Chapter 3

Windows Memory Analysis

Solutions in this chapter:

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

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

In Chapter 1, we discussed collecting volatile data from a live, running Windows system. From the order of volatility listed in RFC 3227, we saw that one of the first items 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, have 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, even today. This field of research has really opened up in the past several years, beginning in summer 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 collect and analyze the contents of RAM? Until now, some investigators have collected the contents of RAM in the hope of finding something they wouldn't find on the hard drive during a postmortem 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.

In 2008, Greg Hoglund (perhaps best known for writing the first viable rootkit for Windows systems and rootkit.com, but also as the CEO of HBGary, Inc.) wrote a short paper titled "The Value of Physical Memory Analysis for Incident Response" (www.hbgary.com/resources.html). In that paper, Greg describes what can be extracted from the contents of physical memory, and perhaps most importantly, the value of that information with respect to addressing issues in incident response and computer 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, Internet Protocol (IP) addresses, e-mail addresses, or other strings that could give the analyst an investigative lead. The drawback of this 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 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 summer 2005, the Digital Forensic Research Workshop (DFRWS; www.dfrws.org) issued a "memory analysis challenge" in order "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 Helix 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 Schuster (http://computer.forensikblog.de/en/topics/windows/memory_analysis) 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 (www.secureworks.com/research/tools/truman.html), which allows you to extract the memory used by a process from a dump of memory (important for malware analysis). Mariusz Burdach 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. 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 summer 2006, Tim Vidas, (http://nucia.unomaha.edu/tvidas/), a senior research fellow at Nebraska University, released procloc.pl, a Perl script to locate processes in RAM dumps as well as crash dumps.

Since then, the field of study with respect to collecting and analyzing memory dumps has grown by leaps and bounds, and in many instances the key figures have risen to the top. Perhaps the most notable individual in the field of memory analysis is Aaron Walters, co-creator of the FATKit (http://4tphi.net/fatkit/) and the Volatility Framework (https://www.volatilesystems.com/default/volatility). Although tools have been released to allow for collecting the contents of physical memory from Windows XP and Vista systems (addressed in detail in this chapter), Aaron and his co-developer, Nick L. Petroni Jr., have focused primarily on providing a framework for analysis of memory dumps. Aaron and Nick have been assisted by Brendan Dolan-Gavitt (you can find Brendan's blog at http://moyix.blogspot.com/, and his graduate research page is www.cc.gatech.edu/~brendan/) and others who

have made significant contributions to the area of memory analysis and specifically to the Volatility Framework. Matthieu Suiche (www.msuiche.net/) has contributed a program for dumping the contents of physical memory, and has also published information and tools for parsing Windows hibernation files (which have been incorporated into the Volatility Framework). Andreas Schuster (http://computer.forensikblog.de/en/) has continued his contributions to parsing of Windows memory dumps, including the release of a PTFinder (discussed later in this chapter) for Vista in November 2008, which is included as part of the PTFinder collection of Perl scripts.

In addition to free, open source tools for parsing Windows memory dumps, the security company Mandiant (the company Web site is www.mandiant.com, and the company blog is http://blog.mandiant.com/) released its Memoryze memory collection and parsing tool, along with the Audit Viewer tool for better presentation of the results of Memoryze. In addition, the folks at HBGary (www.hbgary.com/) have released their own memory analysis application called Responder, which comes in Professional and Field editions. The HBGary Web site includes a wealth of information about the tools, including videos demonstrating how to use them.

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 also what is in virtual memory or the page file.

To do this, a couple of tools are available. One is pmdump.exe (www.ntsecurity.nu/toolbox/pmdump/), 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 Microsoft 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 (available in both Linux and Windows versions from www.trapkit.de/research/forensic/pd/index.html). 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 graphical user interface (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 (www.trapkit.de/research/sslkeyfinder/index.html).

TIP

Jeff Bryner wrote pdgmail (you can find the pdgmail tool at www.jeffbryner.com/code/pdgmail), a Python-based tool that uses the output of Tobias's Process Dumper program to search for Gmail artifacts (e.g., contacts, last access records, account names, etc.). Jeff posted to the SANS Forensic blog (http://sansforensics.wordpress.com) regarding pdgmail, stating that he had not tested it on the Windows platform. Jeff subsequently released pdymail for extracting Yahoo! mail remnants (http://jeffbryner.com/code/pdymail) from process memory, as well. In this section of the chapter, we discuss dumping the contents of process memory, whereas later in the chapter we discuss how an analyst can dump the contents of process memory for a full dump of physical memory. You can use either method to retrieve data for use with pdgmail or pdymail.

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 Microsoft 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 memory used by a process is to use the adplus.vbs script that ships with the debugging tools, as this methodology attaches a debugger to the process and suspends the process to dump it. *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 MS Debugging Tools contains a great deal of information about the script as well as the cdb.exe debugging tool, which it uses to dump the processes you designate. The MS Debugging Tools do not require that an additional driver be installed, and they can be run from a CD. This means the tools (adplus.vbs and cdb.exe, as well as

supporting dynamic link libraries, or 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 use the freely available MS 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. Also, after dumping the process, you can use still other tools (such as those discussed in Chapter 1) to collect additional information about the process from the running system, which may provide a quicker view into the details of the process itself. Handle.exe (which is available from http://technet.microsoft.com/en-us/sysinternals/bb896655.aspx and requires that you have Administrator rights on the system when running it) will provide you with a list of handles (to files, directories, etc.) that have been opened by the process, and listdlls.exe (Version 2.25 at the time of this writing, available from http://technet.microsoft.com/en-us/sysinternals/bb896656.aspx) will show you the full path to and the version numbers of the various modules loaded by a process.

Extensive help is available for using adplus.vbs, not only in the MS Debugging Tools help file but also in Microsoft Knowledge Base article 286350 (http://support.microsoft.com/kb/286350). You can use adplus.vbs to hang the process while it is being dumped (i.e., 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. (You can change the location where the output is written using the *–o* switch.) You will see 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, you can use this tool only on processes that are visible via the application program interface (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.

You can use the *volatility memdmp* command (we discuss the Volatility Framework later in the chapter) to dump the addressable memory for a process from a Windows XP memory dump, as follows:

```
D:\Volatility>python volatility memdmp -f d:\hacking\xp-laptop1.img -p 4012
```

This command results in a 4012.dmp file that is 118,300,672 bytes in size.

Dumping Physical Memory

So, how do you go about collecting the full contents of physical memory, rather than just dumping a process? Several methods are available, each with 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.

DD

DD (http://en.wikipedia.org/wiki/Dd_(Unix)) is the short name given to a powerful tool from the UNIX world 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 was part of the Forensic Acquisition Utilities, but is no longer available. This utility was able to collect the contents of physical memory by accessing the *\Device\PhysicalMemory* object from user mode. The following command line could be used to capture the contents of RAM in the file ram.img on a network share or a USB thumb drive attached to the system:

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

In the preceding command line, the block size (*bs* value) is set to 4 kilobytes (KB) or 4096 bytes, as this is the default size for pages in memory. Therefore, the command tells dd.exe to grab one page at a time. This version of dd.exe also allows compression and the generation of cryptographic hashes for the content. Due to the volatile nature of RAM (i.e., it continues to change throughout the process of dumping it), however, it is not advisable to hash it until it is written from the disk, simply because there is no advantage in doing so. 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 employ a cryptographic hash to address the integrity of the collected memory dump.

WARNING

The version of dd.exe from George M. Garner, Jr., discussed earlier, is no longer available or supported, but there may be responders out there who still have a copy of the tool on a CD or hard drive someplace (I do!). Discussion of and reference to the tool is provided here for completeness, as well as to recognize George's contributions to the field of memory acquisition and analysis.

Nigilant32

Other tools use a process similar to dd.exe to capture the contents of RAM. Nigilant32 (www.agilerm.net/publications_4.html), from Matt Shannon of Agile Risk Management, uses a graphical interface to allow the responder to acquire the contents of physical memory. Figure 3.1 illustrates a portion of the Nigilant32 GUI, showing the option for imaging physical memory.

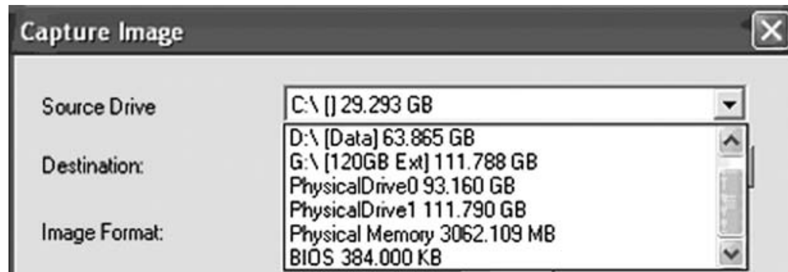
Figure 3.1 Portion of the Nigilant32 GUI

Nigilant32 has the advantage of a GUI, which may be a friendlier tool for some responders to use. You can deploy Nigilant32 on a CD or USB thumb drive along with other tools, and use it to quickly collect the contents of physical memory (primarily from Windows XP systems). As with dd.exe and other tools, Nigilant32 requires local access to the system from which the responder wishes to dump memory. You also can deploy Nigilant32 prior to an actual incident occurring, and responders can either run the tool locally and dump physical memory to a thumb drive or network share, or access the system remotely (via applications such as VNC or Remote Desktop Client) and perform the memory dump.

ProDiscover

ProDiscover IR (Version 4.8 was used in writing this book, and Version 5.0 was released in summer 2008) also 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. Figure 3.2 illustrates the user interface for this capability.

Figure 3.2 Excerpt of the Capture Image Dialog Box from ProDiscover IR



KnTDD

The problem with using tools such as dd.exe or Nigilant32, however, is that as of Windows 2003 SP1, access to the *\Device\PhysicalMemory* object has been restricted from user mode (http://technet.microsoft.com/en-us/library/cc787565.aspx). That is, only kernel drivers are allowed to access this object. As such, tools such as dd.exe, Nigilant32, and ProDiscover (as previously mentioned, the Incident Response edition of ProDiscover includes a servlet that will allow an analyst to access physical memory on a remote Windows XP system) will not allow you to collect the contents of physical memory from Windows 2003 SP1 and later systems, including Vista. To address this issue, George M. Garner, Jr. (a.k.a. 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, including AMD64 versions of the operating systems
- Able to convert a raw memory "image" to Microsoft crash dump format so that the resultant 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 Transmission Control Protocol/Internet Protocol (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 Secure Sockets Layer (SSL), with cryptographic verification of the binaries before they are executed

One of the aspects of using dd.exe, and tools like it, that you need to keep in mind is Locard's Exchange Principle. To use these tools to collect the contents of RAM, they must be loaded into RAM as a running process. This means memory space is consumed and other processes may have pages 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 through 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?

Under most incident response conditions, tools such as dd.exe might be the best method for retrieving the contents of physical memory, particularly from Windows 2000 and XP systems. Such tools do not require that the system be taken down, nor do they restrict how and to where the contents of physical memory are written (e.g., using netcat, you can write the contents of RAM out over a socket to another system rather than to the local hard drive). Tools have been developed and made freely available (discussed later in this chapter) to parse the contents of these RAM dumps to extract information about processes, network connections, and the like. Further, development of the KnTTools and other similar tools allows for continued support of this methodology beyond Windows 2003 SP1.

NOTE

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, 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 traditional "forensic" methodologies. Interacting with a live system, however, may be the only manner with which certain information can be retrieved. Keeping Locard's Exchange Principle in mind, responders must thoroughly and clearly document their actions when performing live response. Issues such as the need for live response, as well as the "footprint" of live-response tools, have been (and will likely continue to be) the subject of discussion for some time.

MDD

The early part of 2008 saw the release of several new tools for collecting the contents of physical memory from Windows systems, particularly Windows 2003 SP1 and Vista. Fortunately, these tools work equally well on Windows XP systems, making them more universal than previously available tools (e.g., dd.exe, Nigilant32, etc.). Perhaps the most notable was mdd.exe which was released to the public by ManTech International (www.mantech.com/msma/MDD.asp). Mdd.exe is a straightforward and simple command-line interface (CLI) tool, allowing a responder to dump the contents of physical memory with a simple command line:

```
E:\response>mdd.exe -o F:\system1\memory.dmp
```

The preceding command line illustrates an example of mdd.exe run from a CD (E:\), sending the output file (via the *–o* switch) to a USB external hard drive (F:\). You can also run mdd.exe from a batch file, as I described in Chapter 1, using a command line such as:

```
mdd.exe -o %1\mdd-o.img
```

Alternatively, you can use the available variables from a live Windows environment to segregate your collected volatile data by prepending the name of the system from which you're collecting data:

```
mdd.exe -o %1\%ComputerName%-mdd-o.img
```

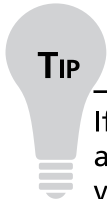
Once mdd.exe completes the memory dump, it displays an MD5 checksum for the resultant dump file:

```
took 108 seconds to write
MD5 is: 6fe975ee3ab878211d3be3279e948649
```

The analyst can save this information and use it to ensure the integrity of the memory dump file later.

