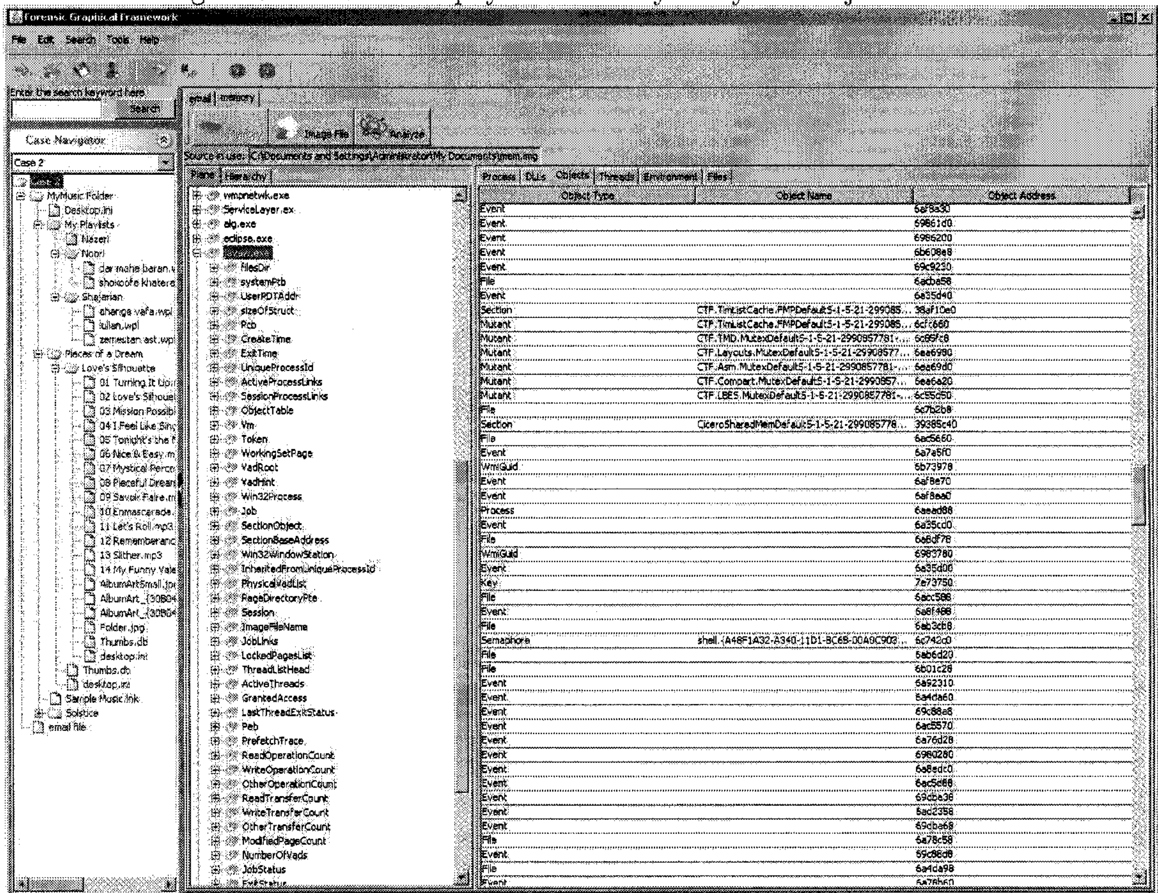


Figure 5.13: Windows physical memory analyzer - environment tab.

