

Windows Memory Analysis

Solutions in this chapter:

- Dumping Physical Memory
- Analyzing a Physical Memory Dump
- Collecting Process Memory

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

In Chapter 1, “Live Response: Collecting Volatile Data,” we discussed collecting volatile data from a live, running Windows system. From the Order of Volatility listed in RFC 3227, we saw that the first item of volatile data that should be collected during live-response activities is the contents of physical memory, commonly referred to as RAM. Although the specifics of collecting particular parts of volatile memory, such as network connections or running processes, has been known for some time and discussed pretty extensively, the issue of collecting, parsing, and analyzing the entire contents of physical memory is a relatively new endeavor. This field of research has really opened up in the past year or two, beginning in the summer of 2005, at least from a public perspective.

The most important question that needs to be answered at this point is, “Why?” Why would you want to collect the contents of RAM? How is doing this useful, how is it important, and what would you miss if you didn’t? Until now, some investigators have collected the contents of RAM in hope of finding something that they wouldn’t find on the hard drive during a post-mortem analysis—specifically, passwords. Programs will prompt the user for a password, and if the dialog box has disappeared from view, the most likely place to find that password is in memory. Malware analysts will look to memory in dealing with encrypted or obfuscated malware, because when the malware is launched, it will be decrypted in memory. More and more, malware is obfuscated in such a way that static, offline analysis is extremely difficult at best. However, if the malware were allowed to execute, it would exist in memory in a decrypted state, making it easier to analyze what the malware does. Finally, rootkits will hide processes, files, Registry keys, and even network connections from view by the tools we usually use to enumerate these items, but by analyzing the contents of RAM we can find what’s been hidden. We can also find information about processes that have since exited.

As this area of analysis grows and more investigators pursue RAM as a viable source of valuable information and evidence, it will become easier to extract information from RAM and correlate that to what is found during the post-mortem forensic analysis.

A Brief History

In the past, the “analysis” of physical memory dumps has consisted of running strings or *grep* against the “image” file, looking for passwords, IP addresses, e-mail addresses, or other strings that could give the analyst an investigative lead. The drawback method of “analysis” is that it is difficult to tie the information you find to a distinct

process. Was the IP address that was discovered part of the case, or was it actually used by some other process? How about that word that looks like a password? Is it the password that an attacker uses to access a Trojan on the system, or is it part of an instant messaging (IM) conversation?

Being able to perform some kind of analysis of a dump of physical memory has been high on the wish lists of many within the forensic community for some time. Others (such as myself) have recognized the need for easily accessible tools and frameworks for retrieving physical memory dumps and analyzing their contents.

In the summer of 2005, the Digital Forensic Research Workshop (DFRWS)¹ issued a “memory analysis challenge” “to motivate discourse, research, and tool development” in this area. Anyone was invited to download the two files containing dumps of physical memory (the dumps were obtained using a modified copy of `dd.exe` available on the Helix² 1.6 distribution) and answer questions based on the scenario provided at the Web site. Chris Betz and the duo of George M. Garner, Jr., and Robert-Jan Mora were selected as the joint winners of the challenge, providing excellent write-ups illustrating their methodologies and displaying the results of the tools they developed. Unfortunately, these tools were not made publicly available.

In the year following the challenge, others continued this research or conducted their own, following their own avenues. Andreas Schuster³ began releasing portions of his research on the English version of his blog, together with the format of the `EPROCESS` and `ETHREAD` structures from various versions of Windows, including Windows 2000 and XP. Joe Stewart posted a Perl script called `pmodump.pl` as part of the TRUMAN Project,⁴ which allows you to extract the memory used by a process from a dump of memory (important for malware analysis). Mariusz Burdach has released information regarding memory analysis (initially for Linux systems but then later specifically for Windows systems) to include a presentation at the BlackHat Federal 2006 conference.⁵ Jesse Kornblum has offered several insights in the area of memory analysis to include determining the original operating system from the contents of the memory dump. During the summer of 2006, Tim Vidas,⁶ a Senior Research Fellow at Nebraska University, released `procloc.pl`, a Perl script to locate processes in RAM dumps as well as crash dumps.

Dumping Physical Memory

So, how do you go about collecting the contents of physical memory? Several ways have been identified, each with its own strengths and weaknesses. The goal of this chapter is to provide an understanding of the various options available as well as the technical aspects associated with each option. This way, as a first responder or investigator, you’ll make educated choices regarding which option is most suitable, taking

the business needs of the client (or victim) into account along with infrastructure concerns.

Hardware Devices

In February 2004, the *Digital Investigation Journal* published a paper by Brian Carrier and Joe Grand, of Grand Idea Studio, Inc.,⁷ titled, “A Hardware-Based Memory Acquisition Procedure for Digital Investigations.” In the paper, Brian and Joe presented the concept for a hardware expansion card dubbed Tribble (possibly a reference to that memorable *Star Trek* episode) that could be used to retrieve the contents of physical memory to an external storage device. This would allow an investigator to retrieve the volatile memory from the system without introducing any new code nor relying on potentially untrusted code to perform the extraction. In the paper, the authors stated that they had built a proof-of-concept Tribble device, designed as a PCI expansion card that could be plugged into a PC bus. Other hardware devices are available that allow you to capture the contents of physical memory and are largely intended for debugging hardware systems. These devices may also be used for forensics.

As illustrated in the DFRWS 2005 Memory Challenge, one of the limitations of a software-based approach to retrieving volatile memory is that the program the investigator is using has to be loaded into memory. Subsequently, particularly on Windows systems, the program may (depending on its design) rely on untrusted code or libraries (DLLs) that have been subverted by the attacker. Let’s examine the pros and cons of such a device:

- **Pros** Hardware devices such as the Tribble are unobtrusive and easily accessible. Dumping the contents of physical memory in this manner introduces no new or additional software to the system, minimizing the chances of data being obscured in some manner.
- **Cons** The primary limitation of using the hardware-based approach is that the hardware needs to be installed prior to the incident. At this point the Tribble devices are not widely available. Other hardware devices *are* available and intended for hardware debugging, but they must still be installed prior to an incident to be of use.

FireWire

Due to technical specifics of FireWire devices and protocols, there is a possibility that with the right software, an investigator can collect the contents of physical

memory from a system. FireWire devices use direct memory access (DMA), meaning that they can access system memory without having to go through the CPU. These devices can read from and/or write to memory at much faster rates than systems that do not use DMA. The investigator would need a controller device that contains the appropriate software and is capable of writing a command into a specific area of the FireWire device's memory space. Memory mapping is performed in hardware without going through the host operating system, allowing for high-speed, low-latency data transfers.

Adam Boileau⁸ came up with a way to extract physical memory from a system using Linux and Python.⁹ The software used for this collection method runs on Linux and relies on support for the `/dev/raw1394` device as well as Adam's `pythonraw1394` library, the `libraw1394` library, and Swig (software that makes C/C++ header files accessible to other languages by generating wrapper code). In his demonstrations, Adam even included the use of a tool that will collect the contents of RAM from a Windows system with the screen locked, then parse out the password, after which Adam logs into the system.

Jon Evans, an officer with the Gwent police department in the United Kingdom, has installed Adam's tools and successfully collected the contents of physical memory from Windows systems as well as from various versions of Linux. As part of his master's thesis, Jon wrote an overview on how to install, set up, and use Adam's tools on several different Linux platforms, including Knoppix v.5.01, Gentoo Linux 2.6.17, and BackTrack, from remote-exploit.org. Once all the necessary packages (including Adam's tools) have been downloaded and installed, Jon then walks through the process of identifying FireWire ports and then tricking the target Windows system into "thinking" that the Linux system is an iPod by using the Linux `romtool` command to load a data file containing the Control Status Register (CSR) for an iPod (the CSR file is provided with Adam's tools). Here are the pros and cons of this approach:

- **Pros** Many systems available today have FireWire /IEEE 1394 interfaces built right into the motherboards. Also, code has been released for directly accessing physical memory on Linux and Mac OS systems.
- **Cons** Arne Vidstrom has pointed out some technical issues¹⁰ regarding the way dumping the contents of physical memory over FireWire can result in a hang or in parts of memory being missed. George M. Garner, Jr., noted in an e-mail exchange on a mailing list in October 2006 that in limited testing, there were notable differences in important offsets between a RAM dump collected using the FireWire technique and one collected using George's own software. This difference could only be explained as an error

in the collection method. Furthermore, this method has caused Blue Screens of Death (BSODs, discussed further in a moment) on some target Windows systems, possibly due to the nature of the FireWire hardware on the system.

Crash Dumps

At one point, we've all seen crash dumps; in most cases they manifest themselves as an infamous Blue Screen of Death¹¹ (BSOD). In most cases they're an annoyance, if not indicative of a much larger issue. However, if you want to obtain a pristine, untainted copy of the content of RAM from a Windows system, perhaps the only way to do that is to generate a full crash dump. The reason for this is that when a crash dump occurs, the system state is frozen and the contents of RAM (along with about 4Kb of header information) are written to the disk. This preserves the state of the system and ensures that no alterations are made to the system, beginning at the time the crash dump was initiated.