Not only is mdd.exe simple to use and deploy, but also the output of the tool is what is referred to as a raw, dd-style memory dump file, similar to what is achieved using other tools (dd.exe, VMware, etc.). However, prior to using mdd.exe, you should be aware that,

according to the ManTech International Web site for the tool, mdd.exe is unable to collect more than 4 GB of RAM. On systems with 8 GB or more of RAM, some users have reported system crashes, so take this into account when considering whether to collect the contents of RAM from a system.



Tip

If you intend to collect the contents of RAM from a live Windows system as part of your response methodology using a batch file, I recommend that you collect the contents of RAM first. This will provide you with as "clean" a dump as possible, particularly if you've included other third-party (tlist.exe, tcpvcon.exe) and native Windows tools (netstat.exe) in your batch file.

Win32dd

Matthieu Suiche released his own tool for dumping the contents of physical memory, called win32dd (http://win32dd.msuiche.net/). Win32dd is described as a "free kernel land and 100% open-source tool"; this means that like mdd.exe, win32dd.exe is free, but unlike ManTech's tool, win32dd.exe is open source. Win32dd.exe has some additional features, including the ability to create a WinDbg-compatible "crash dump" (similar to a Windows crash dump) which you can then analyze using the Microsoft Debugging Tools (www.microsoft.com/whdc/DevTools/Debugging/default.mspx). As with mdd.exe and other CLI tools, win32dd.exe can be included in batch files.

Version 1.2.1.20090106 of win32dd.exe was available at the time this chapter was being written. You can view the syntax information available for win32dd.exe using the following command:

```
D:\tools\win32dd>win32dd -h

   Win32dd - v1.2.1.20090106 - Kernel land physical memory acquisition
   Copyright (c) 2007 - 2009, Matthieu Suiche <http://www.msuiche.net>
   Copyright (c) 2008 - 2009, MoonSols <http://www.moonsols.com>

Usage:
   win32dd [option] [output path]

Option:
   -r    Create a raw memory dump/snapshot. (default)
   -l    Level for the mapping (with -r option only).
      l 0    Open \\Device\\PhysicalMemory device (default).
      l 1    Use Kernel API MmMapIoSpace()
```

```
    -d   Create a Microsoft full memory dump file (WinDbg compliant).
    -t   Type of MSFT dump file (with -d option only).
      t 0   Original MmPhysicalMemoryBlock, like BSOD. (default).
      t 1   MmPhysicalMemoryBlock (with PFN 0).
    -h   Display this help.
Sample:
Usage: win32dd -d physmem.dmp
Usage: win32dd -l 1 -r C:\dump\physmem.bin
```

As you can see, win32dd.exe has a number of options available for dumping the contents of physical memory. To create a raw, dd-style memory dump file, you can use the following command line:

```
D:\tools\win32dd>win32dd -l 0 -r d:\tools\memtest\win32dd-l0-xp.img
```

Including the command-line options (such as *–l 0*) in the name of the output file serves as a modicum of additional documentation for the analyst, to identify the command-line switches used. As with other CLI tools, including such a command in a batch file is simple and straightforward (besides being self-documenting).

Memoryze

The consulting company Mandiant released its own memory collection and analysis tool, called Memoryze (www.mandiant.com/software/memoryze.htm). Memoryze finds its origins in the Mandiant Intelligent Response (MIR) product, and has been made freely available from the Mandiant Web site, along with examples of how to run the tool. Once you've downloaded the Memoryze Microsoft installer (MSI) file to your system and installed it, you can run Memoryze to collect a memory dump via the memorydd.bat batch file by typing the following command:

```
D:\Mandiant\memorydd.bat
```

This will create a memory dump file in a directory structure within the current directory; running the command as is on my system created the directory structure Audits\WINTERMUTE\20090103003442\, and within that final directory it created several XML files and a memory image file. Running the batch file with the *–output* switch will let you configure the location for the dump file.

Also, when run on a live system, Memoryze reportedly "makes use of the page file(s)," incorporating memory contents from the page file into the collection process. This allows for more complete collection of data, as memory contents that have been swapped out to the page file are now available during the analysis process. In addition, Memoryze Version 1.3.0 (announced February 9, 2009) is capable of dumping memory that is accessible via F-Response, which we'll discuss later in this chapter.

Winen

Guidance Software also released its own tool for collecting the contents of physical memory, called winen.exe. Like some of the other tools, winen.exe is a CLI tool, but unlike the other tools, the memory dump is not collected in raw, dd-style format; instead, it is collected in the same proprietary imaging format used by the EnCase image acquisition tool, commonly referred to as Expert Witness Format (or EWF, for short). Most available analysis tools require that the memory dump be in a raw, dd-style format, so memory dumps collected using winen.exe must be converted to a raw format prior to being parsed. As Lance Mueller points out in his blog (www.forensickb.com/2008/06/new-version-of-encase-includes-stand.html), you can run winen.exe by simply providing various options at the console (i.e., typing **winen** at the command prompt and then providing responses to the queries) or by providing the path to a configuration file with the necessary information via the *–f* switch. Winen.exe has a total of six required options, the settings for which can be provided via the command line or in a configuration file: the path to the output file(s), the compression level, the examiner's name, the evidence name, the case number, and the evidence number. An example configuration file is included in the home directory when you download and install EnCase.

Richard McQuown provides additional information about the use of winen.exe in his ForensicZone blog (http://forensiczone.blogspot.com/2008/06/winenexe-ram-imaging-tool-included-in.html), as well as a link to a user manual file for winen.exe (you must have access to the EnCase User Forum and be logged in to obtain the PDF document at https://support.guidancesoftware.com/forum/downloads.php?do=file&id=478).

You can then convert the resultant memory dump to a raw format by opening the memory dump file in FTK Imager and choosing **Create Disk Image** from the menu (http://windowsir.blogspot.com/2008/06/memory-collection-and-analysis-part-ii.html), or by opening the memory dump in EnCase and choosing **Copy/UnErase** from the EnCase menu.

Guidance also has a version of winen.exe for 64-bit versions of the Windows operating system, called winen64.exe. As with the 32-bit version of the tool, winen64.exe allows a responder to dump the contents of physical memory in an EWF format dump file.

Fastdump

Consulting company HBGary released a copy of its tool for collecting the contents of physical memory, called fastdump (www.hbgary.com/download_fastdump.html). Although fastdump.exe (Version 1.2 is available at the time of this writing) is free, as with Nigilant32 and some of the other available tools it is not able to collect memory contents from Windows 2003 SP1 and later systems, including Vista. You can use this tool only to collect the contents of physical memory from Windows XP and Windows 2003 (no service packs installed) systems.

To address this issue, HBGary also released a commercial version of the tool, called FastDump Pro (fdpro.exe), which is available from the same Web page as fastdump.exe, albeit for a fee. The commercial version supports all versions of Windows, including 32- and 64-bit versions, and will also reportedly dump memory from systems with more than 4 GB of RAM.

Besides being able to dump more than 4 GB of physical memory, FastDump Pro has some other differences from and advantages over other tools. Typing **fdpro** at the command prompt displays the syntax information for the tool:

```
D:\HBGary\bin\FastDump>fdpro
-= FDPro v1.3.0.377 by HBGary, Inc =-
***** Usage Help *****
________________________________________________________________________________
General Usage: fdpro output_dumpfile_path [options] [modifiers]
________________________________________________________________________________
FDPro supports dumping .bin and .hpak format files
________________________________________________________________________________
To dump physical memory only to literal .bin format:
        fdpro mymemdump.bin [options] [modifiers]
To dump physical memory to an .hpak formatted file:
        fdpro mysysdump.hpak [options] [modifiers]
________________________________________________________________________________
*** Valid .bin [options] Are: ***
-probe [all|smart|pid|help]     Pre-Dump Memory Probing
________________________________________________________________________________
*** Valid .bin [modifiers] Are: ***
-nodriver                       Use old-style memory acquisition (XP/2k only)
-driver                         Force driver based memory acquisition
________________________________________________________________________________
*** Valid .hpak [options] Are: ***
-probe [all|smart|pid|help]     Pre-Dump Memory Probing
-hpak [list|extract]            HPAK archive management
________________________________________________________________________________
*** Valid .hpak [modifiers] Are: ***
-nodriver                       Use old-style memory acquisition (XP/2k only)
-driver                         Force driver based memory acquisition
-compress                       Create archive compressed
-nocompress                     Create archive uncompressed
________________________________________________________________________________
```

As you can see, fdpro.exe has essentially two modes in which it can be used. The first mode is to dump the contents of physical memory in the usual raw, dd-style format, using either the driver to access physical memory on versions of Windows that require it (i.e., Windows 2003 SP1, Vista, and later), or the *–nodriver* switch to perform "old-style memory acquisition."

Fdpro.exe also has an .hpak-style format, which is described as follows:

> "HPAK is an HBGary proprietary format which is capable of several key features, namely the ability to store and archive the RAM and Pagefile in a single archive. HPAK format also supports compression using the gzip format. This is useful during instances where space on the collecting device/system is limited."

One of the limitations of most of the available tools for dumping physical memory is that they allow access only to physical memory, and do not incorporate the page file. Not only does fdpro.exe provide access to a wide range of Windows operating systems (including 64-bit versions) and memory capacities (i.e., greater than 4 GB), but also (as with Mandiant's Memoryze tool) the tool will incorporate the page file in the collection process, which then allows that data to be incorporated into the analysis process, as well (we will discuss the HBGary Responder product later in this chapter). As you will see later in this chapter, contents of memory that have been swapped out to the page file may contain information pertinent to your response and analysis.

WARNING

It is important to point out that proprietary formats for data collection often lead to the requirement to use one particular tool for data analysis. Memory dumped using winen.exe, for example, must be converted to a raw format prior to being analyzed using other tools (although EnScripts are available to perform some limited parsing of the dump file). The same holds true with HBGary's .hpak format, as well; at this point, only HBGary's Responder product can be used to analyze an .hpak format memory dump. This is something you must keep in mind when deciding which tool to use.

F-Response

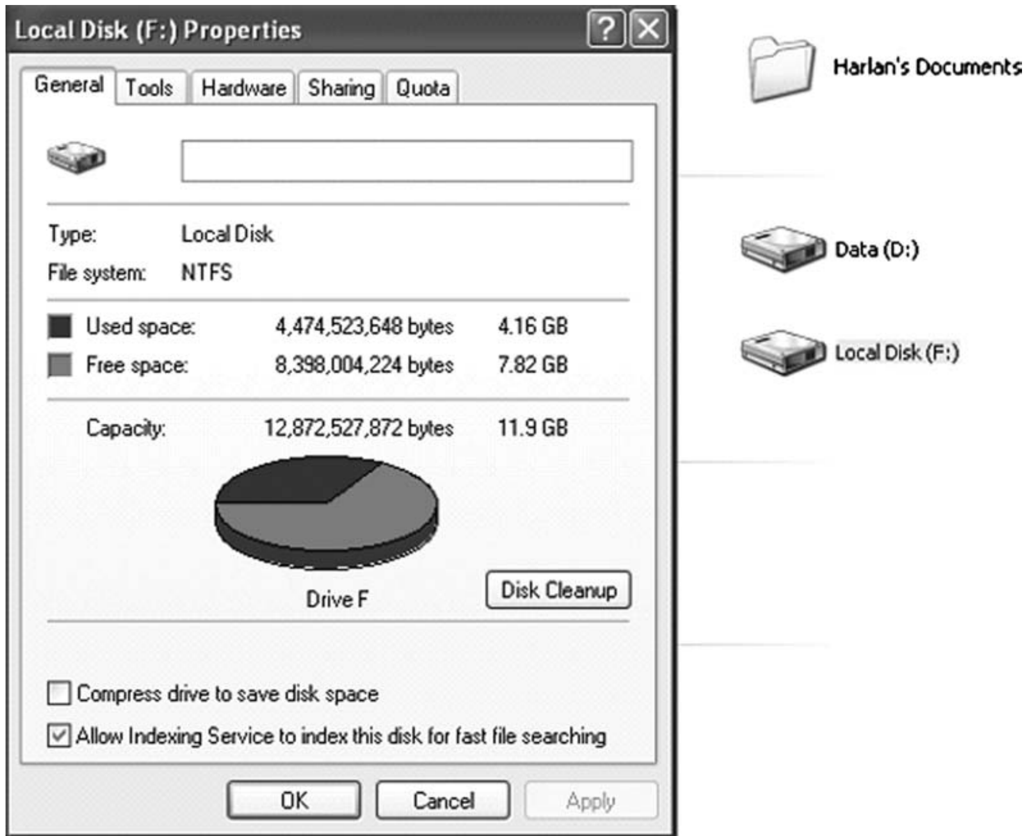
Finally, 2008 heralded the coming of a new age in incident response with Matt Shannon's release of F-Response. F-Response is an acquisition-tool-agnostic framework that uses the iSCSI protocol to provide raw access to disks over the network. Matt designed and wrote F-Response so that the access is read-only; write operations to the remote disk are buffered and silently dropped. The three editions of F-Response (Field Kit, Consultant, and Enterprise) are all deployed differently, but all allow you to "see" drives on remote