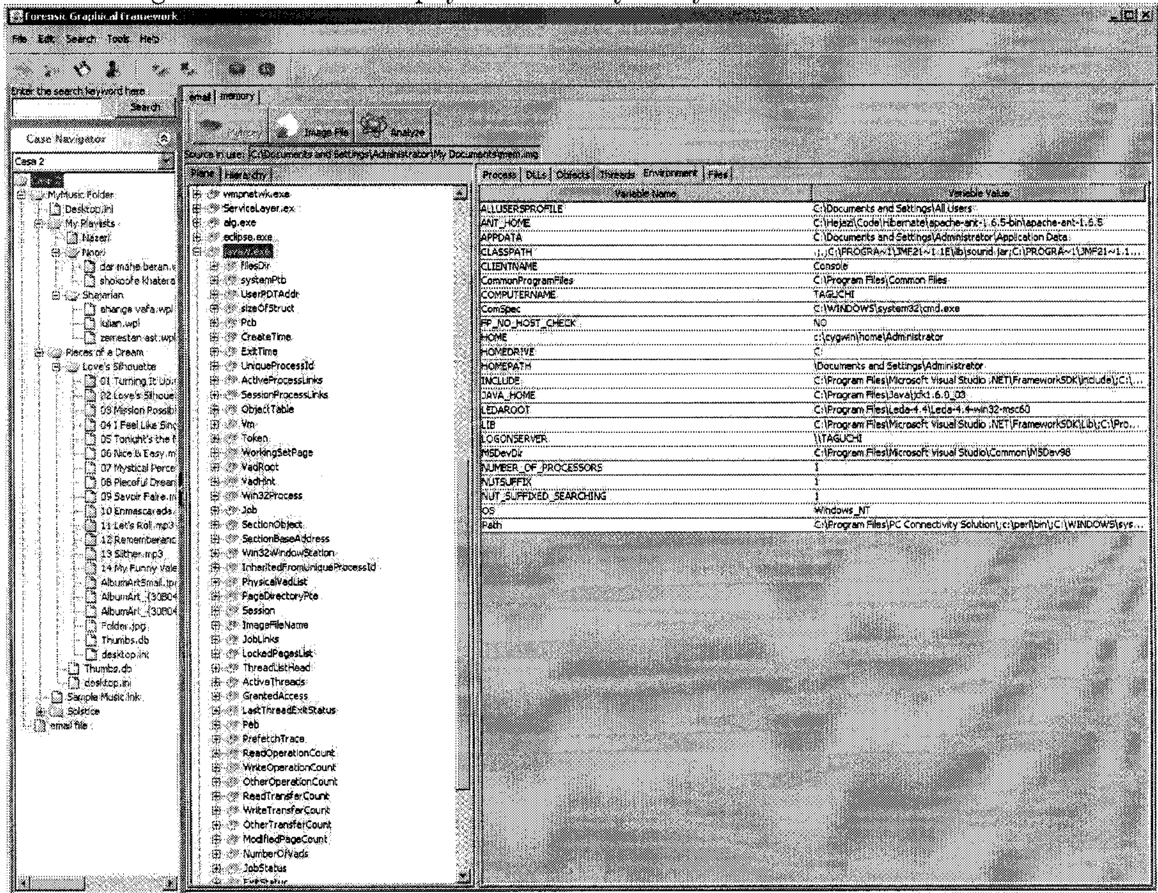


Figure 5.14: Windows physical memory analyzer - files tab.

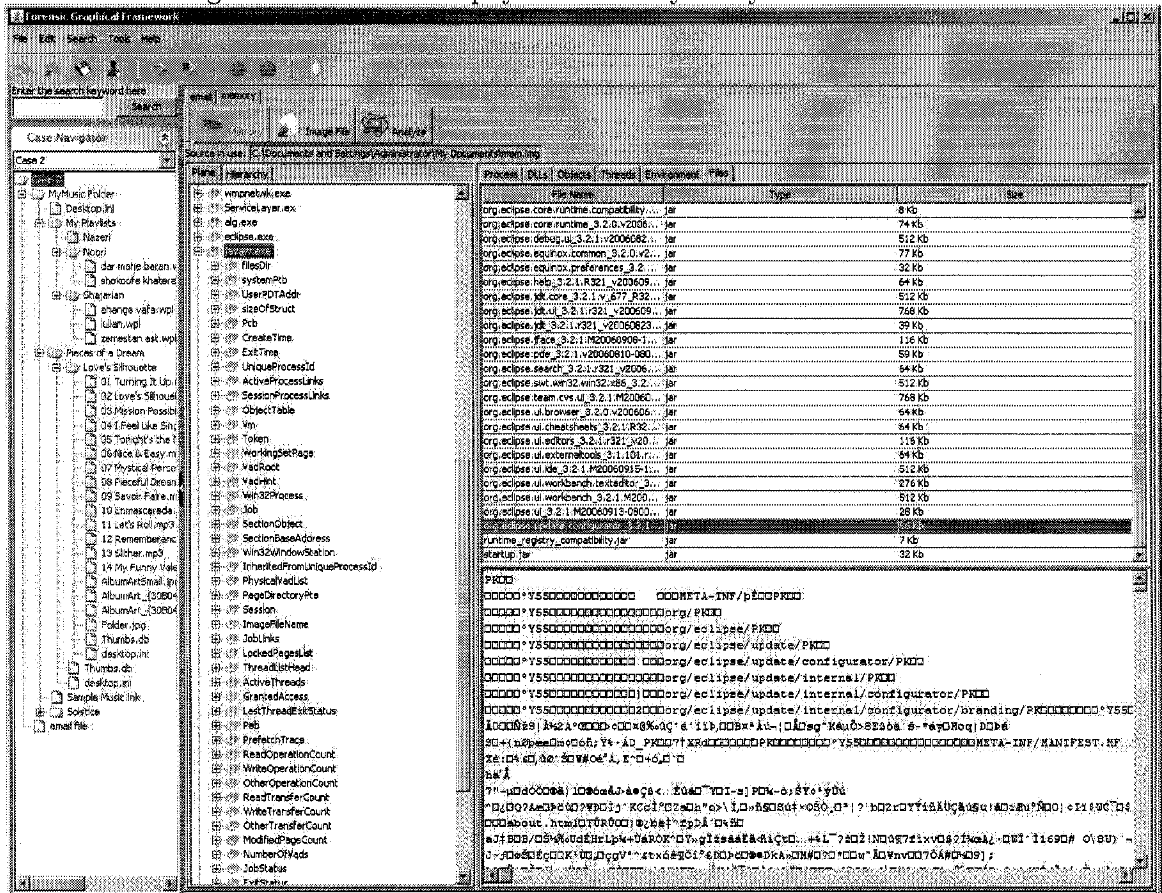


Figure 5.15: Windows physical memory analyzer - dlls tab.