This information can be extremely valuable to an investigator. First of all, the contents of the crash dump are a snapshot of the system, frozen in time. I have been involved in several investigations during which crash dumps have been found and used to determine root causes, such as avenues of infection or compromise. Second, Microsoft provides tools for analyzing crash dumps—not only in the debugging tools¹² but also the Kernel Memory Space Analyzer¹³ tools, which are based on the debugging tools.

Sounds like a good deal, doesn't it? After all, other than having a 1GB file written to the hard drive, possibly overwriting evidence (and not really minimizing the impact of our investigation on the system), it is a good deal, right? Under some circumstances, it could be ... or you might be willing to accept that condition, depending on the circumstances. However, there are still a couple of stumbling blocks. First, not all systems generate full crash dumps by default. Second, by default, Windows systems do not generate crash dumps on command.

The first issue is relatively simple to deal with, according to MS KnowledgeBase (KB) article Q254649.¹⁴ This KB article lists the three types of crash dump: small (64KB), kernel, and complete crash dumps. What we're looking for is the complete crash dump because it contains the complete contents of RAM. The KB article also states that Windows 2000 Pro and Windows XP (both Pro and Home) will generate small crash dumps, and Windows 2003 (all versions) will generate full crash dumps. My experience with Windows Vista RC1 is that it will generate small crash dumps, by default.

Along the same lines, MS KB article Q274598¹⁵ states that complete crash dumps are not available on systems with more than 2GB of RAM. According to the article, this is largely due to the space requirements (i.e., for systems with complete crash dumps enabled, the page file must be as large as the contents of RAM + 1MB) as well as the time it will take to complete the crash dump process.

MS KB article Q307973¹⁶ describes how to set the full range of system failure and recovery options. These settings are more for system administrators and IT managers who are setting up and configuring systems before an incident occurs, but the Registry key settings can provide some significant clues for an investigator. For example, if the system was configured (by default or otherwise) to generate a complete crash dump and the administrator reported seeing the BSoD, the investigator should expect to see a complete crash dump file on the system.

NOTE

Investigators must be extremely careful when working with crash dump files, particularly from systems that process but do not necessarily store sensitive data. In some cases crash dumps have occurred on systems that processed information such as credit card number, individual's Social Security number, or the like, and the crash dumps have been found to contain that information. Even though the programmers specifically wrote the application so that no sensitive personal information was saved locally on the system, a crash dump wrote the contents of memory to the hard drive.

So, let's say that our system failure and recovery configuration options are set ahead of time (as part of the configuration policies for the systems) to perform a full crash dump. How does the investigator "encourage" a system to perform a crash dump on command, when it's needed? It turns out that there's a Registry key (see KB article Q244139¹⁷) that can be set to cause a crash dump when the right Ctrl key is held down and the Scroll Lock key is pressed *twice*. However, once this key is set, the system must be rebooted for the setting to take effect. Let's look at the pros and cons of this technique:

- **Pros** Dumping memory via a crash dump is perhaps the only technically accurate method for creating an image of the contents of RAM. This is due to the fact that when the *KeBugCheck* API function is called, the entire system is halted and the contents of RAM are written to the page file, after

which they are written to a file on the system hard drive. Further, Microsoft provides debugging tools as well as the Kernel Memory Space Analyzer¹⁸. (which consists of an engine, plugins, and UI) for analyzing crash dump files.

- **Cons** Some Windows systems do not generate full crash dumps by default (Vista RC1, for example; I had an issue with a driver when I first installed Vista RC1 and I would get BSoDs whenever I attempted to shut down the machine, which resulted in minidump files). In addition, modifying a system to accept the keystroke sequence to create a crash dump requires a reboot and must be done ahead of time to be used effectively for incident response. Even if this configuration change has been made, the crash dump process will still create a file equal in size to physical memory on the hard drive. To do so, as stated in KB article Q274598, the page file must be configured to be equal to at least the size of physical memory plus 1 MB. This is an additional step that must be corrected to use this method of capturing the contents of physical memory; it's one that is not often followed.

**TIP**

A support article¹⁹ located at the Citrix Web site provides a methodology for using LiveKD.exe²⁰ and the Microsoft Debugging Tools to generate a full kernel dump of physical memory. Once LiveKD.exe is launched, the command `.dump /f <filepath>` is used to generate the dump file. The support article does include the caveat that RAM dumps generated in this manner can be inconsistent due to the fact that the dump can take a considerable amount of time and that the system is live and continues to run during the memory dump.

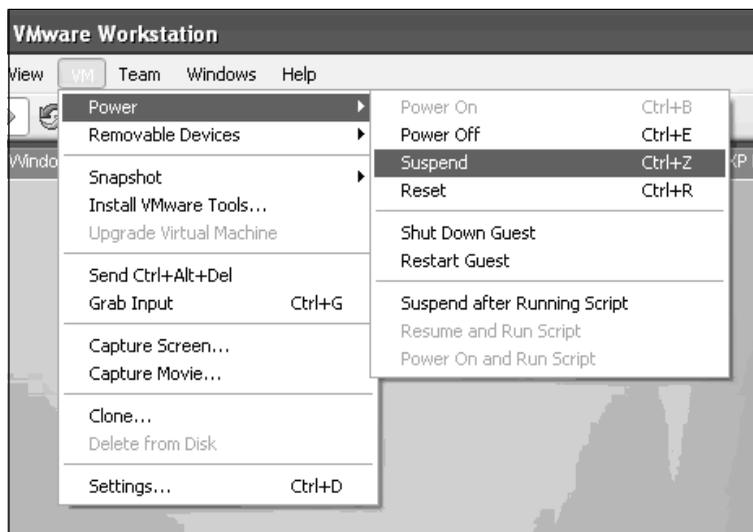
Virtualization

VMware is a popular virtualization product (VMware Workstation 5.5.2 was used extensively in this book) that, for one thing, allows the creation of pseudo-networks utilizing the hardware of a single system. This capability has many benefits. For example, you can set up a guest operating system and create a snapshot of that system once you have it configured to your needs. From there, you can perform all manner of testing, including installing and monitoring malware, and you will always

be able to revert to the snapshot, beginning anew. I have even seen active production systems run from VMware sessions.

When you're running a VMware session, you can suspend that session, freezing it temporarily. Figure 3.1 illustrates the menu items for suspending a VMware session.

Figure 3.1 Menu Items for Suspending a Session in VMware Workstation 5.5.2



When a VMware session is suspended, the contents of “physical memory” are contained in a file with the .vmem extension. The format of this file is very similar to that of a memory dump using dd.exe, another tool we’ll talk about shortly.

VMware isn’t the only virtualization product available. Others include VirtualPC from Microsoft as well as the freeware Bochs.²¹ These virtualization products are also mentioned in Chapter 6, “Executable File Analysis.” However, none of these virtualization products has been tested to see whether it can generate dumps of physical memory. Let’s look at the pros and cons:

- **Pros** If this is an option available to you, suspending a VMware session is quick, easy, and minimizes the investigator’s interaction with and impact on the system.
- **Cons** VMware or other virtualization technologies do not seem to be widely used in systems that require the attention of a first responder.

Hibernation File

When a Windows (Windows 2000 or later) system “hibernates,” the Power Manager saves the compressed contents of physical memory to a file called Hiberfil.sys in the root directory of the system volume. This file is large enough to hold the uncompressed contents of physical memory, but compression is used to minimize disk I/O and to improve resume-from-hibernation performance. During the boot process, if a valid Hiberfil.sys file is located, the NTLDR will load the file’s contents into physical memory and transfer control to code within the kernel that handles resuming system operation after a hibernation (loading drivers and so on). This functionality is most often found on laptop systems. Here are the pros and cons:

- **Pros** Analyzing the contents of a hibernation file could give you a clue as to what was happening on the system at some point in the past.
- **Cons** The hibernation file is compressed and in most cases will not contain the current contents of memory. (The hibernation file might be significantly out of date.)

DD

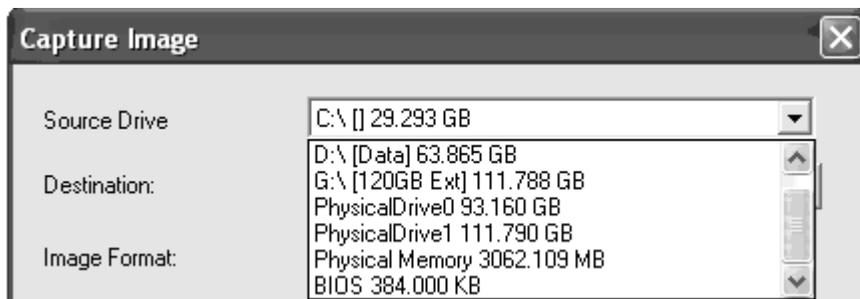
DD is the short name given to the “data dumper” tool from UNIX, which has a variety of uses, not the least of which is to copy files or even entire hard drives. *DD* has long been considered a standard for producing forensic images, and most major forensic imaging/acquisition tools as well as analysis tools support the *dd* format. GMG Systems Inc. produced a modified version of *dd* that runs on Windows systems and can be used to dump the contents of physical memory from Windows 2000 and XP systems. This version of *dd* is part of the Forensic Acquisition Utilities,²² which are available for free download. This utility is able to collect the contents of physical memory by accessing the \Device\PhysicalMemory object from user mode. The following command line can be used to capture the contents of RAM in the file *ram.img* on the local system:

```
D:\tools>dd if=\\.\PhysicalMemory of=ram.img bs=4096 conv=noerror
```

Of course, you can write the output file to a thumb drive or to a shared folder that is already available on the system. This version of *dd.exe* also allows compression and the generation of cryptographic hashes for the content. Because of the volatile nature of RAM, however, it is not advisable to hash it until it is written from the disk. If the user images memory twice, even with little delay, the contents of RAM and thus the subsequent hashes will be different. In this case it is only worthwhile to worry about the integrity of the image after it has been collected.