systems (e.g., in another room, on another floor, or even in another building) as mounted drives on your own system. F-Response is acquisition-tool-agnostic in the sense that once you're connected to the remote system and can "see" the drive, you can use any tool you want (dd.exe, ProDiscover, FTK Imager, etc.) to acquire an image of that drive. You can deploy F-Response on Mac OS X, Linux, and Windows systems, and on Windows systems it has the added benefit of providing remote, read-only access to physical memory. At the SANS Forensic Summit in October 2008, during his presentation with Aaron Walters, Matt announced the release of F-Response 2.03, which provides remote, real-time, read-only access to a Windows system's physical memory, for all versions of Windows, including XP and Vista. Once the connection to the remote system is set up, a responder can then use tools such as dd.exe or FTK Imager to acquire a copy of physical memory.

F-Response's capability (Version 2.06 beta was used in this example) for obtaining a copy of physical memory from a remote Windows system has tremendous implications for incident response, particularly when deploying the Enterprise Edition (see the "F-Response Enterprise Edition (EE)" sidebar for more information on deploying this capability in an enterprise environment). Matt has several videos linked to the F-Response blog (www.f-response.com/index.php?option=com_content&task=blogsection&id=4&Itemid=9) that demonstrate uses and aspects of the F-Response tool, such as illustrating how to acquire specific data, for example, by accessing a live Microsoft Exchange server.

At the SANS Forensic Summit in October 2008, Matt announced that F-Response would provide remote, read-only access to drives on Windows systems and to physical memory. Upon a successful connection, remote drives appear on the responder's system with the familiar drive icon, as Figure 3.3 illustrates.

Figure 3.3 Remote Windows System Drive Connected to As F:\



Tools & Traps…

F-Response Enterprise Edition (EE)

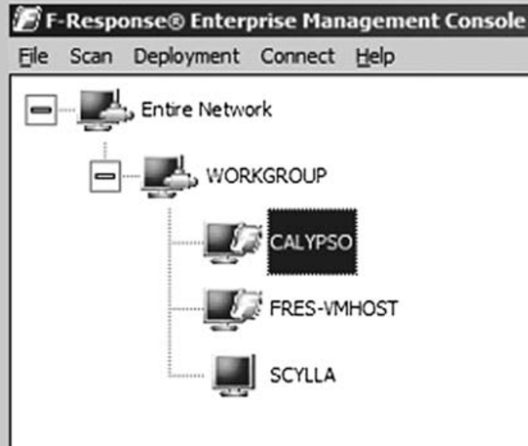
You can deploy F-Response Enterprise Edition (EE) agents in a proactive fashion. Deploying the EE agents ahead of time can significantly speed response time, as the agents install as a Windows service, but by default do *not* start automatically when the system is booted.

Continued

You also can install the agent in a stealthy manner, using tools such as psexec.exe (http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx) or xcmd.exe (http://feldkir.ch/xcmd.htm). The PDF document "Remote (Stealth) Deployment of F-Response EE on Windows Systems," located in the ch3 directory on the media that accompanies this book, clearly illustrates the necessary steps for deploying F-Response EE remotely (or in a stealthy manner) over the network.

In spring 2009, Matt Shannon released the F-Response Enterprise Management Console, or FEMC. The FEMC is a GUI-based tool that completely removes all of the effort required to manually deploy the Enterprise Edition. Clicking through the user interface, a responder (or consultant) can locate systems in the domain (or workgroup) on which to install F-Response EE, as Figure 3.4 illustrates.

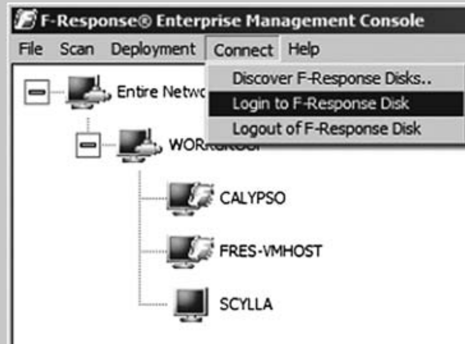
Figure 3.4 FEMC User Interface Populated with Systems



Once systems have been located and selected, deploying and starting the F-Response service is nothing more than a couple of mouse clicks away, as illustrated in part in Figure 3.5. This is the case whether you want to install it on one system or on a dozen systems.

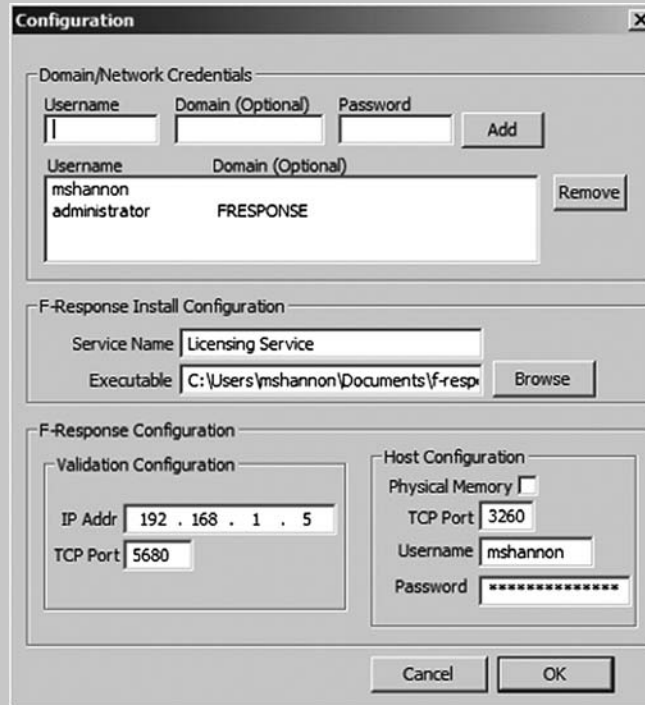
Continued

Figure 3.5 Logging in to F-Response EE via the FEMC User Interface



All the responder needs to do is select the systems onto which she wishes to deploy F-Response, and then with a couple of mouse clicks deploy the agent automatically. Even the configuration of the .ini file is handled automatically through the user interface, as Figure 3.6 illustrates.

Figure 3.6 Configuring F-Response EE via the FEMC User Interface



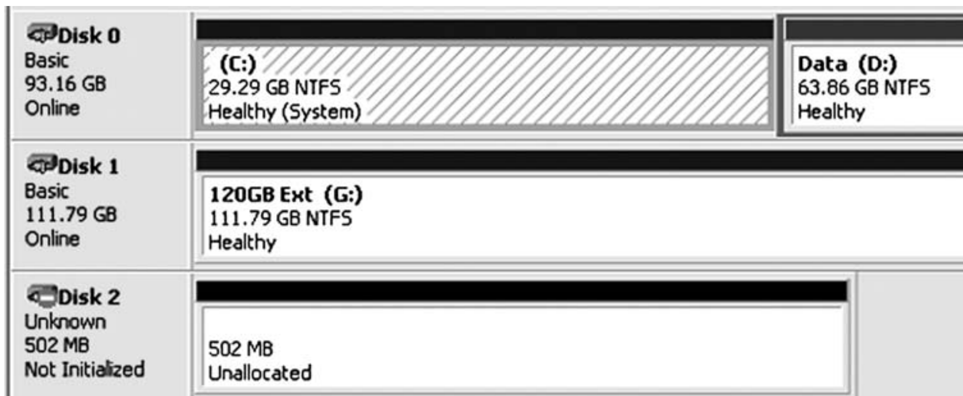
Continued

Information entered into the Domain/Network Credentials section of the Configuration dialog shown in Figure 3.6 is available throughout the duration of the session, but is *not* saved or preserved in any way when the FEMC is closed.

Working with the FEMC, it actually took me more effort to play BrickBreaker on my BlackBerry than it does to deploy F-Response EE across several systems, connect to each, and then completely uninstall the agent. The FEMC pushes the functionality of an extremely powerful and valuable tool forward by a quantum leap (not to take anything away from Scott Bakula…).

As physical memory (RAM) does not contain a partition table, the responder's system will not recognize the connection to the remote system's physical memory as a drive. However, the connection can be seen through the Disk Management Microsoft Management Console (MMC), as Figure 3.7 shows.

Figure 3.7 F-Response EE Connection to Remote RAM Seen via Disk Management MMC



Once the connection to the remote system's RAM has been verified, you can use tools such as FTK Imager to perform a memory dump. Figure 3.8 illustrates selecting the remote system's RAM via FTK Imager.

Figure 3.8 Selecting a Remote System's RAM via FTK Imager

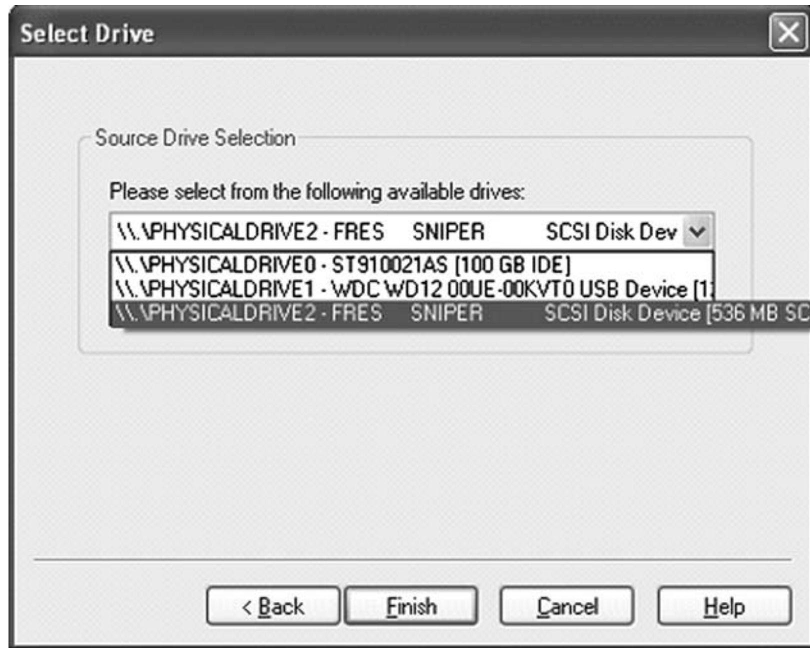
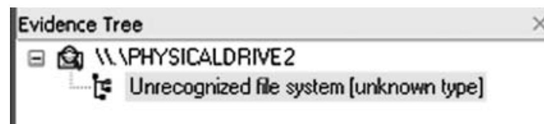


Figure 3.9 illustrates the selected RAM from Figure 3.8 as it appears in the FTK Imager Evidence Tree.

Figure 3.9 Selected RAM in FTK Imager Evidence Tree



The RAM from the remote system can now be acquired using FTK Imager. Other tools can also be used; F-Response is a tool-agnostic framework, as it does not require that you use a specific tool. A responder can use EnCase, X-Ways Forensics, or even versions of dd.exe to dump RAM from the remote system. Again, the connection is read-only, and as illustrated in Figures 3.8 and 3.9, the remote system's RAM appears on the responder's system as a *\\.\PhysicalDrive* object (i.e., *\\.\PhysicalDrive2* or *\\.\PhysicalDrive3*).