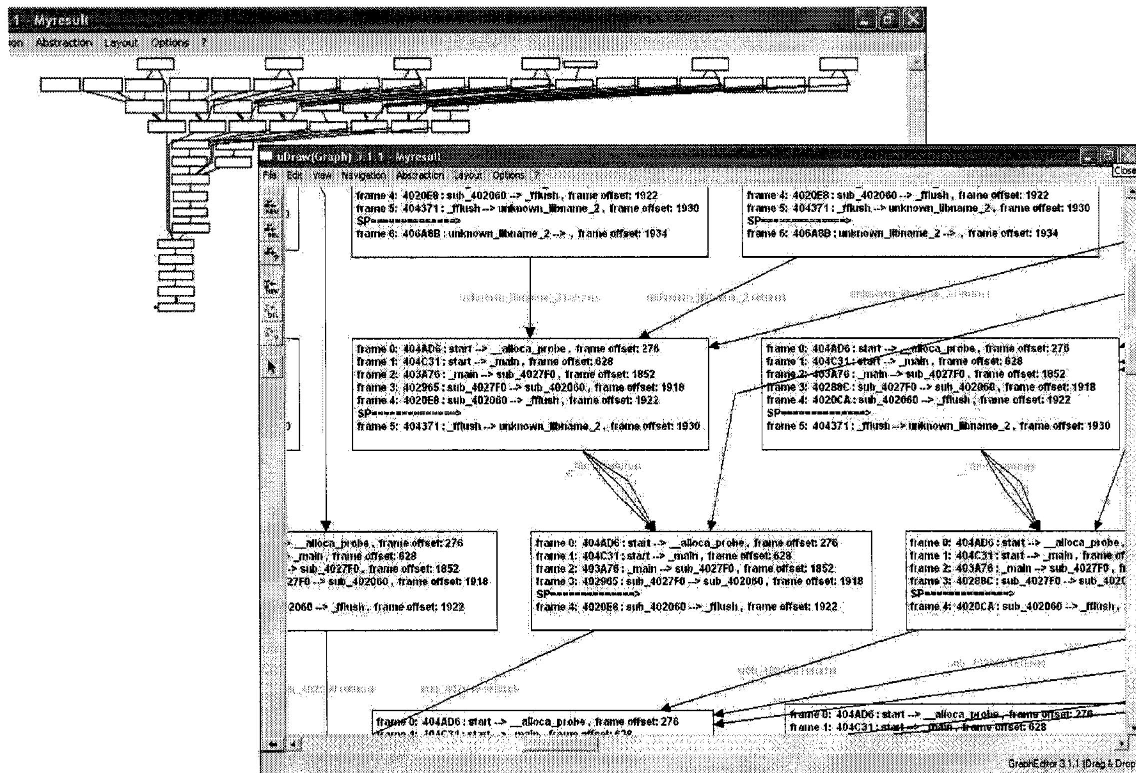


Figure 5.16: The stack analyzer output for nc.exe.

Chapter 6

Conclusion

In this thesis, physical memory was presented as an important evidence that can be useful in a digital investigation. Through a detailed investigation of physical memory, the analyst can discover the chain of events that occurred at the time of the incident. However, this important source of information is often neglected in the course of many investigations. This is mostly due to the complexities that are involved in accurate analysis of this source of evidence that requires a detailed understanding of operating system functionalities. The volatile nature of this media also adds to the complexities involved in the forensic analysis of physical memory. Despite the difficulties involved in the analysis of physical memory, it often contains information about the incident that can not be acquired from other sources and therefore it is necessary to acquire and investigate the physical memory during the course of the investigation.

This thesis shows the importance and advantage of the acquisition and analysis of physical memory as a source of evidence in the course of the investigation. The main focus of this thesis was on Windows physical memory. However, most of the introduced techniques are applicable to other operating systems. The techniques that

were introduced in this thesis can be classified into two different categories.

The first category of the discussed techniques are mainly focused on the extraction of forensic related information. This includes information about processes, files, threads, registry keys, environment variables, etc. For each of these evidential items, the in-memory structures that are managed by the operating system were introduced and their application in extracting the related information was elaborated. Our experimental results showed that a lot of this information still exist in memory long after the process terminates or finishes the related task. In this thesis, our study was limited to the in-memory structures that are directly managed by the main components of the operating system. This is while, many functionalities of the operating systems are performed in the context of other processes such as lsass.exe, svchost.exe, and csrss.exe. These processes whose operation is essential for proper functioning of Windows operating system manage several in-memory structures that may include forensically pertinent information. However, little documentation exists on the functionality and operational details of these processes. The author believes that a detailed study of theses processes can reveal techniques for forensic analysis of physical memory that can provide the investigators with valuable information.

The second category of memory forensic analysis techniques that were detailed in this thesis involves an approach to reconstruct the execution of processes that were executing at the time the image was taken from the memory. These techniques consist of two phases. The first phase is to model the execution of the process by analyzing the process executable. The second phase is to try to find an execution path in the process executable model that generates an execution trace that matches the existing trace in the image. The first phase is conducted in three steps; First, the executable is parsed to extract the control flow graph of each function in the executable. Next each control

flow graph is turned into a finite state machine. Last, the state machines representing each function are combined into a pushdown system that represents the whole program. During the second phase, the pushdown system of the executable is traversed based on the information on the stack to produce all possible execution path of the program in the form of a state machine.

In the second category of memory forensic analysis, the focus of the analysis is to retrieve history rather than extracting leftover structures and object from raw data. In this analysis, we included only the stack leftovers, and the executable code. The result of this analysis entails the execution path that was executed by the process in terms of function call and return. Although this result can provide valuable insights and evidence, more accurate and useful results can be extracted if the argument values that are passed to these functions are determined. The inclusion of arguments in the analysis results requires dynamic analysis of program heap, stack, and registers. The author strongly believes that the inclusion of heap and registry related information and the correlation of these information with other existing evidence such as network logs can reveal many information that are of outmost value in a digital investigation.

Another possible application of the introduced techniques is in debugging and software maintenance procedures. Using the elaborated techniques, one can reproduce the chain of events that happened at the time of the incident. Therefore, the crash information such as memory dumps and system snapshots can be analyzed using the technique introduced in this thesis to point out the faulting execution paths.

Bibliography

[1] Tenable Network Security. http://www.nessus.org/. Visited on: March 16, 2007.