Other tools use a process similar to `dd.exe` to capture the contents of RAM. ProDiscover IR (version 4.8 was used in writing this book) allows the investigator to collect the contents of physical memory (as well as system BIOS) via a remote server applet that can be distributed to a system via removable storage media (CD, thumb drive, etc.) or via the network. The UI for this capability is illustrated in Figure 3.2.

Figure 3.2 Excerpt of Capture Image Dialog Box from ProDiscover IR



One of the problems with this technique, however, is that as of Windows 2003 SP1, access to the `\Device\PhysicalMemory` object has been restricted from user mode. That is, only kernel drivers are allowed to use this object. To address this issue, GMG Systems produced a new utility called `KntDD`, which is part of the `KnTTools` set of utilities. According to the licensing for `KnTTools`, the utilities are available for private sale to law enforcement personnel and bona fide security professionals. `KntDD` includes the following capabilities:

- Able to acquire the contents of physical memory using multiple methods, including via the `\Device\PhysicalMemory` object.
- Runs on Microsoft operating systems from Windows 2000 through Vista, to include AMD64 versions of the operating systems
- Able to convert raw memory “image” to Microsoft crash dump format so that the resulting data can be analyzed using the Microsoft Debugging Tools
- Able to acquire to a local removable (USB, FireWire) storage device as well as via the network using TCP/IP

- Designed specifically for forensic use, with audit logging and cryptographic integrity checks

The KntTools Enterprise Edition includes the following capabilities:

- Bulk encryption of output using X.509 certifications, AES-256 (default), and downgrading to 3DES on Windows 2000
- Memory acquisition using a KnTDDSvc service
- A remote deployment module that is able to deploy the KnTDDSvc service by either “pushing” it to a remote admin share or “pulling” it from a Web server over SSL, with cryptographic verification of the binaries before they are executed

One of the aspects of using `dd.exe`, and tools like it, that the investigator needs to keep in mind is Locard’s Exchange Principle. To use `dd.exe` to collect the contents of RAM, `dd.exe` must be loaded into RAM as a running process. This means that memory space is used and other processes might have pages that are written out to the page file.

Another aspect of these tools to keep in mind is that they do not freeze the state of the system, as occurs when a crash dump is generated. This means that while the tool is reading though the contents of RAM, as the thirtieth “page” is being read, the eleventh page could change as the process using that page continues to run. The amount of time it ultimately takes to complete the dump depends on factors such as processor speed and rates of bus and disk I/O. The question then becomes, are these changes that occur in the limited amount of time enough to affect the results of your analysis?

Let’s look at the pros and cons of this technique:

- **Pros** Under most incident response conditions, `dd.exe` might be the best method for retrieving the contents of physical memory. This tool does not require that the system be taken down, nor does it restrict how and to where the contents of physical memory are written (i.e., using *netcat*, you can write the contents of RAM out over a socket to another system rather than to the local hard drive). Further, tools have been developed and made freely available to parse the contents of these RAM dumps to extract information about processes, network connections, and the like. Further, development of the KnTTools allows for continued support of this methodology beyond Windows 2003 SP1.

- **Cons** The primary issue with using a methodology such as the Forensic Acquisition Utilities or KnTTools is that the system is still running when the contents of physical memory are retrieved. This means that not only are memory pages consumed simply by using the utilities (i.e., executable images are read and loaded into memory), but as the tool enumerates through memory, pages that have already been read can change. That is to say that the state of the system and its memory are not frozen in time, as would be the case with acquiring a forensic image of a hard drive via the traditional means.

Be that as it may, acquiring the contents of physical memory using the Forensic Acquisition Utilities is perhaps the most frequently used methodology to date.

Analyzing a Physical Memory Dump

Now that we have the contents of RAM from a system, what can we do with them? For the most part, prior to the summer of 2005, the standard operating procedure for most folks who had bothered to collect a RAM dump (usually via the Forensic Acquisition Utilities `dd.exe`) was to run `strings.exe` against it, run `grep` searches (for e-mail addresses, IP addresses, and so on), or both. Although this would result in investigative leads (finding what appeared to be a password “close” to a username would give investigators a clue) that would often lead to something definitive, what it does not provide is overall context to the information that is found. For example, is that string that was located part of a word processing or text document, or was it copied to the system clipboard? What process was using the memory where that string or IP address was located?

With the DFRWS 2005 Memory Challenge as a catalyst, steps have been taken in an attempt to add context to the information found in RAM. By locating specific processes (or other objects in maintained in memory) and the memory pages used by those processes, investigators can gain greater insight into the information they discover as well as perform significant data reduction by filtering out “known good” processes and data and focusing on the data that appears “unusual.” Several individuals have written tools that can be used to parse through RAM dumps and retrieve detailed information about processes and other structures.

Throughout the rest of this chapter, we will be using the memory dumps from the DFRWS 2005 Memory Challenge as exemplars, for examples and demonstrations of tools and techniques for parsing memory dumps. You’re probably asking yourself, why even bother with that? Windows 2000 is new MS-DOS, right? Well, that’s probably not far from the truth, but the dumps do provide an excellent basis

for examples because they have already been examined in great detail. Also, they're freely available for download and examination.

Process Basics

Throughout this chapter, we will focus primarily on parsing information regarding processes from a RAM dump. This is due, in part, to the fact that the majority of the publicly available research and tools focus on processes as a source of forensic information. That is not to say that other objects within memory should be excluded but rather that most researchers seem to be focusing on processes. We will discuss another means of retrieving information from a RAM dump later in the chapter, but for now, we will focus our efforts on processes.

EProcess Structure

Each process on a Windows system is represented as an executive process, or EProcess, block. This EProcess block is a data structure in which various attributes of the process, as well as pointers to a number of other attributes and data structures (threads, the process environment block) relating to the process, are maintained. Because the data structure is a sequence of bytes, each sequence with a specific meaning and purpose, these structures can be read and analyzed by an investigator. However, the one thing to keep in mind is that the only thing consistent between versions of the Windows operating system regarding these structures is that they aren't consistent. You heard right: The size and even the values of the structures change not only between operating system versions (for example, Windows 2000 to XP) but also between service packs of the same version of the operating system (Windows XP to XP SP 2).

Andreas Schuster has done a great job of documenting the EProcess block structures in his blog.²³ However, it is relatively easy to view the contents of the EProcess structure (or any other structure available on Windows). First, download and install the Microsoft Debugging Tools²⁴ and the correct symbols for your operating system and Service Pack. Then download LiveKD.exe from SysInternals.com (when you type **sysinternals.com** into the address bar of your browser, you will be automatically redirected to the Microsoft site, since Mark Russinovich is now employed by Microsoft) and for convenience, copy it into the same directory as the debugging tools. Once you've done this, open a command prompt, change to the directory where you installed the debugging tools, and type the following command:

```
D:\debug>livekd -w
```

This command will open WinDbg, the GUI interface to the debugger tools. To see what the entire contents of an EProcess block “look like” (with all the substructures that make up the EProcess structure broken out), type **dt -a -b -v _EPROCESS** into the command window and press **Enter**. The *-a* flag shows each array element on a new line, with its index, and the *-b* switch displays blocks recursively. The *-v* flag creates more verbose output, telling you the overall size of each structure, for example. In some cases it can be helpful to include the *-r* flag for recursive output. The following illustrates a short excerpt from the results of this command, run on a Windows 2000 system:

```
kd> dt -a -b -v _EPROCESS
struct _EPROCESS, 94 elements, 0x290 bytes
    +0x000 Pcb                : struct _KPROCESS, 26 elements, 0x6c bytes
    +0x000 Header            : struct _DISPATCHER_HEADER, 6 elements, 0x10
bytes
    +0x000 Type              : UChar
    +0x001 Absolute          : UChar
    +0x002 Size              : UChar
    +0x003 Inserted         : UChar
    +0x004 SignalState      : Int4B
    +0x008 WaitListHead     : struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x000 Flink        : Ptr32 to
        +0x004 Blink        : Ptr32 to
    +0x010 ProfileListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x000 Flink        : Ptr32 to
        +0x004 Blink        : Ptr32 to
    +0x018 DirectoryTableBase : (2 elements) Uint4B
```

The entire output is much longer (according to the header, the entire structure is 0x290 bytes long), but don’t worry, we will address important (from a forensics/investigative aspect) elements of the structure as we progress through this chapter.