NOTE

At the SANS Forensic Summit in October 2008, Aaron Walters and Matt Shannon gave a presentation that incorporated the use of F-Response and the Volatility Framework in something called "Voltage" (http://volatilesystems.blogspot.com/2008/10/voltage-giving-investigators-power-to.html). Although not available at the time of this writing, Voltage is described as providing an unprecedented level of real-time, read-only, remote access, automation, and visualization to temporally relevant information that has not been available to date. For all the fancy words, Voltage provides a responder with the capability to quickly reach out to remote systems, identify systems that may have malware installed and running, and then collect a sample of physical memory, all without modifying the artifacts on the remote system.

Section Summary

John Sawyer posted to the SANS Forensics blog (http://sansforensics.wordpress.com/2008/12/13/windows-physical-memory-finding-the-right-tool-for-the-job/) on December 13, 2008, listing some of the various tools available for collecting (as well as analyzing) memory dumps from Windows systems, along with brief descriptions of each of them. In addition to John's blog post, other posts (to blogs and discussion lists) have made comments and addressed concerns about the availability and use of the various tools for dumping memory from Windows systems. In many ways, this chapter serves as an initial step into this realm of tools, as I do not doubt that between the time I submit this chapter to the publisher and the time the finished book is available the tools will have changed. This very thing happened between the time I submitted this chapter for technical review and the time I received the tech editor's comments—in the ensuing time, HBGary had released FastDump Pro and announced its availability to the public. I firmly believe that the landscape of memory dumping (and analysis) tools will change, doing so in relatively short order.

So far in this section you've seen some of the tools available for dumping the contents of physical memory from Windows systems, but little else. In an effort to provide more of a side-by-side comparison of some of these tools, I ran some of my own limited testing. Using the command lines (in the case of Memoryze, the memorydd.bat file) for each tool discussed previously in this chapter that might be run from a CD or USB thumb drive, I ran each of them (with the exception of winen.exe) on a Dell Latitude D820 system with 4 GB of RAM installed, running Windows XP Service Pack 3. My testing process was to simply boot the system, run one of the tools, and once the tool had completed its dump of memory,

reboot the system to run the next tool. In doing so, I saved the memory dumps so that I could see the sizes of each of them in relation to the dumps produced by other tools; my final results appear as follows, listed by the size in bytes and the output filename:

```
3,210,854,400 win32dd-l0-xp.img
3,210,854,400 win32dd-l1-xp.img
3,210,854,400 mdd-xp-2.img
3,210,854,400 mdd-xp.img
3,211,329,536 dd_xp.img
3,211,329,536 nigilant_xp.img
3,211,333,632 fdpro-xp.bin
3,211,334,904 fdpro-xp.hpak
3,211,329,536 memory.1e1d2b07.img (Memoryze)
```

As you can see, I included the name of the tool and some additional identifying information from the command line in the name of the output dump file. Also, it's clear that there are some variations in the sizes of the resultant dumps, likely due to the process each tool used. As expected, the exception to this is the .hpak format dump file produced by FastDump Pro, as the *–page* switch had been used to incorporate the page file in the dump process. The file size would likely have been significantly larger had there been greater activity on the system, and had the system been allowed to run longer.

It should be clear at this point that the tool you should use to collect the contents of physical memory really depends on a number of factors, including the version of Windows (not only Windows XP versus Vista, but also 32- versus 64-bit), the amount of physical memory installed in the system, and your analysis tools.

WARNING

As with using any tools that you've downloaded from the Internet, be sure that you've tested the tools and you've read and that you understand the license agreement regarding their use. Some tools, although extremely powerful and useful, cannot be used by consultants. Also, you need to be aware of the artifacts left by various tools. As discussed in Chapter 1, many of the tools available from Microsoft (formerly Sysinternals) create Registry entries when run. Winen.exe does something similar, in that it will write its driver (winen_.sys) to the same directory from which the file is run, and will create a Services subkey within the Registry. The key here, as with all live-response activities, is not to avoid using a tool because it leaves "footprints" or artifacts of its use, but rather to understand this fact and incorporate that understanding into your methodology and documentation.

Alternative Approaches for Dumping Physical Memory

The software we've discussed so far aren't the only means by which the contents of physical memory can be dumped from a live system; several alternative methods have been put forth in the past. Some of those methods use native functionality inherent to the operating system (i.e., crash dumps) or to virtualization platforms (i.e., VMware). As we'll see, other alternative methods for dumping physical memory from a system take a more physical approach.

Hardware Devices

In February 2004, the *Digital Investigation Journal* published a paper by Brian Carrier and Joe Grand 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 or 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 Payment Card Industry (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 (referred to earlier in this chapter), 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 the attacker has subverted. Let's examine the pros and cons of such a device:

Hardware devices such as 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. However, 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 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 Boileau (see www.storm.net.nz/projects/16) came up with a way to extract physical memory from a system using Linux and Python. 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 in to 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 walks through the process of identifying FireWire ports and tricking the target Windows system into "thinking" 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:

Many systems available today have FireWire/IEEE 1394 interfaces built right into the motherboards, increasing the potential accessibility of physical memory using this method. However, Arne Vidstrom has pointed out some technical issues (see http://ntsecurity.nu/onmymind/2006/2006-09-02.html) 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

We've all seen crash dumps at one point or another; in most cases they manifest themselves as an infamous Blue Screen of Death (a.k.a. BSoD, with more descriptive information available at http://en.wikipedia.org/wiki/Blue_Screen_of_Death). 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. This is because when a crash dump occurs, the system state is frozen and the contents of RAM (along with about 4 Kb 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, 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 Microsoft Debugging Tools (www.microsoft.com/whdc/devtools/debugging/default.mspx) but also in the Kernel Memory Space Analyzer (a tool with a ridiculously long URL, so it's best to do a search for it), which is 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 your investigation on the system), there are no limitations to this approach, right? Under some circumstances, this is true … 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 Microsoft Knowledge Base article Q254649 (http://support.microsoft.com/kb/254649). This article lists the three types of crash dumps: small (64 KB), kernel, and complete crash dumps. We're looking for the complete crash dump because it contains the complete contents of RAM. The 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.

NOTE

Microsoft Knowledge Base article 235496 (http://support.microsoft.com/kb/235496) specifies the Registry entries that contain configuration information for Windows systems with respect to the memory.dmp file that is the result of a crash dump. The default location for the memory.dmp file is %SystemRoot%\memory.dmp, but the administrator can specify another location via the *DumpFile* Registry value.

Along the same lines, Microsoft Knowledge Base article Q274598 (http://support.microsoft.com/kb/274598) states that complete crash dumps are not available on systems with more than 2 GB 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 + 1 MB) as well as the time it will take to complete the crash dump process.

Microsoft Knowledge Base article Q307973 (http://support.microsoft.com/kb/307973) describes how to set the full range of system failure and recovery options. These settings are more for system administrators and information technology (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 numbers, Social Security numbers, or the like, and these crash dumps have been found to contain that sensitive 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 the system failure and recovery configuration options on a system 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 Knowledge Base article Q244139, available at http://support.microsoft.com/kb/244139) 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:

Dumping memory via a crash dump is perhaps the only technically accurate (albeit not "forensically sound") method for creating an image of the contents of RAM. This is because 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 (overwriting data). Further, Microsoft provides debugging tools as well as the Kernel Memory Space Analyzer (which consists of an engine, plug-ins, and user interface) for analyzing crash dump files. Some Windows systems do not generate full crash dumps by default (e.g., Vista RC1; 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 mini dump 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 Knowledge Base 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

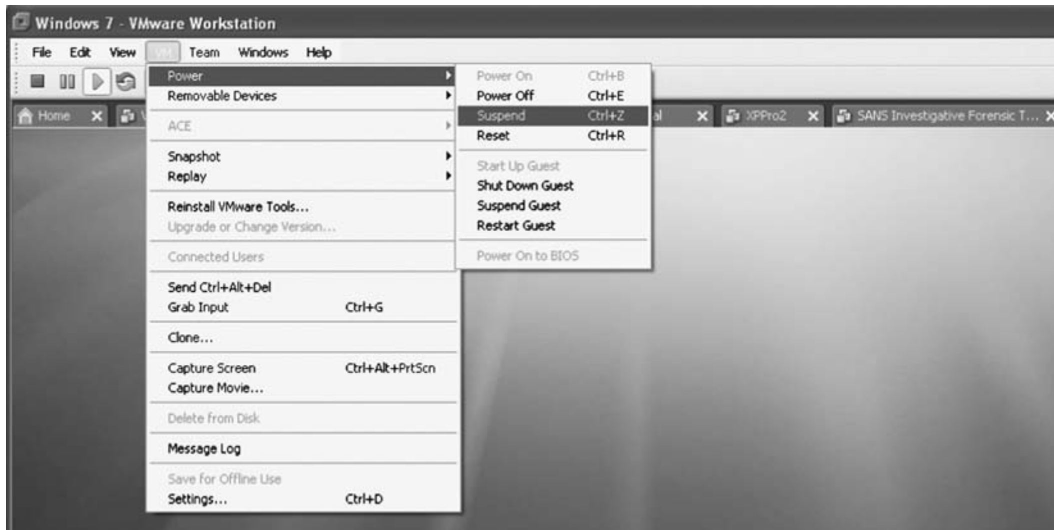
A support article available at the Citrix Web site (http://support.citrix.com/article/CTX107717&parentCategoryID=617) provides a methodology for using livekd.exe (http://technet.microsoft.com/en-us/sysinternals/bb897415.aspx) and the Microsoft Debugging Tools to generate a full kernel dump of physical 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 because the dump can take a considerable amount of time and 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 3.10 illustrates the menu items for suspending a VMware session.

Figure 3.10 Menu Items for Suspending a Session in VMware Workstation 6.5



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 raw, dd-style memory, and in fact, it can be parsed and analyzed with the same tools discussed previously in the chapter.

VMware isn't the only virtualization product available. Others include VirtualPC from Microsoft, as well as the freeware Bochs (http://bochs.sourceforge.net/). None of these virtualization products have been tested to determine whether they can generate dumps of physical memory; however, if this is an option available to you, suspending a VMware session to obtain a dump of physical memory is quick and easy and minimizes your interaction with and impact on the system.

TIP

In May 2006, Brett Shavers wrote an excellent article for the ForensicFocus Web site, titled "VMware as a forensic tool" (available at www.forensicfocus.com/vmware-forensic-tool). Brett followed that article in 2008 with the paper "Virtual Forensics: A Discussion of Virtual Machines Related to Forensics Analysis" (www.forensicfocus.com/downloads/virtual-machines-forensics-analysis.pdf), which is by far the best treatment of the topic that I've been able to find anywhere.

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 NT Loader (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, etc.). This functionality is most often found on laptop systems. Here are the pros and cons:

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. Matthieu Suiche decoded the hibernation file format and presented his findings at the BlackHat USA 2008 conference (www.blackhat.com/presentations/bh-usa-08/Suiche/BH_US_08_Suiche_Windows_hibernation.pdf). This presentation followed his earlier write-up on the subject from February 2008 titled "Sandman Project" (http://sandman.msuiche.net/docs/SandMan_Project.pdf), and his previous work from 2007 with Nicolas Ruff. The Sandman tool is available from www.msuiche.net/category/sandman/, and the functionality is incorporated into the Volatility Framework (discussed later in this chapter).

Something else to consider about hibernation files is that this functionality may be the only option available for capturing the full contents of physical memory. Some of the available tools for collecting the contents of physical memory may have "issues" (more specifically, they may have caused systems with more than 4 GB of physical memory—including 64-bit systems—to crash) or may not be available or feasible for use in some environments. Using powercfg.exe to enable hibernation mode (if it is not already enabled) and then using some other mechanism or tool to force the system to hibernate may be the only option for obtaining a memory dump. You can force a system with hibernation enabled to hibernate using Microsoft's psshutdown.exe (http://technet.microsoft.com/en-us/sysinternals/bb897541.aspx) utility with the *–h* option, or by creating a batch file that contains the line:

```
%windir%\System32\rundll32.exe powrprof.dll,SetSuspendState Hibernate
```

The hibernation file is compressed and in most cases will not contain the *current* contents of memory. Thanks to the work performed by Matthieu and Nicolas, the hibernation file can be accessed in the same manner as a live memory dump, so I can't really think of any "cons" to using it. In addition to dumping memory from a live system, having a hibernation file available will give you memory contents from a previous point in time against which to compare your memory dump.