[2] 2005 digital forensic research workshop (dfrws), memory analysis challenge. http://www.dfrws.org/2005/challenge/index.html, 2005. Visited on: March 16, 2007.

[3] Java plug-in framework project (jpf). http://jpf.sourceforge.net/, 2007. Visited on: April 9th, 2008.

[4] udraw, visualization solution. www.informatik.uni-bremen.de/uDrawGraph, 2007. Visited on: April 9th, 2008.

[5] Windows disassembler. http://www.supershareware.com/disassembler.html, 2007. Visited on: April 9th, 2008.

[6] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Monica S. Lam. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, volume 1. Addison Wesley, 2 edition, August 2006.

[7] V. Baryamureeba and F. Tushabe. The enhanced digital investigation process model. *Digital Forensic Research Workshop*, 2004.

[8] M. Burdach. Digital forensics of the physical memory. http://forensic.seccure.net/pdf/, 2005. Visited on: March 16, 2007.

[9] Jamie Butler. Dkom (direct kernel object manipulation). http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf, 2004. Black Hat Europe 2004. Visited on: April 9th, 2008.

[10] Betz C. Memparser. http://www.dfrws.org/2005/challenge/memparser.html, 2005. Visited on: March 16, 2007.

[11] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. Technical Report UCB//CSD-02-1197, UC Berkeley, 2002., 2002.

[12] DataRescue. The ida pro disassembler and debugger. http://www.datarescue.com/ida.htm. Visited on: March 16, 2007.

[13] Eldad Eilam. *Reversing: Secrets of Reverse Engineering.* Wiley, April 2005.

[14] Jonathon T. Giffin. *Model based intrusion detection system design and evaluation.* PhD thesis, http://www.cc.gatech.edu/~giffin/papers/phd06/Gif06.pdf, 2006. Visited on: March 16, 2007.

[15] P. Gladyshev and A. Patel. Finite state machine approach to digital event reconstruction. *Digital Investigation Journal*, 1(2), 2004.

[16] P. Gladyshev and A. Patel. Formalising event time bounding in digital investigations. *Digital Investigation Journal*, 4(2), 2005.

[17] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley Professional, July 2005.

[18] C. Hosmer. Time lining computer evidence. http://www.wetstonetech.com/f/timelining.pdf, 1998. Visited on: March 16, 2007.

[19] W. G. Kruse II and J. G. Heiser. *Computer Forensics: Incident Response Essentials.* Addison-Wesley, Boston, MA, 2002.

[20] George M. Garner Jr. Kntlist. http://www.dfrws.org/2005/challenge, 2005. Visited on: March 16, 2007.

[21] R. S. Greenfield Jr. A. J. Marcella. *CYBER FORENSICS: A Field Manual for Collecting, Examining, and Preserving Evidence of Computer Crimes.* Auerbach Publications, 2002.

[22] P. Mell K. Scarfone. Guide to intrusion detection and prevention systems (idps). Special Publication 800-94, National Institution of Standards and Technology, Gaithersburg, MD 20899-8930, February 2007.

[23] J. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation Journal*, January 2007.

[24] R. Leigland and A. W. Krings. A formalization of digital forensics. *Digital Investigation Journal*, 3(2), 2004.

[25] Microsoft. Debugging tools for windows sdk. http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx. Visited on: April 9th, 2008.

[26] K. Monroe and D. Bailey. System base-lining: A forensic perspective. http://ftimes.sourceforge.net/Files/Papers/baselining.pdf, 2003. Visited on: March 16, 2007.

[27] A. Walters N. Petroni. Fatkit: A framework for the extraction and analysis of digital forensics data from volatile system memory.

[28] C. Peikari and A. Chuvakin. *Security Warrior.* O'Reilly, Sebastopol, CA, 2004.

[29] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 1. Microsoft Press, 4th edition, December 2004.

[30] E. P. Markatos M. Polychronakis S. Antonatos, K. G. Anagnostakis. Performance analysis of content matching intrusion detection systems. *International Symposium on Applications and the Internet*, 1:208 – 215, 2004.

[31] Sven Schreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook.* Addison-Wesley Professional, May 2001.

[32] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06)*, pages 10–16, September 2006.

[33] T. Stallard and K. Levitt. Automated Analysis for Digital Forensic Science: Semantic Integrity Checking. In *19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2003.

[34] P. Stephenson. Modeling of post-incident root cause analysis. *International Journal of Digital Evidence*, 2(2), 2003.

[35] US-CERT. United state computer emergency readiness team. `http://www.us-cert.gov/federal/incidentDefinition.html`, 2007. Visited on: April 9th, 2008.

[36] Dimitri van Heesch. Doxygen project. `http://www.doxygen.org`. Visited on: April 9th, 2008.

[37] Ric Vieler. *Professional Rootkits (Programmer to Programmer)*, volume 1. Wrox, 1st edition, March 2007.

[38] A. Walters and N. Petroni. Volatools: Integrating volatile memory forensics into the digital investigation process. *Black Hat DC 2007*, February 2007.