An important element of a process that is pointed to by the EProcess structure is the *process environment block*, or PEB. This structure contains a great deal of information, but the elements that are important to us, as forensic investigators, are:

- A pointer to the loader data (referred to as *PPEB_LDR_DATA*) structure that includes pointers or references to modules (DLLs) used by the process
- A pointer to the image base address, where we can expect to find the beginning of the executable image file

- A pointer to the process parameters structure, which itself maintains the DLL path, the path to the executable image, and the command line used to launch the process

Parsing this information from a dump file can prove to be extremely useful to an investigator, as we will see throughout the rest of this chapter.

Process Creation Mechanism

Now that we know a little bit about the various structures involved with processes, it would be helpful to know something about how those structures are used by the operating system, particularly when it comes to creating an actual process.

There are a number of steps that are followed when a process is created. These steps can be broken down into six stages:²⁵

1. The image (.exe) file to be executed is opened. During this stage, the appropriate subsystem (Posix, MS-DOS, Win 16, etc.) is identified. Also, the Image File Execution Options Registry key (see Chapter 4, “Registry Analysis”) is checked to see if there is a Debugger value, and if there is, the process starts over.
2. The EProcess object is created. The kernel process block (KProcess), the process environment block, and the initial address space are also set up.
3. The initial thread is created.
4. The Windows subsystem is notified of the creation of the new process and thread, along with the ID of the process’s creator and a flag to identify whether the process belongs to a Windows process.
5. Execution of the initial thread starts. At this point, the process environment has been set up and resources have been allocated for the process’s thread(s) to use.
6. The initialization of the address space is completed, in the context of the new process and thread.

At this point, the process now consumes space in memory in accordance with EProcess structure (which includes the KProcess structure) and the PEB structure. The process has at least one thread and may begin consuming additional memory resources as the process itself executes. At this point, if the process or memory as a whole is halted and dumped, there will at the very least be something to analyze.

Parsing Memory Contents

The tools described in the DFRWS 2005 Memory Challenge used a methodology for parsing memory contents of locating and enumerating the active process list, using specific values/offsets (derived from system files) to identify the beginning of the list and then walking through the doubly linked list until all the active processes had been identified. The location of the offset for the beginning of the active process list was derived from one of the important system files, `ntoskrnl.exe`.

Andreas Schuster took a different approach in his Perl script called `ptfinder.pl`. His idea was to take a “brute-force” approach to the problem—identifying specific characteristics of processes in memory and then enumerating the `EProcess` blocks as well as other information about the processes based on those characteristics. Andreas began his approach by enumerating the structure of the `DISPATCHER_HEADER`, which is located at offset 0 for each `EProcess` block (actually, it’s within the structure known as the `KProcess` block). Using LiveKD, we see that the enumerated structure from a Windows 2000 system has the following elements:

```
+0x000 Header      : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
+0x000 Type        : UChar
+0x001 Absolute    : UChar
+0x002 Size        : UChar
+0x003 Inserted    : UChar
+0x004 SignalState : Int4B
```

In a nutshell, Andreas found that some of the elements for the `DISPATCHER_HEADER` were consistent in all processes on the system. He examined the `DISPATCHER_HEADER` elements for processes (and threads) on systems ranging from Windows 2000 up through early betas of Vista and found that the *Type* value remained consistent across each version of the operating system. He also found that the *Size* value remained consistent within various versions of the operating system (for example, all processes on Windows 2000 or XP had the same *Size* value) but changed between those versions (for instance, for Windows 2000, the *Size* value is 0x1b, but for early versions of Vista, it was 0x20).

Using this information as well as the total size of the structure and the way the structure itself could be broken down, Andreas wrote his `ptfinder.pl` Perl script, which would enumerate processes and threads located in a memory dump. At the DFRWS 2006 conference he also presented a paper, *Searching for Processes and Threads in Microsoft Windows Memory Dumps*,²⁶ which addressed not only the data structures that make up processes and threads but also various rules to determine whether what was found was a legitimate structure or just a bunch of bytes in a file.

NOTE

In fall 2006, Richard McQuown of ForensicZone.com put together a GUI front end for Andreas Schuster's PTFinder tools. The PTFinder tools are Perl scripts and require that the Perl interpreter be installed on a system to run them. (Perl is installed by default on many Linux distributions and is freely available for Windows platforms from ActiveState.com.)

Not only can Richard's tool detect the operating system of the RAM dump (rather than have the user enter it manually) using code I'll discuss later in this chapter; it can also provide a graphical representation of the output. PTFinderFE has some interesting applications, particularly with regard to visualization.

In spring 2006, I wrote some of my own tools to assist in parsing through Windows RAM dump files. Since the currently available exemplars at the time were the dumps for Windows 2000 systems available from the DFRWS 2005 Memory Challenge, I focused my initial efforts on producing code that worked for that platform. This allowed me to address various issues in code development without getting too wrapped up in the myriad differences between the various versions of the Windows operating system. The result was four separate Perl scripts, each run from the command line. These scripts are all provided on the accompanying DVD, and we'll go through each one separately.

Lsproc.pl



Lsproc, short for *list processes*, is similar to Andreas's PTFinder.pl; however, *lsproc.pl* locates processes but not threads. *Lsproc.pl* takes a single argument, the path and name to a RAM dump file:

```
c:\perl\memory>lsproc.pl d:\dumps\drfws1-mem.dmp
```

The output of *lsproc.pl* appears at the console (i.e., *STDOUT*) in six columns; the word *Proc* (I was anticipating adding threads at a later date), the parent process ID (*PPID*), the process ID (*PID*), the name of the process, the offset of the process structure within the dump file, and the creation time of the process. An excerpt of the *lsproc.pl* output appears as follows:

```
Proc 820 324 helix.exe 0x00306020 Sun Jun 5 14:09:27
2005
Proc 0 0 Idle 0x0046d160
```

Proc 2005	600	668	UMGR32.EXE	0x0095f020	Sun Jun 5	00:55:08
Proc 2005	324	1112	cmd2k.exe	0x00dcc020	Sun Jun 5	14:14:25
Proc 2005	668	784	dfrws2005.exe(x)	0x00e1fb60	Sun Jun 5	01:00:53
Proc 2005	156	176	winlogon.exe	0x01045d60	Sun Jun 5	00:32:44
Proc 2005	156	176	winlogon.exe	0x01048140	Sat Jun 4	23:36:31
Proc 2005	144	164	winlogon.exe	0x0104ca00	Fri Jun 3	01:25:54
Proc 2005	156	180	csrss.exe	0x01286480	Sun Jun 5	00:32:43
Proc 2005	144	168	csrss.exe	0x01297b40	Fri Jun 3	01:25:53
Proc 2005	8	156	smss.exe	0x012b62c0	Sun Jun 5	00:32:40
Proc	0	8	System	0x0141dc60		
Proc 2005	668	784	dfrws2005.exe(x)	0x016a9b60	Sun Jun 5	01:00:53
Proc 2005	1112	1152	dd.exe(x)	0x019d1980	Sun Jun 5	14:14:38
Proc 2005	228	592	dfrws2005.exe	0x02138640	Sun Jun 5	01:00:53
Proc 2005	820	1076	cmd.exe	0x02138c40	Sun Jun 5	00:35:18
Proc 2005	240	788	metasploit.exe(x)	0x02686cc0	Sun Jun 5	00:38:37
Proc 2005	820	964	Apoint.exe	0x02b84400	Sun Jun 5	00:33:57
Proc 2005	820	972	HKserv.exe	0x02bf86e0	Sun Jun 5	00:33:57
Proc 2005	820	988	DragDrop.exe	0x02c46020	Sun Jun 5	00:33:57
Proc 2005	820	1008	alogserv.exe	0x02e7ea20	Sun Jun 5	00:33:57
Proc 2005	820	972	HKserv.exe	0x02f806e0	Sun Jun 5	00:33:57
Proc 2005	820	1012	tgcmd.exe	0x030826a0	Sun Jun 5	00:33:58
Proc 2005	176	800	userinit.exe(x)	0x03e35020	Sun Jun 5	00:33:52

```

Proc 800      820      Explorer.Exe      0x03e35ae0  Sun Jun  5 00:33:53
2005
Proc 820     1048     PcfMgr.exe       0x040b4660  Sun Jun  5 00:34:01
2005

```

The first process listed in the `lsproc.pl` output is “`helix.exe`. According to the information provided at the DFRWS 2005 Memory Challenge Web site, utilities on the Helix Live CD²⁷ were used to acquire the memory dump.

The aforementioned listing shows only an excerpt of the `lsproc.pl` output. There were a total of 45 processes located in the memory dump file. You’ll notice in the output that several of the processes have (x) after the process name. This indicates that the processes have exited. In these cases, the contents of physical memory (for example, pages) have been freed for use but have not been overwritten yet, so it would seem that some information is retained in physical memory following a reboot.