Analyzing a Physical Memory Dump

Now that you have seen ways to acquire the contents of RAM from a system (both local and remote methods), what can you do with the memory dump? For the most part, prior to summer 2005, the standard operating procedure for most folks who had bothered to collect a memory dump (usually via the version of dd.exe available with George M. Garner, Jr.'s Forensic Acquisition Utilities) was to run strings.exe against it, or run *grep* searches (for e-mail addresses, IP addresses, etc.), or both. Although this would result in investigative leads (finding what appeared to be a password geographically "close" to a username might give an investigator a clue or something to examine further) that would often lead to something definitive, it does not provide overall context to the information that is discovered. 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 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.

In 2007, Aaron Walters released the Volatility Framework (https://www.volatilesystems.com/), an open source, Python-based framework for parsing memory dumps from (at the time of this writing) Windows XP systems. During summer 2008, Aaron held the first Open Memory Forensics Workshop (https://www.volatilesystems.com/default/omfw) just prior to the DFRWS 2008 conference, during which researchers, analysts, and practitioners could rub elbows and discuss issues surrounding memory acquisition and analysis. Version 1.3 of the Volatility Framework was available at the conference, and was even used during the DFRWS Forensic Rodeo exercise.

Throughout the rest of this chapter, we will look at these tools for performing analysis of memory dumps. We will initially use the memory dumps from the DFRWS 2005 Memory Challenge as exemplars, for examples and demonstrations of tools and techniques for parsing Windows 2000 memory dumps. You're probably asking yourself, why even bother with that? Windows 2000 is the 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.

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 got "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") awhile 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 in particular the information that's displayed as it starts up (shown here 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 resultant physical location within the dump file. If the first two bytes located at that address are *MZ*, we could have a full-blown Windows portable executable (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. Because 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 whether there was a way to shorten and clarify the output, so I made some modifications to the file (changed the output, added some switches, etc.) and renamed it.

In its simplest form, you can 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 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
```

Running this script against other memory dumps mentioned previously in this chapter illustrates how well it works across various versions of Windows. For example, when run against the memory dump used in the Memoryze example, we see the following:

```
C:\Perl\memory>osid.pl -f d:\hacking\boomer-win2003.img -v
OS       : 2003
Product  : Microsoft« Windows« Operating System ver 5.2.3790.0
```

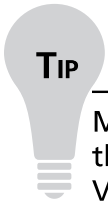
Running the script against another Windows 2003 memory dump gives us slightly different results:

```
C:\Perl\memory>osid.pl -f d:\hacking\win2k3sp1_physmem.img -v
OS       : 2003SP1
Product  : Microsoft« Windows« Operating System ver 5.2.3790.1830
```

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
```

I think that more than anything else, this demonstrates the utility of scripts or tools such as this, as it provides an analyst with the ability to more completely document the various items to be analyzed, particularly when such things may not have been completely documented during the response activities when data was initially collected.



Most analysis tools, discussed later in this chapter, are capable of performing the same function as was just discussed. For example, you can use the Volatility *ident* command to identify the operating system of a memory dump.

Process Basics

Throughout this chapter, we will focus primarily on parsing information regarding processes from a memory 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. To that end, we need to have a pretty good idea of what a process "looks like" in memory. The following section focuses on processes in Windows 2000 memory, but most of the concepts remain the same for all versions of Windows. The biggest difference is in the actual structure of the process itself, and going into the details of the process structures on all versions of Windows is beyond the scope of this book.

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, with each sequence having 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 (e.g., Windows 2000

to XP) but also between service packs of the same version of the operating system (Windows XP to XP SP2).

Andreas Schuster has done a great job of documenting the EProcess block structures in his blog (http://computer.forensikblog.de/en/topics/windows/memory_analysis). 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 Tools 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, because by now, Mark Russinovich has long since been in the employ of 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 "looks 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 forensic/investigative aspect) elements of the structure as we progress through this chapter.

NOTE

The Windows kernel keeps track of active processes by way of a doubly linked list; this means that each process "points to" both the process after it and the process before it, in a circular list. The operating system enumerates a list of active processes by walking *PsActiveProcessList* and developing a list of known active processes. Within the EProcess structure is a *LIST_ENTRY* item named *ActiveProcessLinks*. This entry has two values, *flink* and *blink*, which are pointers to the next and previous processes, respectively. Many memory analysis tools will do the same thing (i.e., walk the list of active processes) in a memory dump file, whereas others will perform a brute force enumeration of process objects (e.g., lsproc.pl and Volatility), enumerating even exited processes. This is very important, as some rootkits (see Chapter 7) hide processes by unlinking their processes from this doubly linked list.

An important element of a process that the EProcess structure points to 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

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

Process Creation Mechanism

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

A number of steps are followed when a process is created. These steps can be broken down into six stages (taken from *Windows Internals*, 4th Edition, Chapter 6, by Russinovich and Solomon):

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) is checked to see whether 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 the 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 least be something to analyze.

Parsing Memory Dump 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 (e.g., all processes on Windows 2000 or XP had the same *Size* value) but changed between those versions (e.g., 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" (www.dfrws.org/2006/proceedings/2-Schuster.pdf), 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 (http://forensiczone.blogspot.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, but 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. Because 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. All of these scripts are provided on the accompanying DVD, and we'll discuss them here.

NOTE

The following tools (lsproc.pl, lspd.pl, lspi.pl, and lspm.pl) are designed to be used solely with Windows 2000 memory dumps. As we've discussed so far, there are significant changes in the EProcess structure format between the various versions of Windows (2000, XP, 2003, Vista, etc.). As such, significant work needs to be done to produce a single application that will allow you to parse memory dumps from all versions.

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 identifier (PPID), the process identifier (PID), the name of the process, the offset of the process structure within the dump file, and the creation time of the process. Here is an excerpt of the lsproc.pl output:

```
Proc    820     324    helix.exe          0x00306020   Sun    Jun 5   14:09:27   2005
Proc      0       0    Idle               0x0046d160
Proc    600     668    UMGR32.EXE         0x0095f020   Sun    Jun 5   00:55:08   2005
Proc    324    1112    cmd2k.exe          0x00dcc020   Sun    Jun 5   14:14:25   2005
Proc    668     784    dfrws2005.exe(x)   0x00e1fb60   Sun    Jun 5   01:00:53   2005
Proc    156     176    winlogon.exe       0x01045d60   Sun    Jun 5   00:32:44   2005
Proc    156     176    winlogon.exe       0x01048140   Sat    Jun 4   23:36:31   2005
```

```
Proc    144     164    winlogon.exe         0x0104ca00   Fri   Jun 3   01:25:54   2005
Proc    156     180    csrss.exe            0x01286480   Sun   Jun 5   00:32:43   2005
Proc    144     168    csrss.exe            0x01297b40   Fri   Jun 3   01:25:53   2005
Proc      8     156    smss.exe             0x012b62c0   Sun   Jun 5   00:32:40   2005
Proc      0       8    System               0x0141dc60
Proc    668     784    dfrws2005.exe(x)     0x016a9b60   Sun   Jun 5   01:00:53   2005
Proc   1112    1152    dd.exe(x)            0x019d1980   Sun   Jun 5   14:14:38   2005
Proc    228     592    dfrws2005.exe        0x02138640   Sun   Jun 5   01:00:53   2005
Proc    820    1076    cmd.exe              0x02138c40   Sun   Jun 5   00:35:18   2005
Proc    240     788    metasploit.exe(x)    0x02686cc0   Sun   Jun 5   00:38:37   2005
Proc    820     964    Apoint.exe           0x02b84400   Sun   Jun 5   00:33:57   2005
Proc    820     972    HKserv.exe           0x02bf86e0   Sun   Jun 5   00:33:57   2005
Proc    820     988    DragDrop.exe         0x02c46020   Sun   Jun 5   00:33:57   2005
Proc    820    1008    alogserv.exe         0x02e7ea20   Sun   Jun 5   00:33:57   2005
Proc    820     972    HKserv.exe           0x02f806e0   Sun   Jun 5   00:33:57   2005
Proc    820    1012    tgcmd.exe            0x030826a0   Sun   Jun 5   00:33:58   2005
Proc    176     800    userinit.exe(x)      0x03e35020   Sun   Jun 5   00:33:52   2005
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 CD were used to acquire the memory dump.

The preceding listing shows only an excerpt of the lsproc.pl output. A total of 45 processes were 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.

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 (http://computer.forensikblog.de/en/2006/04/persistance_through_the_boot_process.html), the system boot time for the dump file was determined to be Sunday, January 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 (http://computer.forensikblog.de/en/2006/04/data_lifetime.html), 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 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 full path of the dump file and the physical offset of the process that you're interested in (the physical offset of the process within memory is obtained from the lsproc.pl output). Although lsproc.pl takes some time to parse through the contents of the dump file, lspd.pl is much quicker, because 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\dfrws1-mem.dmp 0x0414dd60
```

NOTE

The lsproc.pl output just shown is an excerpt of the entire output; I didn't list the entire output simply because the excerpt illustrates enough information for me to make my point. However, the process referenced in the lspd.pl command line (i.e., at offset 0x0414dd60) is not listed in that excerpt, although it is visible in the full output of lsproc.pl.

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)
```

NOTE

Earlier in the chapter, I mentioned that the list of active processes on a live system is maintained in a doubly linked list. The *flink* and *blink* values seen in the preceding lspd.pl output are the values that point to the next and previous processes, respectively. As displayed in the output of lspd.pl, these pointers are to addresses in memory, not physical addresses or offsets within the dump file.

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:\IR\bin\;E:\IR\WFT;E:\IR\
                          windbg\;E:\IR\Foundstone\;E:\IR\Cygwin;E:\IR\
                          somarsoft\;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, etc.) it can find in memory. For example, lspd.pl found that dd.exe had the following file handle open:

```
Type : File
Name = \intrusion2005\audit.log
```

As you 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, you can assume that the E:\ drive listed in *ImagePathName* is a CD-ROM drive, because Helix can be run from a CD. You 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). You can also use similar information to find out a little bit more about the F:\ drive. I will cover this information in Chapter 4.

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 whether a valid executable image is located at that address. This check is very simple; all it does is read the first two bytes starting at the translated

address to see whether they're *MZ*. These two bytes are not a definitive check, but PE files (files with .exe, .dll, .ocs, .sys, and similar 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.

TIP

If you dumped the contents of physical memory from a Windows 2000 or XP system using winen.exe and you have a licensed EnCase dongle, you can parse process information from a memory dump using EnScripts written by TK_Lane and available through the "EDD and Forensics" blog (http://eddandforensics.blogspot.com/2008/04/windows-memory-analysis.html).

Volatility Framework

Aaron Walters provides some valuable information about the Volatility Framework in his OMFW presentation, available from https://www.volatilesystems.com/volatility/omfw/Walters_OMFW_2008.pdf.

The readme.txt file that is part of the Volatility distribution (Version 1.3 beta at the time of this writing) provides a great deal of information about how to use Volatility and what types of commands and capabilities are available, as well as examples of how to launch the various commands. Aaron designed Volatility to use some of the commands that are commonly used in incident response activities; for example, to get a list of running processes from a memory dump, Volatility uses *pslist*. Before using Volatility, be sure to read through the readme.txt file to see what type of information can be retrieved from a Windows XP SP2 or SP3 memory dump.

To illustrate what type of information is available from a raw, dd-style memory dump, let's take a look at an example; in this case a 512MB memory dump from a Windows XP SP2 laptop. We can start by getting some basic information about the memory dump using the *ident* command:

```
D:\Volatility>python volatility ident -f d:\hacking\xp-laptop1.img
              Image Name    : d:\hacking\xp-laptop1.img
              Image Type    : Service Pack 2
                VM Type     : nopae
                    DTB     : 0x39000
                  Datetime  : Sat Jun 25 12:58:47 2005
```

This can be very useful information in documenting our analysis of the memory dump, as in some instances, we may not have access to the *ident* information as part of our

documentation. Using the *pslist* command, we can retrieve the active process list from the memory dump in a format similar to what we're used to seeing when running pslist.exe on a live system:

```
D:\Volatility>python volatility pslist -f d:\hacking\xp-laptop1.img
Name              Pid      PPid    Thds     Hnds    Time
System              4         0      61     1140     Thu  Jan 01 00:00:00   1970
smss.exe          448         4       3       21     Sat  Jun 25 16:47:28   2005
csrss.exe         504       448      12      596     Sat  Jun 25 16:47:30   2005
winlogon.exe      528       448      21      508     Sat  Jun 25 16:47:31   2005
services.exe      580       528      18      401     Sat  Jun 25 16:47:31   2005
lsass.exe         592       528      21      374     Sat  Jun 25 16:47:31   2005
svchost.exe       740       580      17      198     Sat  Jun 25 16:47:32   2005
svchost.exe       800       580      10      302     Sat  Jun 25 16:47:33   2005
svchost.exe       840       580      83     1589     Sat  Jun 25 16:47:33   2005
Smc.exe           876       580      22      423     Sat  Jun 25 16:47:33   2005
svchost.exe       984       580       6       90     Sat  Jun 25 16:47:35   2005
svchost.exe      1024       580      15      207     Sat  Jun 25 16:47:35   2005
spoolsv.exe      1224       580      12      136     Sat  Jun 25 16:47:39   2005
ssonsvr.exe      1632      1580       1       24     Sat  Jun 25 16:47:46   2005
explorer.exe     1812      1764      22      553     Sat  Jun 25 16:47:47   2005
Directcd.exe     1936      1812       4       40     Sat  Jun 25 16:47:48   2005
TaskSwitch.exe   1952      1812       1       21     Sat  Jun 25 16:47:48   2005
Fast.exe         1960      1812       1       22     Sat  Jun 25 16:47:48   2005
VPTray.exe       1980      1812       2       89     Sat  Jun 25 16:47:49   2005
atiptaxx.exe     2040      1812       1       51     Sat  Jun 25 16:47:49   2005
jusched.exe       188      1812       1       22     Sat  Jun 25 16:47:49   2005
EM_EXEC.exe       224       112       2       74     Sat  Jun 25 16:47:50   2005
ati2evxx.exe      432       580       4       38     Sat  Jun 25 16:47:55   2005
Crypserv.exe      688       580       3       34     Sat  Jun 25 16:47:55   2005
DefWatch.exe      864       580       3       27     Sat  Jun 25 16:47:55   2005
msdtc.exe        1076       580      14      166     Sat  Jun 25 16:47:55   2005
Rtvscan.exe      1304       580      37      300     Sat  Jun 25 16:47:58   2005
tcpsvcs.exe      1400       580       2       94     Sat  Jun 25 16:47:58   2005
snmp.exe         1424       580       5      192     Sat  Jun 25 16:47:58   2005
svchost.exe      1484       580       6      119     Sat  Jun 25 16:47:59   2005
wdfmgr.exe       1548       580       4       65     Sat  Jun 25 16:47:59   2005
Fast.exe         1700       580       2       32     Sat  Jun 25 16:48:01   2005
mqsvc.exe        1948       580      23      205     Sat  Jun 25 16:48:02   2005
mqtgsvc.exe      2536       580       9      119     Sat  Jun 25 16:48:05   2005
alg.exe          2868       580       6      108     Sat  Jun 25 16:48:11   2005
```

```
wuauclt.exe         2424     840    4    160    Sat  Jun 25 16:49:21  2005
firefox.exe         2160    1812    6    182    Sat  Jun 25 16:49:22  2005
PluckSvr.exe         944     740    9    227    Sat  Jun 25 16:51:00  2005
iexplore.exe        2392    1812    9    365    Sat  Jun 25 16:51:02  2005
PluckTray.exe       2740     944    3    105    Sat  Jun 25 16:51:10  2005
PluckUpdater.ex     3076    1812    0     -1    Sat  Jun 25 16:51:15  2005
PluckUpdater.ex     1916     944    0     -1    Sat  Jun 25 16:51:40  2005
PluckTray.exe       3256    1812    0     -1    Sat  Jun 25 16:54:28  2005
cmd.exe             2624    1812    1     29    Sat  Jun 25 16:57:36  2005
wmiprvse.exe        4080     740    7      0    Sat  Jun 25 16:57:53  2005
PluckTray.exe       3100    1812    0     -1    Sat  Jun 25 16:57:59  2005
dd.exe              4012    2624    1     22    Sat  Jun 25 16:58:46  2005
```

We can run similar commands to retrieve information about all process objects in the memory dump, including exited processes, using the *psscan* or *psscan2* command. Commands to retrieve information about all objects (network connections, processes, etc.) are slower, as they use a linear scanning method to run completely through the memory dump file, examining all possible objects, rather than using specific offsets provided by the operating system (see the discussion about LiveKD earlier in this chapter).

One of the more useful things most analysts look to when responding to an intrusion or compromise is network connections. You can retrieve a list of active network connections (similar to using the *netstat –ano* command) from a memory dump using the *connections* command, as follows:

```
D:\Volatility>python volatility connections -f d:\hacking\xp-laptop1.img
Local Address          Remote Address         Pid
127.0.0.1:1056         127.0.0.1:1055         2160
127.0.0.1:1055         127.0.0.1:1056         2160
192.168.2.7:1077       64.62.243.144:80       2392
192.168.2.7:1082       205.161.7.134:80       2392
192.168.2.7:1066       199.239.137.200:80     2392
```

Taking this a step further, you can scan the entire memory dump file for indications of network connection objects, specifically looking for network connections that may have been closed at the time the memory dump was acquired:

```
D:\Volatility>python volatility connscan2 -f d:\hacking\xp-laptop1.img
Local Address          Remote Address         Pid
------------------     -------------------    ------
192.168.2.7:1115       207.126.123.29:80      1916
3.0.48.2:17985         66.179.81.245:20084    4287933200
192.168.2.7:1164       66.179.81.247:80       944
192.168.2.7:1082       205.161.7.134:80       2392
```

```
192.168.2.7:1086        199.239.137.200:80      1916
192.168.2.7:1162        170.224.8.51:80         1916
127.0.0.1:1055          127.0.0.1:1056          2160
192.168.2.7:1116        66.161.12.81:80         1916
192.168.2.7:1161        66.135.211.87:443       1916
192.168.2.7:1091        209.73.26.183:80        1916
192.168.2.7:1151        66.150.96.111:80        1916
192.168.2.7:1077        64.62.243.144:80        2392
192.168.2.7:1066        199.239.137.200:80      2392
192.168.2.7:1157        66.151.149.10:80        1916
192.168.2.7:1091        209.73.26.183:80        1916
192.168.2.7:1115        207.126.123.29:80       1916
192.168.2.7:1155        66.35.250.150:80        1916
127.0.0.1:1056          127.0.0.1:1055          2160
192.168.2.7:1115        207.126.123.29:80       1916
192.168.2.7:1155        66.35.250.150:80        1916
```

Volatility is a powerful open source framework, allowing others to extend its capabilities by developing additional modules (knowledge of Python programming is a significant requirement). Brendan Dolan-Gavitt (a.k.a. Moyix) created a Volatility module that looks for Windows messages (http://moyix.blogspot.com/2008/09/window-messages-as-forensic-resource.html), which are various events generated by Windows GUI applications and handled by the message queue. As Brendan points out, an application may be poorly written and may not handle its own messages very well; if this is the case, you may be able to find remnants of those messages still visible in the memory dump. This information may be useful during a forensic examination.

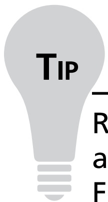
TIP

Brendan also produced several Volatility plug-ins for accessing Registry data found in Windows memory dumps (his blog post is at http://moyix.blogspot.com/2009/01/memory-registry-tools.html, and updates to the code are at http://moyix.blogspot.com/2009/01/registry-code-updates.html). In his own blog (http://forensiczone.blogspot.com/2009/01/using-volatility-1.html), Richard McQuown demonstrated using these modules to extract passwords from a Security Account Manager (SAM) hive file located in memory so that he could crack those passwords using his tool of choice.

To use Volatility and Brendan's modules to extract passwords from hive files located in memory, you have to install the PyCrypto modules (available as prebuilt Windows binaries from www.voidspace.org.uk/python/modules.shtml#pycrypto).

In addition, Jesse Kornblum produced two modules: suspicious, which looks for suspicious entries in process command lines, and cryptoscan, which looks for TrueCrypt passphrases. This last module can be extremely beneficial to an analyst, as TrueCrypt (www.truecrypt.org/) is a powerful, albeit free, application that can be used to encrypt volumes and disks.

Volatility works with much more than just simply raw memory dumps. Thanks to the efforts of Matthieu Suiche (www.msuiche.net/), Volatility includes the capability to parse hibernation files, as well. This started out as the Sandman Project, and later became an integral part of the Volatility Framework. In December 2008, Matthieu released a stand-alone, closed source (alpha) version of the hibernation framework shell, called hibrshell (www.msuiche.net/hibrshell/). This version of hibrshell reportedly works with hibernation files from Windows XP, 2003, Vista, and 2008 systems.



Regardless of the framework used to analyze it, the hibernation file provides a responder or analyst with several options that were not previously available. First, the hibernation file can be used as historical data, providing information about the system's live, running state at a previous point in time. This can be extremely valuable in malware analysis, as well as to assist in determining a timeline for an intrusion, particularly if the analyst also has a current memory dump to analyze. In circumstances where the previously mentioned tools (e.g., mdd.exe, etc.) cannot be used to dump the contents of physical memory from a system, the responder may be able to force the system to hibernate to create a memory dump that can then be analyzed.

Volatility can also parse crash dump files, as well as convert a raw, dd-style memory dump to crash dump format so that the analyst can use Microsoft's debugger tools.

By now it should be clear that the Volatility Framework provides some extremely powerful capabilities, and just how much information the analyst can retrieve from a memory dump. To help correlate some of the data that can be retrieved using Volatility, Jamie Levy wrote a Perl script called vol2html.pl (http://gleeda.blogspot.com/2008/11/vol2html-perl-script.html). The script takes the output of the Volatility *pslist*, *files*, and *dlllist* commands and correlates them into an HTML report, an example of which you can see at http://venus.cs.qc.edu/~jlevy/code/report/index.html. Similar to the familiar listdlls.exe available from Microsoft (Sysinternals), the Volatility *dlllist* command includes the process command line as part of its output; this command line also appears in the HTML output of vol2html.pl.

Examples of Windows XP memory dump files are available as part of the DFRWS 2008 Forensic Rodeo (www.dfrws.org/2008/rodeo.shtml), as well as from the National Institute of Standards and Technology (www.cfreds.nist.gov/mem/Basic_Memory_Images.html).

Michael Hale Ligh provides two blog posts that describe how he has used the Volatility Framework to great effect, particularly with respect to malware analysis; see "Recovering CoreFlood Binaries with Volatility" (http://mnin.blogspot.com/2008/11/recovering-coreflood-binaries-with.html), and "Locating Hidden Clampi DLLs (VAD-style)" (http://mnin.blogspot.com/2008/11/locating-hidden-clampi-dlls-vad-style.html). Both blog posts provide excellent examples of how the Volatility Framework can maximize an analyst's capabilities.

Memoryze

Mandiant's Memoryze tool provides the analyst with the ability to parse and analyze memory dumps from several versions of Windows. To install Memoryze, download the MSI file from the Mandiant Web site (mentioned previously in this chapter) and install it. I chose to install it in the D:\Mandiant directory. Then, to install Audit Viewer, download the zipped archive, and be sure that you've downloaded the dependencies (i.e., Python 2.5 or 2.6, wxPython GUI extensions) as described at the Mandiant Web site (if you've already installed and tried Volatility, you already have Python installed). I chose to unzip the Audit Viewer files into the directory D:\Mandiant\AV.

To demonstrate the use of Memoryze and Audit Viewer, we'll start by selecting a memory dump from a Windows 2003 system: boomer-win2003.img. The first thing we'll need to do to analyze this memory dump is to run Memoryze against it to extract various data:

```
D:\mandiant>process.bat -input d:\hacking\boomer-win2003.img -ports true -handles
true -sections true
```

The preceding command tells Memoryze to parse the process information from the memory dump, and get ports, handles, and memory sections (I've purposely opted not to get the strings from each process) for the processes in the active process list. The full range of usage options for process.bat includes the following:

```
Usage: process.bat
-input      name of snapshot. Exclude for live memory.
-pid        PID of the process to inspect. Default: 4294967295 = All
-process    optional name of the process to inspect. Default: Excluded
-handles    true|false inspect all the process handles. Default: false
-sections   true|false inspect all process memory ranges. Default: false
-ports      true|false inspect all the ports of a process. Default: false
-strings    true|false inspect all the strings of a process. Default: false
-output     directory to write the results. Default .\Audits
```

By default, the preceding command line places its resultant XML files in the .\Audit directory. In this case, the full path is D:\mandiant\Audits\WINTERMUTE\20090103134554.

Mandiant's Audit Viewer is a GUI tool that provides the analyst with a graphical interface into the XML files created by using Memoryze to parse memory dumps. To launch Audit Viewer, double-click the **AuditViewer.py** file in the directory where you unzipped the

archive downloaded from the Mandiant site. As you've installed the wxPython modules, you will see the Audit Viewer GUI open, at which point you will need to configure the tool by changing the **Memoryze Install Directory** (if necessary), selecting the **Running on image** checkbox, and providing the **Path to image file**, as Figure 3.11 illustrates.

Figure 3.11 Audit Viewer User Interface Showing Configuration Changes

Once you've made the necessary changes, click the **Open Audit** button at the top of the Audit Viewer user interface, and navigate to the directory where the XML audit files were created. Once the directory has been selected, Audit Viewer will parse the available files and populate the Processes tree in the user interface, as shown in Figure 3.12.

Figure 3.12 Audit Viewer User Interface Showing Processes Tree

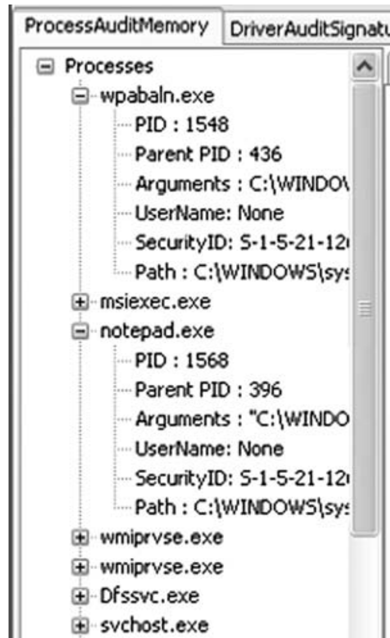
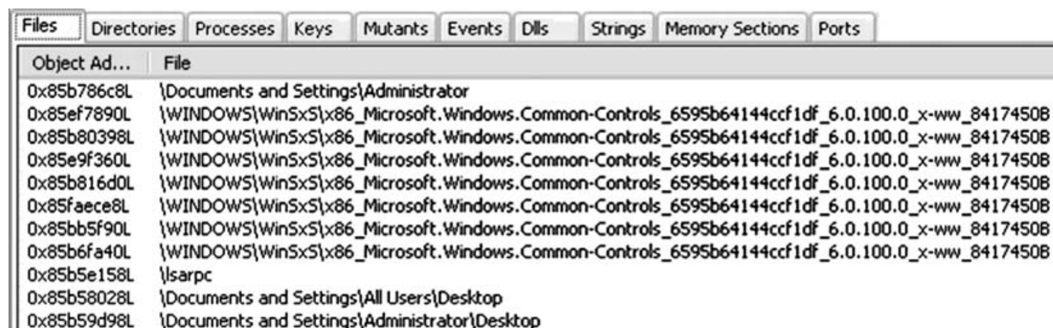


Figure 3.12 also illustrates two processes expanded to show the PID, PPID, arguments (or command line), as well as other information about each process. To dig deeper into each process, double-click the process name in the Processes tree, and then view the contents of the various tabs (Files, Directories, etc.) visible in the Audit Viewer user interface, as Figure 3.13 illustrates.

Figure 3.13 Audit Viewer User Interface Showing Process Detail Tabs



Memoryze and Audit Viewer provide a number of additional options to the analyst. For example, based on your findings in Audit Viewer, you may decide that you'd like to acquire an image of a process executable from the image file. To do so, use the processdd.bat batch file as follows:

```
D:\mandiant\processdd.bat -pid PID -input d:\hacking\boomer-win2003.img
```

You can also use other batch files provided with Memoryze to perform rootkit and hook detection, as well as search for drivers (www.mandiant.com/software/usememoryze.htm). The Mandiant M-unition blog (http://blog.mandiant.com/) provides additional examples of how to use Memoryze and Audit Viewer, such as how to integrate the two tools into Guidance Software's EnCase forensic analysis application.

HBGary Responder

HBGary's Responder product is a commercial GUI memory dump analysis tool that is described on the company Web site (www.hbgary.com) as a "live memory and runtime analysis software suite used to detect, diagnose, and respond to today's advanced computer threats". As with other analysis tools, the Responder products (Professional and Field editions) allow a responder to parse and analyze a memory dump without having to utilize the potentially compromised or infected system's API. Although Responder was written with malware analysis in mind, it is also a fast and capable tool that provides a great deal of functionality with respect to incident response, and is very easy for responders to use to get the information they need quickly.

NOTE

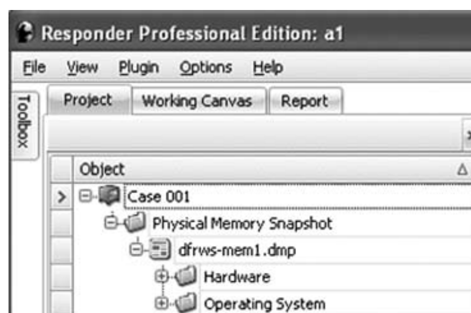
An evaluation copy of Responder Professional Edition 1.3.0.377 was used in the examples listed in this section of the chapter. However, the functionality presented and observed in this section is inherent to both the Professional and Field Edition products. We will not conduct a comprehensive review of the Responder Professional product (the Professional product includes binary disassembly, control flow graphing, and reverse engineering capabilities), as doing so is beyond the scope of this book and we are focusing on aspects of the product that most directly pertain to the memory dump analysis pursuant to incident response activities.

All you need to do to begin analyzing a memory dump with Responder Pro is to create a case and then import a physical memory snapshot by selecting **File** from the menu bar, then **Import**, and then **Import a Physical Memory Snapshot**. For this example, we'll use the first Windows 2000 memory dump from the DFRWS 2005 Memory Challenge; however, like Memoryze, Responder works with memory dumps from Windows 2000, all the way up through the latest versions of Windows.

When importing a memory dump "snapshot" file into a Responder case, an option is available to "extract and analyze all suspicious binaries". The rules that the Responder product uses to determine what constitutes "suspicious" are in a text-based file that you can open and review, and to which you can even add or remove comments, reducing false positives; you also can add entries to the file based on experience, thereby increasing the product's effectiveness.

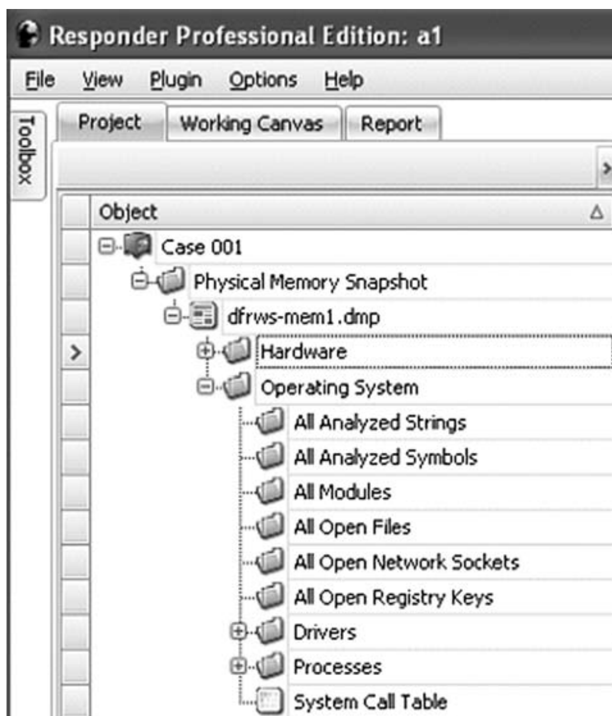
Once the memory dump file has been imported and parsed, Responder will show in the left-hand pane the memory dump file with two folders, Hardware and Operating System, as Figure 3.14 illustrates.

Figure 3.14 Memory Dump File Imported into a Responder Project



Expanding the folder beneath the Hardware folder will display the Interrupt Table. Expanding the Operating System folder will display a great deal of additional information, including Processes and All Open Network Sockets (in part, what responders may be most interested in), as shown in Figure 3.15.

Figure 3.15 Expanded Operating System Folder in Responder User Interface



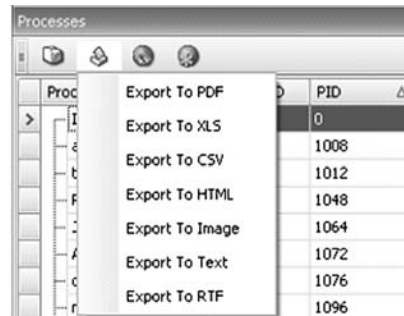
You can then expand the Processes folder to see all of the active processes extracted from the memory dump, or double-click the Processes folder to have the processes and detailed information about each process visible in the right-hand pane of the Responder user interface, as shown in Figure 3.16.

Figure 3.16 Excerpt of Processes Listed with Details in Responder User Interface

nc.exe	592	1096	"c:\winnt\system32\nc.exe" -L -p 3000 -t -e cmd.exe
cmd2k.exe	324	1112	"E:\Shells\cmd2k.exe" /D /T:80 /F:ON /K cmdenv.bat
cmd2k.exe	324	1132	"E:\Shells\cmd2k.exe" /D /T:80 /F:ON /K cmdenv.bat
smss.exe	8	156	\SystemRoot\System32\smss.exe
winlogon.exe	156	176	winlogon.exe
csrss.exe	156	180	C:\WINNT\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,3072,51...
services.exe	176	228	C:\WINNT\system32\services.exe
lsass.exe	176	240	
dd.exe	1112	284	..\Acquisition\FAU\dd.exe if=\\.\PhysicalMemory of=F:\intrusion2005\physicalmemory.dd ...
helix.exe	820	324	E:\helix.exe

The Responder user interface provides a good deal of information that is immediately useful to you. You can also adjust the columns by selecting them and dragging them to a new location within the same view pane. Also, you can export information (this functionality is not supported in the evaluation version) from the view pane to various formats by clicking the appropriate icon at the top of the view pane, as Figure 3.17 illustrates.

Figure 3.17 Selecting to Export Data in Responder User Interface



Double-clicking the **All Open Network Sockets** folder will open the Network tab in the right-hand view pane, similar to the output of netstat.exe, as shown in Figure 3.18.

Figure 3.18 Open Network Sockets from Memory Dump in Responder User Interface

Network

Source	Destination	Type	Process
0.0.0.0:1033	192.168.0.5:4321	TCP	lsass.exe (240)
0.0.0.0:1055	192.168.0.5:4321	TCP	lsass.exe (240)
0.0.0.0:1025	0.0.0.0:0	TCP	MSTask.exe (552)
0.0.0.0:3000	0.0.0.0:0	TCP	nc.exe (1096)
0.0.0.0:1026	0.0.0.0:0	UDP	services.exe (228)
0.0.0.0:135	0.0.0.0:0	UDP	svchost.exe (408)
0.0.0.0:135	0.0.0.0:0	TCP	svchost.exe (408)
0.0.0.0:641	0.0.0.0:0	TCP	tgcmd.exe (1012)
0.0.0.0:653	0.0.0.0:0	TCP	tgcmd.exe (1012)
0.0.0.0:44444	0.0.0.0:0	TCP	UMGR32.EXE (668)

You can also view the open file handles for all processes by double-clicking the **All Open Files** folder, as Figure 3.19 illustrates.

Figure 3.19 Partial Listing of Open Files from Responder User Interface

audit.log	\intrusion2005\audit.log	dd.exe (284)
physicalmemory.dd	\intrusion2005\physicalmemory.dd	dd.exe (284)
shells	\shells	dd.exe (284)
hxdef-rk100sb4d1ba5d	\hxdef-rk100sb4d1ba5d	dfrws2005.exe (592)
hxdef-rk100sb4d1ba5d	\hxdef-rk100sb4d1ba5d	dfrws2005.exe (592)
ntcontrolpipe9	\net\ntcontrolpipe9	dfrws2005.exe (592)
svcctl	\svcctl	dfrws2005.exe (592)

You can view open file handles for a specific process by expanding the tree for each process and selecting **Open Files**. You can do the same for open network sockets and open Registry keys.

In addition to being able to quickly view all of this information, both on a memory dump-wide format as well as on a per-process format, you can parse executable images (.exe and .dll files) for strings, as well as search for specific items within the memory dump using substrings, regular expressions, or exact matches. Having the ability to sort items visible in columns also allows you to identify suspicious processes or modules (i.e., DLLs) much more quickly.

Parsing Process Memory

We discussed the need for context for evidence earlier in this chapter, and you can achieve this, 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 you're parsing through a file that is about half a megabyte in size, there isn't a great deal of context to the information you 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 you to do this automatically when working with Windows 2000 memory dumps. 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, use the following command line:

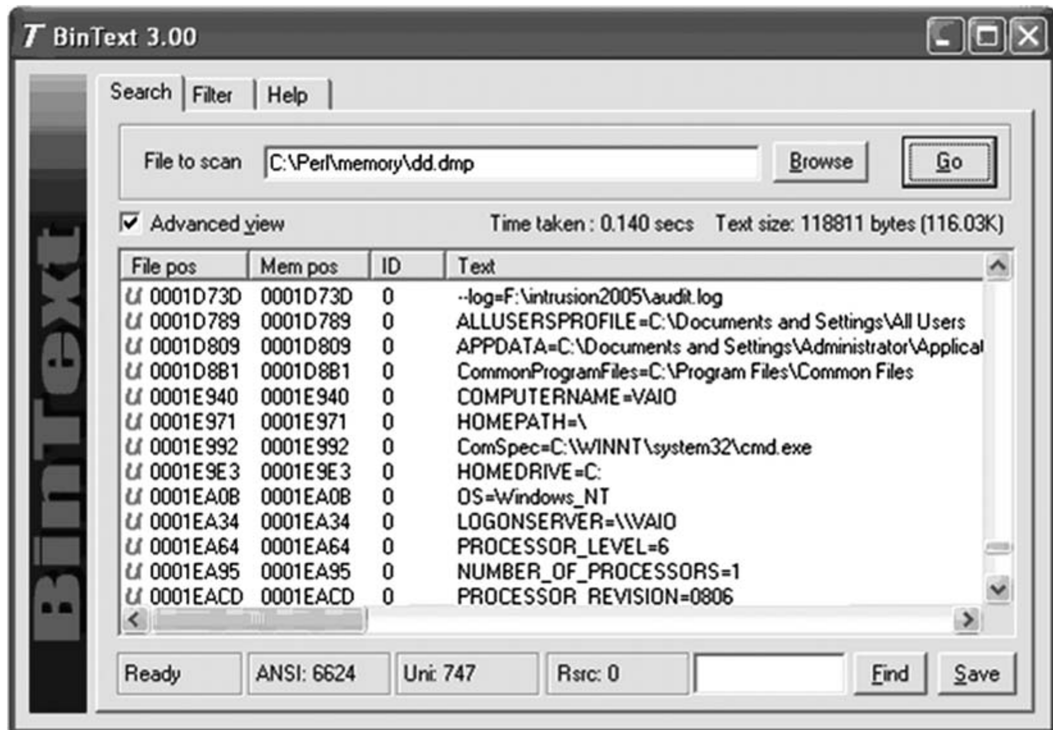
```
c:\perl\memory>lspm.pl d:\dumps\dfrws1-mem.dmp 0x0414dd60
```

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 you 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 (illustrated in Figure 3.20) 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 3.20 Contents of Process Memory in BinText



In Figure 3.20, you 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 your understanding of the case; an important aspect of this capability is that now you can correlate what you find to a specific process.

Volatility incorporates this same functionality in the *memdmp* command. As mentioned previously in the chapter, you can use the *volatility memdmp* command to dump the addressable memory for a process from a Windows XP memory dump, as follows:

```
D:\Volatility>python volatility memdmp -f d:\hacking\xp-laptop1.img -p 4012
```

TIP

You can use Volatility to collect process memory for processes that are hidden by rootkits, even those hidden using direct kernel object manipulation (DKOM) techniques (see Chapter 7). Specifically, DKOM techniques "unlink" the EProcess block for the hidden process from the doubly linked active process list that the operating system "sees." However, using Volatility to examine a Windows XP raw memory dump or hibernation file, you can search for processes that are not part of that doubly linked list (discussed later in the chapter), and then use the *memdmp* command to retrieve the memory used by the process from the memory dump file.

Extracting the Process Image

As you saw earlier in this chapter, when a process is launched the executable file is read into memory. One of the pieces of information that you can get from the process details (via lspd.pl) is the offset within a Windows 2000 memory dump file to the Image Base Address. As you saw, lspd.pl will do a quick check to see whether an executable image can be found at that location. One of the things you can do to develop this information further is to parse the PE file header (the contents of which we will cover in detail in Chapter 6) and see whether you can extract the entire contents of the executable image from the Windows 2000 memory dump file. Lspi.pl lets you 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 you 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\dfrws1-mem.dmp 0x0414dd60
```

The output of the command appears as follows:

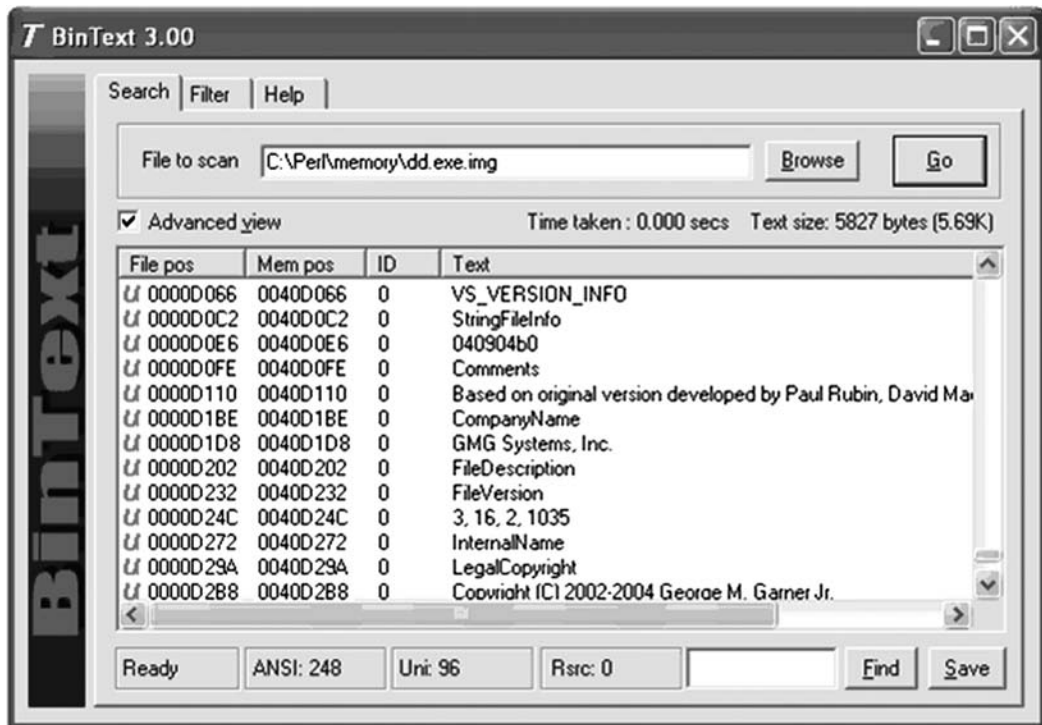
```
Process Name         : dd.exe
PID                  : 284
DTB                  : 0x01d9e000
PEB                  : 0x7ffdf000 (0x02c2d000)
ImgBaseAddr          : 0x00400000 (0x00fee000)
e_lfanew = 0xe8
NT Header = 0x4550
Reading the Image File Header
Sections = 4
Opt Header Size = 0x000000e0 (224 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 lspi.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, we will discuss this information in detail in Chapter 6. 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 you with a road map for reassembling the executable image because it lets you 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 (e.g., 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 you extract from the memory dump will not be exactly the same as the original executable file. This is because some of the file's sections are writeable, and those sections will change as the process is executing. As the process executes, various elements of the executable code (addresses, variables, etc.) will change according to the environment and the stage of execution. However, there are a couple of ways you 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 you 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, you can look for the Unicode string *VS_VERSION_INFO* and see whether any identifying information is available in the executable image file. Figure 3.21 illustrates some of the strings found in the dd.exe.img file using BinText.

Figure 3.21 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 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 ssdeep (http://ssdeep.sourceforge.net) 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 paper (http://dfrws.org/2006/proceedings/12-Kornblum.pdf) 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, and computed by ssdeep. Because 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)
```

We can also do this in one command line using either the *–d* or the *–p* switch:

```
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 awhile before hash sets either are generated using ssdeep.exe or incorporate hashes calculated using ssdeep.exe.

We can use the Volatility Framework to attempt to extract the executable image from a Windows XP memory dump file using the *procdump* command. The *procdump* command syntax (from the Volatility readme.txt file) appears as follows:

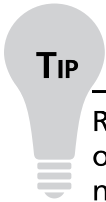
```
procdump
--------
For each process in the given image, extract an executable sample.
If -t and -b are not specified, Volatility will attempt to infer
reasonable values.
Options:
  -f       <Image>    Image file to load
  -b       <base>     Hexadecimal physical offset of valid Directory Table Base
  -t       <type>     Image type (pae, nopae, auto)
  -o       <offset>   Hexadecimal physical offset of EPROCESS object
  -p       <pid>      Pid of process
  -m       <mode>     Strategy to use when extracting executable sample.
                      Use "disk" to save using disk-based section sizes or "mem"
                      for memory based sections (default": "mem").
```

Continuing with the Volatility example from earlier in this chapter, we can extract the executable image file for the dd.exe process (PID 4012 in the xp-laptop1.img memory dump file) using this command:

```
D:\Volatility>python volatility procdump -f d:\hacking\xp-laptop1.img -p 4012
***************************************************************************
Dumping dd.exe, pid: 4012 output: executable.4012.exe
D:\Volatility>dir exe*.exe
```

```
Volume in drive D is Data
Volume Serial Number is 8049-F885
Directory of D:\Volatility
01/01/2009 12:45 PM    57,344 executable.4012.exe
1 File(s)              57,344 bytes
```

Although not as verbose as the lspi.pl Perl script for Windows 2000 memory dumps, the *volatility procdump* command extracts the executable image file for the process. This can be extremely useful during malware analysis, as a good deal of the current malware is obfuscated (encrypted, compressed, or both) while on disk, making static analysis (see Chapter 6) difficult. Also, some malware may be memory-resident only, never actually being written to the hard drive; being able to extract an executable image file from a memory dump may be the only way to get a copy of the file for analysis.



Responders may often be confronted with systems that employ some sort of encryption of either a specific volume or the entire disk. I've acquired a number of these systems, and when I have to conduct that acquisition with the intention of performing analysis (as opposed to simply acquiring an image), I've opted to perform a live acquisition of the hard drive. In May 2007, Brian Kaplan wrote a thesis paper titled "RAM is Key: Extracting Disk Encryption Keys from Volatile Memory." Along with that paper, Brian also released a proof of concept tool for extracting Pretty Good Privacy (PGP) Whole Disk Encryption (WDE) keys from a memory dump. The paper and proof of concept tool are available from www.andrew.cmu.edu/user/bfkaplan/#KeyExtraction.

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 tools such as lspm.pl that rely solely on the contents of the RAM dump will provide an incomplete memory dump, because memory pages that have been swapped out to the page file (pagefile.sys on Windows systems) will not be incorporated in the resultant memory dump. To overcome this deficiency, in spring 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.

In early 2007, Jesse Kornblum's "Buffalo" paper was published in the *Journal of Digital Investigation* (the full title of the paper is "Using Every Part of the Buffalo in Windows Memory Analysis"), and the publisher of the *Journal* allowed Jesse to post a copy of this paper on his Web site.

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.

Pool Allocations

When the Windows memory manager allocates memory, it generally does so in 4 KB (4096 bytes) pages. However, allocating an entire 4 KB 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 (www.microsoft.com/whdc/driver/tips/PoolMem.mspx) file provided with the Microsoft Debugging Tools, and Microsoft provides a Knowledge Base article that describes how to locate pool tags/headers used by third-party applications (http://support.microsoft.com/default.aspx?scid=kb;en-us;298102). Andreas used a similar method to determine the format of memory pools used to preserve information about network connections in Windows 2000 memory dumps (http://computer.forensikblog.de/en/2006/07/finding_network_socket_activity_in_pools.html); 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.

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 (tlist.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).

Solutions Fast Track

Collecting Process Memory

- ☑ A responder may 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 only for 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) might not be visible, and the investigator will not be able to provide the process identifier to the tools he 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 also the memory located in the page file.
- ☑ Once process memory has been collected, additional information about the process, such as open handles and loaded modules, can then be collected.

Dumping Physical Memory

- ☑ Several methodologies are available for dumping the contents of physical memory. The responder should be aware of the available options as well as their pros and cons so that 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 because the system is still live and processing information while the memory dump is being generated.

☑ When dumping the contents of physical memory, both the responder and the analyst 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 reveals 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.

Frequently Asked Questions

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, but also can "see" things that might not be "visible" via traditional means (e.g., things hidden by a rootkit; see Chapter 7 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 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 his 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 became clear to me that useful information could be 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, the command line used to launch the process, or open handles. Once all the information is collected, the next step could be to save 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.