Appendices

APPENDIX I - Internal Windows Structures

JOBs

```
kd> dt _EJOB
   +0x000 Event            : _KEVENT
   +0x010 JobLinks         : _LIST_ENTRY
   +0x018 ProcessListHead  : _LIST_ENTRY
   +0x020 JobLock          : _ERESOURCE
   +0x058 TotalUserTime    : _LARGE_INTEGER
   +0x060 TotalKernelTime  : _LARGE_INTEGER
   +0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER
   +0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER
   +0x078 TotalPageFaultCount : Uint4B
   +0x07c TotalProcesses   : Uint4B
   +0x080 ActiveProcesses  : Uint4B
   +0x084 TotalTerminatedProcesses : Uint4B
   +0x088 PerProcessUserTimeLimit : _LARGE_INTEGER
   +0x090 PerJobUserTimeLimit : _LARGE_INTEGER
   +0x098 LimitFlags       : Uint4B
   +0x09c MinimumWorkingSetSize : Uint4B
   +0x0a0 MaximumWorkingSetSize : Uint4B
   +0x0a4 ActiveProcessLimit : Uint4B
   +0x0a8 Affinity         : Uint4B
   +0x0ac PriorityClass    : UChar
   +0x0b0 UIRestrictionsClass : Uint4B
```

```
+0x0b4 SecurityLimitFlags : Uint4B
+0x0b8 Token              : Ptr32 Void
+0x0bc Filter             : Ptr32 _PS_JOB_TOKEN_FILTER
+0x0c0 EndOfJobTimeAction : Uint4B
+0x0c4 CompletionPort     : Ptr32 Void
+0x0c8 CompletionKey      : Ptr32 Void
+0x0cc SessionId          : Uint4B
+0x0d0 SchedulingClass    : Uint4B
+0x0d8 ReadOperationCount : Uint8B
+0x0e0 WriteOperationCount : Uint8B
+0x0e8 OtherOperationCount : Uint8B
+0x0f0 ReadTransferCount : Uint8B
+0x0f8 WriteTransferCount : Uint8B
+0x100 OtherTransferCount : Uint8B
+0x108 IoInfo             : _IO_COUNTERS
+0x138 ProcessMemoryLimit : Uint4B
+0x13c JobMemoryLimit     : Uint4B
+0x140 PeakProcessMemoryUsed : Uint4B
+0x144 PeakJobMemoryUsed : Uint4B
+0x148 CurrentJobMemoryUsed : Uint4B
+0x14c MemoryLimitsLock : _FAST_MUTEX
+0x16c JobSetLinks        : _LIST_ENTRY
+0x174 MemberLevel        : Uint4B
+0x178 JobFlags           : Uint4B
```

Structure _EPROCESS

```
kd> dt _EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
```

```
+0x11c VadRoot          : Ptr32 Void
+0x120 VadHint          : Ptr32 Void
+0x124 CloneRoot        : Ptr32 Void
+0x128 NumberOfPrivatePages : Uint4B
+0x12c NumberOfLockedPages : Uint4B
+0x130 Win32Process     : Ptr32 Void
+0x134 Job              : Ptr32 _EJOB
+0x138 SectionObject    : Ptr32 Void
+0x13c SectionBaseAddress : Ptr32 Void
+0x140 QuotaBlock       :
            Ptr32 _EPROCESS_QUOTA_BLOCK
+0x144 WorkingSetWatch  :
            Ptr32 _PAGEFAULT_HISTORY
+0x148 Win32WindowStation : Ptr32 Void
+0x14c InheritedFromUniqueProcessId:
            Ptr32 Void
+0x150 LdtInformation   : Ptr32 Void
+0x154 VadFreeHint      : Ptr32 Void
+0x158 VdmObjects       : Ptr32 Void
+0x15c DeviceMap        : Ptr32 Void
+0x160 PhysicalVadList  : _LIST_ENTRY
+0x168 PageDirectoryPte :
            _HARDWARE_PTE_X86
+0x168 Filler           : Uint8B
+0x170 Session          : Ptr32 Void
```

```
+0x174 ImageFileName    : [16] UChar
+0x184 JobLinks         : _LIST_ENTRY
+0x18c LockedPagesList  : Ptr32 Void
+0x190 ThreadListHead   : _LIST_ENTRY
+0x198 SecurityPort     : Ptr32 Void
+0x19c PaeTop           : Ptr32 Void
+0x1a0 ActiveThreads    : Uint4B
+0x1a4 GrantedAccess    : Uint4B
+0x1a8 DefaultHardErrorProcessing : Uint4B
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 Peb              : Ptr32 _PEB
+0x1b4 PrefetchTrace    : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER
+0x1c0 WriteOperationCount : _LARGE_INTEGER
+0x1c8 OtherOperationCount : _LARGE_INTEGER
+0x1d0 ReadTransferCount : _LARGE_INTEGER
+0x1d8 WriteTransferCount : _LARGE_INTEGER
+0x1e0 OtherTransferCount : _LARGE_INTEGER
+0x1e8 CommitChargeLimit : Uint4B
+0x1ec CommitChargePeak : Uint4B
+0x1f0 AweInfo          : Ptr32 Void
+0x1f4 SeAuditProcessCreationInfo :
             _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm               : _MMSUPPORT
+0x238 LastFaultCount   : Uint4B
```

```
+0x23c ModifiedPageCount : Uint4B
+0x240 NumberOfVads      : Uint4B
+0x244 JobStatus         : Uint4B
+0x248 Flags             : Uint4B
+0x248 CreateReported    : Pos 0, 1 Bit
+0x248 NoDebugInherit    : Pos 1, 1 Bit
+0x248 ProcessExiting    : Pos 2, 1 Bit
+0x248 ProcessDelete     : Pos 3, 1 Bit
+0x248 Wow64SplitPages   : Pos 4, 1 Bit
+0x248 VmDeleted         : Pos 5, 1 Bit
+0x248 OutswapEnabled    : Pos 6, 1 Bit
+0x248 Outswapped        : Pos 7, 1 Bit
+0x248 ForkFailed        : Pos 8, 1 Bit
+0x248 HasPhysicalVad    : Pos 9, 1 Bit
+0x248 AddressSpaceInitialized : Pos 10, 2 Bits
+0x248 SetTimerResolution : Pos 12, 1 Bit
+0x248 BreakOnTermination : Pos 13, 1 Bit
+0x248 SessionCreationUnderway : Pos 14, 1 Bit
+0x248 WriteWatch         : Pos 15, 1 Bit
+0x248 ProcessInSession : Pos 16, 1 Bit
+0x248 OverrideAddressSpace : Pos 17, 1 Bit
+0x248 HasAddressSpace  : Pos 18, 1 Bit
+0x248 LaunchPrefetched : Pos 19, 1 Bit
+0x248 InjectInpageErrors : Pos 20, 1 Bit
+0x248 VmTopDown          : Pos 21, 1 Bit
```

```
+0x248 Unused3          : Pos 22, 1 Bit
+0x248 Unused4          : Pos 23, 1 Bit
+0x248 VdmAllowed       : Pos 24, 1 Bit
+0x248 Unused           : Pos 25, 5 Bits
+0x248 Unused1          : Pos 30, 1 Bit
+0x248 Unused2          : Pos 31, 1 Bit
+0x24c ExitStatus       : Int4B
+0x250 NextPageColor    : Uint2B
+0x252 SubSystemMinorVersion : UChar
+0x253 SubSystemMajorVersion : UChar
+0x252 SubSystemVersion : Uint2B
+0x254 PriorityClass    : UChar
+0x255 WorkingSetAcquiredUnsafe : UChar
+0x258 Cookie           : Uint4B
```