NOTE

Looking closely, you’ll notice some interesting things about the `lsproc.pl` output. One is that the `csrss.exe` process (PID = 168) has a creation date that appears to be a day or two earlier than the other listed processes. Looking even more closely, you’ll see something similar for two `winlogon.exe` processes (PID = 164 and 176). Andreas Schuster noticed these as well, and according to an entry on data persistence in his blog,²⁸ the system boot time for the dump file was determined to be Sunday, Jan 5, 2005, at approximately 00:32:27. So, where do these processes come from?

As Andreas points out in his blog, without having more definitive information about the state of the test system prior to collecting data for the memory challenge, it is difficult to develop a complete understanding of this issue. However, the specifications of the test system were known and documented, and it was noted that the system suffered a crash dump during data collection.

It is entirely possible that the data survived the reboot. There don’t seem to be any specifications that require that when a Windows system shuts down or suffers a crash dump, the contents of physical memory are zeroed out or wiped in some manner. It is possible, then, that contents of physical memory remain in their previous state, and if they are not overwritten when the system is restarted, the data is still available for analysis. Many BIOS versions have a feature to overwrite memory during boot as part of a RAM test, but this feature is usually disabled to speed up the boot process.

This is definitely an area that requires further study. As Andreas states,²⁹ this area of study has “a bright future.”



Lspd.pl

Lspd.pl is a Perl script that will allow you to list the details of the process. Like the other tools that we will be discussing, lspd.pl is a command-line Perl script that relies on the output of lsproc.pl to obtain its information. Specifically, lspd.pl takes two arguments: the path and name of the dump file and the offset from the lsproc.pl output of the process that you’re interested in. Although lsproc.pl took some time to parse through the contents of the dump file, lspd.pl is much quicker, since you’re telling it exactly where to go in the file to enumerate its information.

Let’s take a look at a specific process. In this case, we’ll look at dd.exe, the process with PID 284. The command line to use lspd.pl to get detailed information about this process is:

```
c:\perl\memory>lspd.pl d:\dumps\dfrrs1-mem.dmp 0x0414dd60
```

Notice that with lspd.pl, we’re using two arguments: the name and path to the dump file and the physical offset in the dump file where we found the process with lsproc.pl. We’ll take a look at the output of lspd.pl in sections, starting with some useful information pulled directly from the EProcess structure itself:

```
Process Name : dd.exe
PID          : 284
Parent PID   : 1112
TFLINK      : 0xff2401c4
TBLINK      : 0xff2401c4
FLINK       : 0x8046b980
BLINK       : 0xff1190c0
SubSystem   : 4.0
Exit Status  : 259
Create Time  : Sun Jun  5 14:53:42 2005
Exit Called  : 0
DTB         : 0x01d9e000
ObjTable    : 0xff158708 (0x00eb6708)
PEB        : 0x7ffdf000 (0x02c2d000)
             InheritedAddressSpace      : 0
             ReadImageFileExecutionOptions : 0
```

```

BeingDebugged          : 0
CSDVersion             : Service Pack 1
Mutant                 = 0xffffffff
Img Base Addr         = 0x00400000 (0x00fee000)
PEB_LDR_DATA          = 0x00131e90 (0x03a1ee90)
Params                = 0x00020000 (0x03a11000)

```

Lspd.pl also follows pointers provided by the EProcess structure to collect other data as well. For example, we can also see the path to the executable image and the command line used to launch the process (bold added for emphasis):

```

Current Directory Path = E:\Shells\
DllPath                =
E:\Acquisition\FAU\.;C:\WINNT\System32;C:\WINNT\system;
C:\WINNT;E:\Acquisition\FAU\;E:\Acquisition\GNU\;E:\Acquisition\CYGWIN\;E:\I
R\bin\;E:\IR\WFT;E:\IR\windbg\;E:\IR\Foundstone\;E:\IR\Cygwin;E:\IR\somarsot
t\;E:\IR\sysinternals\;E:\IR\ntsecurity\;E:\IR\perl\;E:\Static-
Binaries\gnu_utils_win32\;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem
ImagePathName         = E:\Acquisition\FAU\dd.exe
Command Line          = ..\Acquisition\FAU\dd.exe if=\\.\PhysicalMemory
of=F:\intrusion2005\physicalmemory.dd conv=noerror --md5sum --verifymd5 --
md5out=F:\intrusion2005\physicalmemory.dd.md5 --
log=F:\intrusion2005\audit.log
Environment Offset     = 0x00000000 (0x00000000)
Window Title         = ..\Acquisition\FAU\dd.exe if=\\.\PhysicalMemory
of=F:\intrusion2005\physicalmemory.dd conv=noerror --md5sum --verifymd5 --
md5out=F:\intrusion2005\physicalmemory.dd.md5 --
log=F:\intrusion2005\audit.log
Desktop Name         = WinSta0\Default

```

Lspd.pl also retrieves a list of the names of various modules (DLLs) used by the process and whatever available handles (file handles and so on) it can find in memory. For example, lspd.pl found that the dd.exe had the following file handle open:

```

Type : File
      Name = \intrusion2005\audit.log

```

As we can see from the preceding command line, the file \intrusion\audit.log is located on the F:\ drive and is the output file for the log of activity generated by dd.exe, which explains why it would be listed as an open file handle in use by the process. Using this information as derived from other processes, you can get an understanding of files you should be concerned with during an investigation. In this particular instance, we can assume that the E:\ drive listed in *ImagePathName* is a

CD-ROM drive, since Helix can be run from a CD. We can confirm this by checking Registry values in an image of the system in question (a system image is not provided as part of the memory challenge, however). We can also use similar information to find out a little bit more about the F:\ drive. This information will be covered in Chapter 4, “Registry Analysis.”

Finally, one other thing that `lspd.pl` will do is go to the location pointed to by the Image Base *Addr* value (once it has been translated from a virtual address to a physical offset within the memory dump file) and check to see if a valid executable image is located at that address. This check is very simple; all it does is read the first 2 bytes starting at the translated address to see if they're *MZ*. These 2 bytes are not a definitive check, but portable executable (PE) files (files with `.exe`, `.dll`, `.ocs`, `.sys`, and etc. extensions) start with the initials of Mark Zbikowski, one of the early architects of MS-DOS and Windows NT. The format of the PE file and its header is addressed in greater detail in Chapter 6, “Executable File Analysis.”

Parsing Process Memory

We discussed the need for context for evidence earlier in this chapter, and this can be achieved, in part, by extracting the memory used by a process. In the past, investigators have used tools such as `strings.exe` or `grep` searches to parse through the contents of a RAM dump and look for interesting strings (passwords), IP or e-mail addresses, URLs, and the like. However, when we're parsing through a file that is about half a megabyte in size, there isn't a great deal of context to the information we find. Sometimes an investigator will open the dump file in a hex editor and locate the interesting string, and if she saw what appeared to be a username nearby, she might assume that the string is a password. However, investigating a RAM dump file in this manner does not allow the investigator to correlate that string to a particular process. Remember the example of Locard's Exchange Principle from Chapter 1? Had we collected the contents of physical memory during the example, we would have had no way to definitively say that a particular IP address or other data, such as a directory listing, was tied to a specific event or process. However, if we use the information provided in the process structure within memory and locate all the pages the process used that were still in memory when the contents were dumped, we could then run our searches and determine which process was using that information.



The tool `lspm.pl` allows us to do this automatically. `Lspm.pl` takes the same arguments as `lspd.pl` (the name and path of the dump file, and the physical offset within the file of the process structure) and extracts the available pages from the dump file,

writing them to a file within the current working directory. To run `lspm.pl` against the `dd.exe` process, we use the following command line:

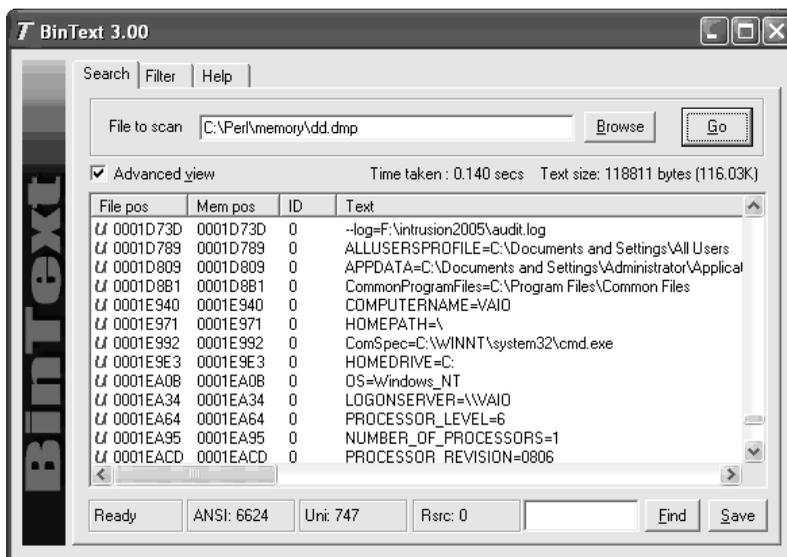
```
c:\perl\memory>lspm.pl d:\dumps\dfrrs1-mem.dmp 0x0414dd60
```

The output looks like this:

```
Name : dd.exe -> 0x01d9e000
There are 372 pages (1523712 bytes) to process.
Dumping process memory to dd.dmp...
Done.
```

Now we have a file called `dd.dmp` that is 1,523,712 bytes in size and contains all the memory pages (372 in total) for that process that were still available when the dump file was created. You can run `strings.exe` or use `BinText` (see Figure 3.3) from Foundstone.com to parse through the file looking for Unicode and ASCII strings, or run `grep` searches for IP or e-mail addresses and credit card or Social Security numbers.