Structure _PEB

```
kd> dt _PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged    : UChar
+0x003 SpareBool        : UChar
+0x004 Mutant           : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr              : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData    : Ptr32 Void
+0x018 ProcessHeap      : Ptr32 Void
+0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : Uint4B
+0x02c KernelCallbackTable : Ptr32 Void
+0x030 SystemReserved   : [1] Uint4B
+0x034 AtlThunkSListPtr32 : Uint4B
+0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
+0x03c TlsExpansionCounter : Uint4B
+0x040 TlsBitmap        : Ptr32 Void
+0x044 TlsBitmapBits    : [2] Uint4B
+0x04c ReadOnlySharedMemoryBase : Ptr32 Void
+0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
```

```
+0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
+0x058 AnsiCodePageData : Ptr32 Void
+0x05c OemCodePageData  : Ptr32 Void
+0x060 UnicodeCaseTableData : Ptr32 Void
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag     : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER
+0x078 HeapSegmentReserve : Uint4B
+0x07c HeapSegmentCommit : Uint4B
+0x080 HeapDeCommitTotalFreeThreshold : Uint4B
+0x084 HeapDeCommitFreeBlockThreshold : Uint4B
+0x088 NumberOfHeaps    : Uint4B
+0x08c MaximumNumberOfHeaps : Uint4B
+0x090 ProcessHeaps     : Ptr32 Ptr32 Void
+0x094 GdiSharedHandleTable : Ptr32 Void
+0x098 ProcessStarterHelper : Ptr32 Void
+0x09c GdiDCAttributeList : Uint4B
+0x0a0 LoaderLock       : Ptr32 Void
+0x0a4 OSMajorVersion   : Uint4B
+0x0a8 OSMinorVersion   : Uint4B
+0x0ac OSBuildNumber    : Uint2B
+0x0ae OSCSDVersion     : Uint2B
+0x0b0 OSPlatformId     : Uint4B
+0x0b4 ImageSubsystem   : Uint4B
+0x0b8 ImageSubsystemMajorVersion : Uint4B
```

```
+0x0bc ImageSubsystemMinorVersion : Uint4B
+0x0c0 ImageProcessAffinityMask : Uint4B
+0x0c4 GdiHandleBuffer  : [34] Uint4B
+0x14c PostProcessInitRoutine : Ptr32
+0x150 TlsExpansionBitmap : Ptr32 Void
+0x154 TlsExpansionBitmapBits : [32] Uint4B
+0x1d4 SessionId        : Uint4B
+0x1d8 AppCompatFlags   : _ULARGE_INTEGER
+0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
+0x1e8 pShimData        : Ptr32 Void
+0x1ec AppCompatInfo    : Ptr32 Void
+0x1f0 CSDVersion       : _UNICODE_STRING
+0x1f8 ActivationContextData : Ptr32 Void
+0x1fc ProcessAssemblyStorageMap : Ptr32 Void
+0x200 SystemDefaultActivationContextData : Ptr32 Void
+0x204 SystemAssemblyStorageMap : Ptr32 Void
+0x208 MinimumStackCommit : Uint4B
```

Structure _rtl_user_process_parameters

```
dt _rtl_user_process_parameters


0x000 MaximumLength    : Uint4B
0x004 Length           : Uint4B
0x008 Flags            : Uint4B
0x00c DebugFlags       : Uint4B
0x010 ConsoleHandle    : Ptr32 Void
0x014 ConsoleFlags     : Uint4B
0x018 StandardInput    : Ptr32 Void
0x01c StandardOutput   : Ptr32 Void
0x020 StandardError    : Ptr32 Void
0x024 CurrentDirectory : _CURDIR
0x030 DllPath          : _UNICODE_STRING
0x038 ImagePathName    : _UNICODE_STRING
0x040 CommandLine      : _UNICODE_STRING
0x048 Environment      : Ptr32 Void
0x04c StartingX        : Uint4B
0x050 StartingY        : Uint4B
0x054 CountX           : Uint4B
0x058 CountY           : Uint4B
0x05c CountCharsX      : Uint4B
0x060 CountCharsY      : Uint4B
0x064 FillAttribute    : Uint4B
0x068 WindowFlags      : Uint4B
```

```
0x06c ShowWindowFlags   : Uint4B
0x070 WindowTitle       : _UNICODE_STRING
0x078 DesktopInfo       : _UNICODE_STRING
0x080 ShellInfo         : _UNICODE_STRING
0x088 RuntimeData       : _UNICODE_STRING
0x090 CurrentDirectores : [32] _RTL_DRIVE_LETTER_CURDIR
```

Structure _KPROCESS

```
kd> dt _KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
```

```
+0x067 DisableBoost     : UChar
+0x068 PowerState       : UChar
+0x069 DisableQuantum   : UChar
+0x06a IdealNode        : UChar
+0x06b Flags            : _KEXECUTE_OPTIONS
+0x06b ExecuteOptions   : UChar
```

Structure _ETHREAD

```
kd> dt _ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
```

```
+0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
+0x220 ThreadsProcess   : Ptr32 _EPROCESS
+0x224 StartAddress     : Ptr32 Void
+0x228 Win32StartAddress : Ptr32 Void
+0x228 LpcReceivedMessageId : Uint4B
+0x22c ThreadListEntry  : _LIST_ENTRY
+0x234 RundownProtect   : _EX_RUNDOWN_REF
+0x238 ThreadLock       : _EX_PUSH_LOCK
+0x23c LpcReplyMessageId : Uint4B
+0x240 ReadClusterSize  : Uint4B
+0x244 GrantedAccess    : Uint4B
+0x248 CrossThreadFlags : Uint4B
+0x248 Terminated       : Pos 0, 1 Bit
+0x248 DeadThread       : Pos 1, 1 Bit
+0x248 HideFromDebugger : Pos 2, 1 Bit
+0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
+0x248 SystemThread     : Pos 4, 1 Bit
+0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
+0x248 BreakOnTermination : Pos 6, 1 Bit
+0x248 SkipCreationMsg  : Pos 7, 1 Bit
+0x248 SkipTerminationMsg : Pos 8, 1 Bit
+0x24c SameThreadPassiveFlags : Uint4B
+0x24c ActiveExWorker   : Pos 0, 1 Bit
+0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
+0x24c MemoryMaker      : Pos 2, 1 Bit
```

```
+0x250 SameThreadApcFlags : Uint4B
+0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
+0x250 LpcExitThreadCalled : Pos 1, 1 Bit
+0x250 AddressSpaceOwner : Pos 2, 1 Bit
+0x254 ForwardClusterOnly : UChar
+0x255 DisablePageFaultClustering : UChar
```

Structure _KTHREAD

```
kd> dt _KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
   +0x050 IdleSwapBlock    : UChar
   +0x051 Spare0           : [3] UChar
   +0x054 WaitStatus       : Int4B
   +0x058 WaitIrql         : UChar
   +0x059 WaitMode         : Char
   +0x05a WaitNext         : UChar
   +0x05b WaitReason       : UChar
```

```
+0x05c WaitBlockList      : Ptr32 _KWAIT_BLOCK
+0x060 WaitListEntry      : _LIST_ENTRY
+0x060 SwapListEntry      : _SINGLE_LIST_ENTRY
+0x068 WaitTime           : Uint4B
+0x06c BasePriority       : Char
+0x06d DecrementCount     : UChar
+0x06e PriorityDecrement : Char
+0x06f Quantum            : Char
+0x070 WaitBlock          : [4] _KWAIT_BLOCK
+0x0d0 LegoData           : Ptr32 Void
+0x0d4 KernelApcDisable : Uint4B
+0x0d8 UserAffinity       : Uint4B
+0x0dc SystemAffinityActive : UChar
+0x0dd PowerState         : UChar
+0x0de NpxIrql            : UChar
+0x0df InitialNode        : UChar
+0x0e0 ServiceTable       : Ptr32 Void
+0x0e4 Queue              : Ptr32 _KQUEUE
+0x0e8 ApcQueueLock       : Uint4B
+0x0f0 Timer              : _KTIMER
+0x118 QueueListEntry     : _LIST_ENTRY
+0x120 SoftAffinity       : Uint4B
+0x124 Affinity           : Uint4B
+0x128 Preempted          : UChar
+0x129 ProcessReadyQueue : UChar
```

```
+0x12a KernelStackResident : UChar
+0x12b NextProcessor    : UChar
+0x12c CallbackStack    : Ptr32 Void
+0x130 Win32Thread      : Ptr32 Void
+0x134 TrapFrame        : Ptr32 _KTRAP_FRAME
+0x138 ApcStatePointer  : [2] Ptr32 _KAPC_STATE
+0x140 PreviousMode     : Char
+0x141 EnableStackSwap  : UChar
+0x142 LargeStack       : UChar
+0x143 ResourceIndex    : UChar
+0x144 KernelTime       : Uint4B
+0x148 UserTime         : Uint4B
+0x14c SavedApcState    : _KAPC_STATE
+0x164 Alertable        : UChar
+0x165 ApcStateIndex    : UChar
+0x166 ApcQueueable     : UChar
+0x167 AutoAlignment    : UChar
+0x168 StackBase        : Ptr32 Void
+0x16c SuspendApc       : _KAPC
+0x19c SuspendSemaphore : _KSEMAPHORE
+0x1b0 ThreadListEntry  : _LIST_ENTRY
+0x1b8 FreezeCount      : Char
+0x1b9 SuspendCount     : Char
+0x1ba IdealProcessor   : UChar
+0x1bb DisableBoost     : UChar
```