Figure 3.3 Contents of Process Memory in BinText



In Figure 3.3, we can see some of the Unicode strings contained in the memory used by the `dd.exe` process, including the name of the system and the name of the *LogonServer* for the session. All of this information can help further the investigator's understanding of the case; an important aspect of this capability is that now we can correlate what we find to a specific process.

Extracting the Process Image

As we saw earlier in this chapter, when a process is launched, the executable file is read into memory. One of the pieces of information that we can get from the process details (via `lspd.pl`) is the offset within the dump file to the Image Base Address. As we saw, `lspd.pl` will do a quick check to see whether an executable image can be found at that location. One of the things we can do to develop this information further is to parse the PE file header (the contents of which will be covered in detail in Chapter 6, “Executable File Analysis”) and see whether we can extract the entire contents of the executable image from the dump file. `Lspi.pl` lets us do this automatically.



`Lspi.pl` is a Perl script that takes the same arguments as `lspd.pl` and `lspm.pl` and locates the beginning of the executable image for that process. If the Image Base Address offset does indeed lead to an executable image file, `lspi.pl` will parse the values contained in the PE header to locate the pages that make up the rest of the executable image file.

Okay, so we can run `lspi.pl` against the `dd.exe` process (with the PID of 284) using the following command line:

```
c:\perl\memory>lspi.pl d:\dumps\dfrrs1-mem.dmp 0x0414dd60
```

The output of the command appears as follows:

```
Process Name : dd.exe
PID          : 284
DTB         : 0x01d9e000
PEB         : 0x7ffdf000 (0x02c2d000)
ImgBaseAddr : 0x00400000 (0x00fee000)
e_lfanew = 0xe8
NT Header = 0x4550
Reading the Image File Header
Sections = 4
Opt Header Size = 0x000000e0 (224 bytes)
Characteristics:
    IMAGE_FILE_EXECUTABLE_IMAGE
    IMAGE_FILE_LOCAL_SYMS_STRIPPED
    IMAGE_FILE_RELOCS_STRIPPED
    IMAGE_FILE_LINE_NUMS_STRIPPED
    IMAGE_FILE_32BIT_MACHINE
Machine = IMAGE_FILE_MACHINE_I860
Reading the Image Optional Header
```

```

Opt Header Magic = 0x10b
Subsystem       : IMAGE_SUBSYSTEM_WINDOWS_CUI
Entry Pt Addr  : 0x00006bda
Image Base     : 0x00400000
File Align     : 0x00001000

```

Reading the Image Data Directory information

Data Directory	RVA	Size
-----	---	----
ResourceTable	0x0000d000	0x00000430
DebugTable	0x00000000	0x00000000
BaseRelocTable	0x00000000	0x00000000
DelayImportDesc	0x0000af7c	0x000000a0
TLSTable	0x00000000	0x00000000
GlobalPtrReg	0x00000000	0x00000000
ArchSpecific	0x00000000	0x00000000
CLIHeader	0x00000000	0x00000000
LoadConfigTable	0x00000000	0x00000000
ExceptionTable	0x00000000	0x00000000
ImportTable	0x0000b25c	0x000000a0
unused	0x00000000	0x00000000
BoundImportTable	0x00000000	0x00000000
ExportTable	0x00000000	0x00000000
CertificateTable	0x00000000	0x00000000
IAT	0x00007000	0x00000210

Reading Image Section Header Information

Name	Virt Sz	Virt Addr	rData Ofs	rData Sz	Char
----	-----	-----	-----	-----	----
.text	0x00005ee0	0x00001000	0x00001000	0x00006000	0x60000020
.data	0x000002fc	0x0000c000	0x0000c000	0x00001000	0xc0000040
.rsrc	0x00000430	0x0000d000	0x0000d000	0x00001000	0x40000040
.rdata	0x00004cfa	0x00007000	0x00007000	0x00005000	0x40000040

Reassembling image file into dd.exe.img

Bytes written = 57344

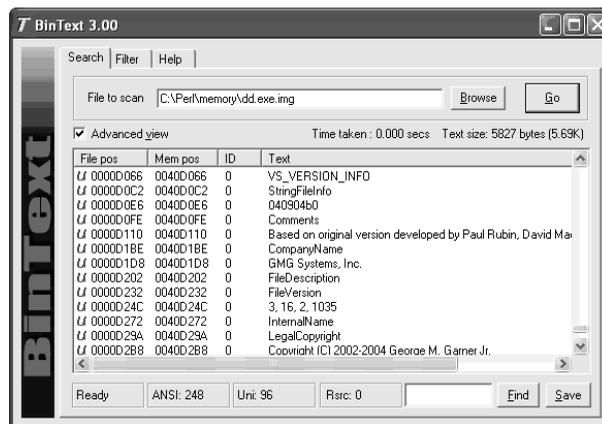
New file size = 57344

As you can see, the output of `lsapi.pl` is pretty verbose, and much of the information displayed might not be readily useful to (or understood by) an investigator unless that investigator is interested in malware analysis. Again, this information will be discussed in detail in Chapter 6, “Executable File Analysis.” For now, the important elements are

the table that follows the words “Reading Image Section Header Information” and the name of the file to which the executable image was reassembled. The section header information provides us with a road map for reassembling the executable image because it lets us know where to find the pages that make up that image file. `Lspi.pl` uses this road map and attempts to reassemble the executable image into a file. If it’s successful, it writes the file out to the file based on the name of the process, with `.img` appended (to prevent accidental execution of the file). `Lspi.pl` will not reassemble the file if any of the memory pages have been marked as invalid and are no longer located in memory (for example, they have been paged out to the swap file, `pagefile.sys`). Instead, `lspi.pl` will report that it could not reassemble the complete file because some pages (even just one) were not available in memory.

Now, the file that we extract from the memory dump will not be exactly the same as the original executable file. This is due to the fact that some of the file’s sections are writable, and those sections will change as the process is executing. As the process executes, various elements of the executable code (addresses, variables, and so on) will change according to the environment and the stage of execution. However, there are a couple of ways that we can determine the nature of a file and get some information about its purpose. One of those ways is to see whether the file has any file version information compiled into it, as is done with most files created by legitimate software companies. As we saw from the section headers of the image file, there is a section named `.rsrc`, which is the name often used for a resource section of a PE file. This section can contain a variety of resources, such as dialogs and version strings, and is organized like a file system of sorts. Using `BinText`, we can look for the Unicode string `VS_VERSION_INFO`³⁰, and see whether there is any identifying information available in the executable image file. Figure 3.4 illustrates some of the strings found in the `dd.exe.img` file using `BinText`.

Figure 3.4 Version Strings Found in `dd.exe.img` with `BinText`



Another method of determining the nature of the file is to use file hashing. You're probably thinking, "Hey, wait a minute! You just said that the file created by `lspi.pl` isn't exactly the same as the original file, so how can we use hashing?" Well, you're right ... up to a point. We can't use MD5 hashes for comparison, because as we know, altering even a single bit—flipping a 1 to a 0—will cause an entirely different hash to be computed. So what can we do?

In summer 2006, Jesse Kornblum released a tool called `sdeep`³¹, that implements something called *context-triggered piecewise hashing*, or *fuzzy hashing*. For a detailed understanding of what this entails, be sure to read Jesse's DFRWS 2006 paper³² on the subject. In a nutshell, Jesse implemented an algorithm that will tell you a weighted percentage of the identical sequences of bits the files have in common, based on their hashes, computed by `ssdeep`. Since we know that in this case, George Garner's version of `dd.exe` was used to dump the contents of RAM from a Windows 2000 system for the DFRWS 2005 Memory Challenge, we can compare the `dd.exe.img` file to the original `dd.exe` file that we just happen to have available.

First, we start by using `ssdeep.exe` to compute a hash for our image file:

```
D:\tools>ssdeep c:\perl\memory\dd.exe.img > dd.sdp
```

We've now generated the hash and saved the information to the `dd.sdp` file. Using other switches available for `ssdeep.exe`, we can quickly compare the `.img` file to the original executable image:

```
D:\tools>ssdeep -v -m dd.sdp d:\tools\dd\old\dd.exe
d:\tools\dd\old\dd.exe matches c:\perl\memory\dd.exe.img (97)
```

This can also be done in one command line using either the `-d` or `-p` switches:

```
D:\tools\> ssdeep -d c:\perl\memory\dd.exe.img d:\tools\dd\old\dd.exe
C:\perl\memory\dd.exe.img matches d:\tools\dd\old\dd.exe (97)
```

We see that the image file generated by `lspi.pl` has a 97 percent likelihood of matching the original `dd.exe` file.

Remember, for a hash comparison to work properly, we need something to which we can compare the files created by `lspi.pl`. `Ssdeep.exe` is a relatively new, albeit extremely powerful, tool, and it will likely be a while before hash sets either are generated using `ssdeep.exe` or incorporate hashes calculated using `ssdeep.exe`.

Memory Dump Analysis and the Page File

So far, we've looked at parsing and analyzing the contents of a RAM dump in isolation—that is, without the benefit of any additional information. This means that tools such as `lspm.pl` that rely solely on the contents of the RAM dump will provide

an incomplete memory dump, since memory pages that have been swapped out to the page file (pagefile.sys on Windows systems) will not be incorporated in the resulting memory dump. To overcome this deficiency, in the spring of 2006 Nicholas Paul Maclean published his thesis work, *Acquisition and Analysis of Windows Memory* (at the time of this writing, I could not locate an active link to the thesis), which explains the inner workings of the Windows memory management system and provides an open-source tool called *vtop* (written in Python) to reconstruct the virtual address space of a process.

Jesse Kornblum released his “Buffalo” paper (the full title is *Using Every Part of the Buffalo in Windows Memory Analysis*) early in 2007 via his Web site; the paper was also published in the *Digital Investigation Journal*. In this paper, Jesse demonstrates the nuances of page address translation and how the page file can be incorporated into the memory analysis process to establish a more complete (and accurate) view of the information that is available.

Determining the Operating System of a Dump File

Have you ever been handed an image of a system, and when you asked what the operating system is/was, you simply get “Windows” in response? Shakespeare doesn’t cut it here, my friends, because a rose by any other name might *not* smell as sweet. When you’re working with an image of a system, the version of Windows that you’re confronted with *matters*, and depending on the issue you’re dealing with, it could matter a lot. The same is true when you’re dealing with a RAM dump file; in fact, it could be even more so. As I’ve already stated, the structures that are used to define threads and processes in memory vary not only between major versions of the operating system but also within the same version with different service packs installed.

So, when someone hands you a RAM dump and says, “Windows,” you’d probably want to know how to figure that out, wouldn’t you? After all, you don’t want to waste a lot of time running the dump file through every known tool until one of them starts producing valid hits on processes, right? Through personal correspondence (that’s a fancy term for “e-mail”) a while ago, Andreas Schuster suggested to me that the Windows kernel might possibly be loaded into the same location in memory every time Windows boots. Now, that location is likely to, and does, change for every version of Windows, but so far, it seems to be consistent for each version. The easiest way to find this location is to run LiveKD as we did earlier in this chapter, but note the information that’s displayed as it starts up, particularly (on a Windows XP SP2 system):

```
Windows XP Kernel Version 2600 (Service Pack 2) MP (2 procs) Free x86
compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp.050329-1536
Kernel base = 0x804d7000 PsLoadedModuleList = 0x8055c700
```

We're most interested in the information that I've boldfaced—the address of the kernel base. We subtract 0x80000000 from that address and then go to the resulting physical location within the dump file. If the first 2 bytes located at that address are *MZ*, we could have a full-blown Windows PE file at that location, and we *might* have the kernel. From this point, we can use code similar to what's in `lspi.pl` to parse apart the PE header and locate the various sections within the PE file. Since the Windows kernel is a legitimate Microsoft application file, we can be sure that there is a resource section within the file that contains a *VS_VERSION_INFO* section. Following information provided by Microsoft regarding the various structures that make up this section, we can then parse through it looking for the file description string.



On the accompanying DVD, you'll find a file called `osid.pl` that does just that. `Osid.pl` began life as `kern.pl` and found its way into Rick McQuown's `PTFinderFE` utility. Rick asked me via e-mail one day if there was a way to shorten and clarify the output, so I made some modifications to the file (changed the output, added some switches, and so on) and renamed it.

In its simplest form, you can simply run `osid.pl` from the command line, passing in the path to the image file as the sole argument:

```
C:\Perl\memory>osid.pl d:\hacking\xp-laptop1.img
```

Alternatively, you can designate a specific file using the `-f` switch:

```
C:\Perl\memory>osid.pl -f d:\hacking\xp-laptop1.img
```

Both of these commands will give you the same output; in this case, the RAM dump was collected from a Windows XP SP2 system, so the script returns *XPSP2*. If this isn't quite enough information and you'd like to see more, you can add the `-v` switch (for *verbose*), and the script will return the following for the `xp-laptop1.img` file:

```
OS      : XPSP2
Product : Microsoft« Windows« Operating System ver 5.1.2600.2622
```

As you can see, the strings within the *VS_VERSION_INFO* structure that refer to the product name and the product version get concatenated to produce the additional output. If we run the script with both the `-v` and the `-f` switches against the first RAM dump file from the DFRWS 2005 Memory Challenge, we get:

```
OS      : 2000
Product : Microsoft(R) Windows (R) 2000 Operating System ver 5.00.2195.1620
```

This script also works equally well against VMware .vmem files. I ran the script against a .vmem file from a Windows 2000 VMware session and received the following output:

```
OS      : 2000
Product : Microsoft(R) Windows (R) 2000 Operating System ver 5.00.2195.7071
```

Pool Allocations

When the Windows memory manager allocates memory, it generally does so in 4K (4096 bytes) pages. However, allocating an entire 4K page for, say, a sentence copied to the clipboard would be a waste of memory. So the memory manager allocates several pages ahead of time, keeping an available *pool* of memory. Andreas Schuster has done extensive research in this area, and even though Microsoft provides a list of pool headers used to designate commonly used pools, documentation for any meaningful analysis of these pools is simply not available. Many of the commonly used pool headers are listed in the `pooltag.txt`³³ file provided with the Microsoft Debugging Tools, and Microsoft provides a KnowledgeBase article that describes how to locate pool tags/headers used by third-party applications.³⁴ Andreas used a similar method to determine the format of memory pools used to preserve information about network connections in Windows 2000 memory dumps;³⁵ he searched for the pool header in the `tcpip.sys` driver on a Windows 2000 system and was able to determine the format of network connection information within the memory pool.

The downside to searching for memory pool allocations is that although the pool headers do not seem to change between versions of Windows, the format of the data resident within the memory pool changes, and there is no available documentation regarding the format of these memory pools.

Collecting Process Memory

During an investigation, you might be interested in only particular processes rather than a list of all processes, and you'd like more than just the contents of process memory available in a RAM dump file. For example, you might have quickly identified processes of interest that required no additional extensive investigation. There are ways to collect all the memory used by a process—not just what is in physical memory but what is in virtual memory or the page file as well.

To do this, the investigator has available a couple of tools. One is `pmdump.exe`,³⁶ written by Arne Vidstrom and available from NTSecurity.nu. However, as the NTSecurity.nu Web site states, `pmdump.exe` allows you to dump the contents of process memory *without* stopping the process. As we discussed earlier, this allows the process to continue and the contents of memory to change while being written to a file, thereby creating a “smear” of process memory. Also, `pmdump.exe` does not create an output file that can be analyzed with the debugging tools.

Tobias Klein has come up with another method for dumping the contents of process memory in the form of a free (albeit not open-source) tool called Process Dumper.³⁷ Process Dumper (`pd.exe`) dumps the entire process space along with additional metadata and the process environment to the console (STDOUT) so that the output can be redirected to a file or a socket (via *netcat* or other tools; see Chapter 1 for a discussion of some of those tools). A review of the documentation that Tobias makes available for `pd.exe` provides no indication that the process is debugged, halted, or frozen prior to the dumping process. Tobias also provides the Memory Parser GUI utility for parsing the metadata and memory contents collected by the Process Dumper. These tools appear to be an extension of Tobias’s work toward extracting RSA private keys and certificates from process memory.³⁸

Another tool that is available and recommended by a number of sources is `userdump.exe`, available from Microsoft. `Userdump.exe` will allow you to dump any process on the fly, without attaching a debugger and without terminating the process once the dump has been completed. Also, the dump file generated by `userdump.exe` can be read by the MS debugging tools. However, `userdump.exe` requires that a driver be installed for it to work, and depending on the situation, this might not be something you’d want to do.

Based on conversations with Robert Hensing, formerly of the Microsoft PSS Security team, the preferred method of dumping a process is to use the `adplus.vbs` script that ships with the debugging tools. *Adplus* stands for *Autodumplus* and was originally written by Robert (the documentation for the script states that versions 1 through 5 were written by Robert, and as of version 6, Israel Burman has taken over). The help file (`debugger.chm`) for the debugging tools contains a great deal of information about the script as well as the debugging tool (`cdb.exe`) that it uses to dump the processes that you designate. The debugging tools do not require that an additional driver be installed and can be run from a CD. This means that the tools (`adplus.vbs` and `cdb.exe` as well as supporting DLLs) can be written to a CD (`adplus.vbs` uses the Windows scripting host version 5.6, also known as `cscript.exe`, which comes installed on most systems) and used to dump processes to a shared drive or to a USB-connected storage device. Once the dumps have been completed, you can then use the freely available debugging tools to analyze the dump files. In

addition, you can use other tools, such as BinText to extract ASCII, Unicode, and resource strings from the dump file. You can use still other tools to collect additional information about the process. Handle.exe³⁹ (which requires that you have Administrator rights on the system when running it) will provide you with a list of handles (to files, directories, and so on) that have been opened by the process, and listdlls.exe⁴⁰ will show you the full path to and the version numbers of the various modules loaded by a process.