Structure _TOKEN

```
kd> dt _TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr32 _ERESOURCE
   +0x038 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x040 ModifiedId       : _LUID
   +0x048 SessionId        : Uint4B
   +0x04c UserAndGroupCount : Uint4B
   +0x050 RestrictedSidCount : Uint4B
   +0x054 PrivilegeCount   : Uint4B
   +0x058 VariableLength   : Uint4B
   +0x05c DynamicCharged   : Uint4B
   +0x060 DynamicAvailable : Uint4B
   +0x064 DefaultOwnerIndex : Uint4B
   +0x068 UserAndGroups    : Ptr32 _SID_AND_ATTRIBUTES
   +0x06c RestrictedSids   : Ptr32 _SID_AND_ATTRIBUTES
   +0x070 PrimaryGroup     : Ptr32 Void
   +0x074 Privileges       : Ptr32 _LUID_AND_ATTRIBUTES
   +0x078 DynamicPart      : Ptr32 Uint4B
   +0x07c DefaultDacl      : Ptr32 _ACL
   +0x080 TokenType        : _TOKEN_TYPE
```

```
+0x084 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
+0x088 TokenFlags         : Uint4B
+0x08c TokenInUse         : UChar
+0x090 ProxyData          : Ptr32 _SECURITY_TOKEN_PROXY_DATA
+0x094 AuditData          : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
+0x098 OriginatingLogonSession : _LUID
+0x0a0 VariablePart       : Uint4B
```

Structure _KDDEBUGGER_DATA64

```
struct _KDDEBUGGER_DATA64 {

    DBGKD_DEBUG_DATA_HEADER64 Header;
    ULONG64   KernBase;
    ULONG64   BreakpointWithStatus;
    ULONG64   SavedContext;
    USHORT  ThCallbackStack;
    USHORT  NextCallback;
    USHORT  FramePointer;
    USHORT  PaeEnabled:1;
    ULONG64   KiCallUserMode;
    ULONG64   KeUserCallbackDispatcher;   // address in ntdll
    ULONG64   PsLoadedModuleList;
    ULONG64   PsActiveProcessHead;
    ULONG64   PspCidTable;
    ULONG64   ExpSystemResourcesList;
```

```
ULONG64    ExpPagedPoolDescriptor;
ULONG64    ExpNumberOfPagedPools;
ULONG64    KeTimeIncrement;
ULONG64    KeBugCheckCallbackListHead;
ULONG64    KiBugcheckData;
ULONG64    IopErrorLogListHead;
ULONG64    ObpRootDirectoryObject;
ULONG64    ObpTypeObjectType;
ULONG64    MmSystemCacheStart;
ULONG64    MmSystemCacheEnd;
ULONG64    MmSystemCacheWs;
ULONG64    MmPfnDatabase;
ULONG64    MmSystemPtesStart;
ULONG64    MmSystemPtesEnd;
ULONG64    MmSubsectionBase;
ULONG64    MmNumberOfPagingFiles;
ULONG64    MmLowestPhysicalPage;
ULONG64    MmHighestPhysicalPage;
ULONG64    MmNumberOfPhysicalPages;
ULONG64    MmMaximumNonPagedPoolInBytes;
ULONG64    MmNonPagedSystemStart;
ULONG64    MmNonPagedPoolStart;
ULONG64    MmNonPagedPoolEnd;
ULONG64    MmPagedPoolStart;
ULONG64    MmPagedPoolEnd;
```

```
ULONG64    MmPagedPoolInformation;
ULONG64    MmPageSize;
ULONG64    MmSizeOfPagedPoolInBytes;
ULONG64    MmTotalCommitLimit;
ULONG64    MmTotalCommittedPages;
ULONG64    MmSharedCommit;
ULONG64    MmDriverCommit;
ULONG64    MmProcessCommit;
ULONG64    MmPagedPoolCommit;
ULONG64    MmExtendedCommit;
ULONG64    MmZeroedPageListHead;
ULONG64    MmFreePageListHead;
ULONG64    MmStandbyPageListHead;
ULONG64    MmModifiedPageListHead;
ULONG64    MmModifiedNoWritePageListHead;
ULONG64    MmAvailablePages;
ULONG64    MmResidentAvailablePages;
ULONG64    PoolTrackTable;
ULONG64    NonPagedPoolDescriptor;
ULONG64    MmHighestUserAddress;
ULONG64    MmSystemRangeStart;
ULONG64    MmUserProbeAddress;
ULONG64    KdPrintCircularBuffer;
ULONG64    KdPrintCircularBufferEnd;
ULONG64    KdPrintWritePointer;
```

```
    ULONG64    KdPrintRolloverCount;
    ULONG64    MmLoadedUserImageList;
    ULONG64    NtBuildLab;
    ULONG64    KiNormalSystemCall;
    ULONG64    KiProcessorBlock;
    ULONG64    MmUnloadedDrivers;
    ULONG64    MmLastUnloadedDriver;
    ULONG64    MmTriageActionTaken;
    ULONG64    MmSpecialPoolTag;
    ULONG64    KernelVerifier;
    ULONG64    MmVerifierData;
    ULONG64    MmAllocatedNonPagedPool;
    ULONG64    MmPeakCommitment;
    ULONG64    MmTotalCommitLimitMaximum;
    ULONG64    CmNtCSDVersion;
    ULONG64    MmPhysicalMemoryBlock;
    ULONG64    MmSessionBase;
    ULONG64    MmSessionSize;
    ULONG64    MmSystemParentTablePage;
}
```