There is extensive help available for using adplus.vbs, not only in the debugging tools help file but also in KB article 286350.⁴¹ Adplus.vbs can be used to hang the process while it is being dumped (in other words, halt it, dump it, and then resume the process) or to crash the process (halt the process, dump it, and then terminate). To run adplus.vbs in hang mode against a process, you would use the following command line:

```
D:\debug>cscript adplus.vbs -quiet -hang -p <PID>
```

This command will create a series of files within the debug directory within a subdirectory prefaced with the name *Hang_mode_* that includes the date and time of the dump. (*Note:* You can change the location where the output is written using the *-o* switch.) What you will see is an adplus.vbs report file, the dump file for the process (multiple processes can be designated using multiple *-p* entries), a process list (generated by default using tlist.exe; you can turn this off using the *-noTList* switch), and a text file showing all the loaded modules (DLLs) used by the process.

Although all the information collected about processes using adplus.vbs can be extremely useful during an investigation, this tool can be used only on processes that are visible via the API. If a process is not visible (say, if it's hidden by a rootkit), you cannot use these tools to collect information about the process.

Summary

By now it should be clear that you have several options for collecting physical or process memory from a system during incident response. In Chapter 1, we examined a number of tools for collecting various portions of volatile memory during live response (processes, network connections, and the like), keeping in mind that there's always the potential for the Windows API (on which the tools rely) being compromised by an attacker. This is true in any case where live response is being performed, and therefore, we might decide to use multiple disparate means of collecting volatile information. A rootkit can hide the existence of a process from most tools that enumerate the list of active processes (`tlst.exe`, `pslist.exe`), but dumping the contents of RAM will allow the investigator to list active and exited processes as well as processes hidden using kernel-mode rootkits. (More about rootkits in Chapter 7, "Rootkits and Rootkit Detection.")

Notes

1. For more information go to www.dfrws.org.
2. For more information go to www.e-fense.com/helix/.
3. For more information go to http://computer.forensikblog.de/en/topics/windows/memory_analysis/.
4. For more information go to www.lurhq.com/truman/.
5. For more information go to www.blackhat.com/html/bh-federal-06/bh-fed-06-speakers.html#Burdach.
6. For more information go to <http://nucia.unomaha.edu/tvidas/>.
7. For more information go to www.grandideastudio.com/.
8. For more information go to www.storm.net.nz/projects/16.
9. For more information go to www.python.org.
10. For more information go to <http://ntsecurity.nu/onmymind/2006/2006-09-02.html>.
11. For more information go to http://en.wikipedia.org/wiki/Blue_Screen_of_Death.
12. For more information go to www.microsoft.com/whdc/devtools/debugging/default.msp.
13. For more information go to www.microsoft.com/downloads/details.aspx?FamilyID=E84D3B35-63C3-445B-810D-9FED3FDEB13F&displaylang=en.
14. For more information go to <http://support.microsoft.com/kb/254649/>.
15. For more information go to <http://support.microsoft.com/kb/274598/>.

16. For more information go to <http://support.microsoft.com/kb/307973/>.
17. For more information go to <http://support.microsoft.com/kb/244139/>.
18. For more information go to www.microsoft.com/downloads/details.aspx?FamilyID=E84D3B35-63C3-445B-810D-9FED3FDEB13F&displaylang=en.
19. For more information go to <http://support.citrix.com/article/CTX107717&parentCategoryID=617>.
20. For more information go to www.microsoft.com/technet/sysinternals/SystemInformation/LiveKd.mspx.
21. For more information go to <http://bochs.sourceforge.net/>.
22. For more information go to <http://users.erols.com/gmgarner/forensics/develop>.
23. For more information go to http://computer.forensikblog.de/en/topics/windows/memory_analysis/.
24. For more information go to www.microsoft.com/whdc/devtools/debugging/default.mspx.
25. Mark E. Russinovich and David A. Solomon, *Windows Internals*, fourth edition, Chapter 6 (Redmond, WA: Microsoft Press, 2005).
26. For more information go to www.dfrws.org/2006/proceedings/2-Schuster.pdf.
27. For more information go to www.e-fense.com/helix/.
28. For more information go to http://computer.forensikblog.de/en/2006/04/persistence_through_the_boot_process.html.
29. For more information go to http://computer.forensikblog.de/en/2006/04/data_lifetime.html.
30. For more information go to <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mobilesdk5/html/wce50lrfVSVERSIONINFO.asp>.
31. For more information go to <http://ssdeep.sourceforge.net/>.
32. For more information go to <http://dfrws.org/2006/proceedings/12-Kornblum.pdf>.
33. For more information go to www.microsoft.com/whdc/driver/tips/PoolMem.mspx.
34. For more information go to <http://support.microsoft.com/default.aspx?scid=kb;en-us;298102>.
35. For more information go to http://computer.forensikblog.de/en/2006/07/finding_network_socket_activity_in_tools.html.
36. For more information go to www.ntsecurity.nu/toolbox/pmdump/.
37. For more information go to www.trapkit.de/research/forensic/pd/index.html.
38. For more information go to www.trapkit.de/research/sslkeyfinder/index.html.

39. For more information go to

www.microsoft.com/technet/sysinternals/ProcessesAndThreads/Handle.mspx.

40. For more information go to

www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ListDlls.mspx.

41. For more information go to <http://support.microsoft.com/kb/286350>

Solutions Fast Track

Dumping Physical Memory

- ☑ Several methodologies are available for dumping the contents of physical memory. The investigator should be aware of the available options as well as their pros and cons so that he or she can make an intelligent choice as to which methodology should be used.
- ☑ Dumping the contents of physical memory from a live system can present issues with consistency due to the fact that the system is still live and processing information while the memory dump is being generated.
- ☑ When dumping the contents of physical memory, the investigator must keep Locard's Exchange Principle in mind.

Analyzing a Physical Memory Dump

- ☑ Depending on the means used to collect the contents of physical memory, various tools are available to extract useful information from the memory dump. The use of `strings.exe`, `BinText`, and `grep` with various regular expressions has been popular, and research conducted beginning in spring 2005 revealed how to extract specific processes.
- ☑ Dumps of physical memory contain useful information and objects such as processes, the contents of the clipboard, and network connections.
- ☑ Continuing research in this area has demonstrated how the page file can be used in conjunction with a RAM dump to develop a more complete set of information.

Collecting Process Memory

- ☑ An investigator could be presented with a situation in which it is not necessary to collect the entire contents of physical memory; rather, the contents of memory used by a single process would be sufficient.
- ☑ Collecting the memory contents of a single process is an option that is available for only those processes that are seen in the active process list by both the operating system and the investigator's utilities. Processes hidden via some means (see Chapter 7, "Rootkits and Rootkit Detection") might not be visible, and the investigator will not be able to provide the process identifier to the tools he or she is using to collect the memory used by the process.
- ☑ Dumping process memory allows the investigator to collect not only the memory used by the process that can be found in RAM but the memory located in the page file as well.
- ☑ Once process memory has been collected, additional information about the process, such as open handles and loaded modules, can then be collected.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the "Ask the Author" form.

- Q:** Why should I dump the contents of RAM from a live system? What use does this have, and what potentially useful or important information will be available to me?
- A:** As we discussed in Chapter 1, a significant amount of information available on a live system can be extremely important to an investigation. This volatile information exists in memory, or RAM, while the system is running. We can use various third-party tools (discussed in Chapter 1) to collect this information, but it might be important to collect the entire contents of memory so that we not only have a complete record of information available, we can also "see" those things that might not be "visible" via traditional means (for example, things hidden by a rootkit; see Chapter 7, "Rootkits and Rootkit Detection," for more

information regarding rootkits). You might also find information regarding exited processes as well as process remnants left over after the system was rebooted.

Q: Once I've dumped the contents of RAM, what can I then do to analyze them?

A: Investigators have historically used standard file-based search tools to “analyze” RAM dumps. *Strings.exe* and *grep* searches have been used to locate passwords, e-mail addresses, URLs, and the like within RAM dumps. Tools now exist to parse RAM dumps for processes, process details (command lines, handles) threads, and other objects as well as extract executable images, which is extremely beneficial to malware analysis (see Chapter 6, “Executable File Analysis,” for more information on this topic) as well as more traditional computer forensic examinations.

Q: I have an issue in which a person is missing. On examination of a computer system in their home, I found an active instant messaging (IM) application window open on the desktop. When I scrolled back through the window and reviewed the conversation, it is clear that there could be useful information available from that process. What can I do?

A: If the issue you're faced with is primarily one that centers around a single visible process, dumping the entire contents of physical memory might not be necessary. One useful approach would be to dump the contents of process memory, then use other tools to extract specific information about the process, such as loaded modules, command line used to launch the process, or open handles. Once all information is collected, the next step could be saving the contents of the IM conversation. After all pertinent information has been collected, searching the contents of process memory for remnants of a previous conversation or other data might provide you with useful clues.