Submitted for the Degree of M.Sc. in Forensic Informatics 2006

"Acquisition and Analysis of Windows Memory"

200557997

Nicholas Paul Maclean

"Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during the course. It has not been submitted for assessment in any other context. "

"I agree to this material being made available in whole or in part to benefit the education of future students."

Signed:_____ 04/09/2006

This page intentionally blank

Part 2:

Acquisition and Analysis of Windows Memory

Abstract

The examination of volatile memory is a relatively new but important area in computer forensics: criminals are becoming more forensically aware and are now able to commit crimes without accessing the hard disk of the target computer. This means that traditional incident response practice of pulling the plug will destroy the only evidence of the crime. While some techniques are available for acquiring the contents of main memory, few exist which can interpret these data in a meaningful way. One reason for this is the complex abstraction used in the virtual memory systems of modern operating systems. This means that data belonging to one process can be distributed in a seemingly arbitrary way across the entire range of the physical memory space and also on the hard disk, making it very difficult to recover any useful information. This report will focus on how these disparate sources of information can be combined to give a single, contiguous address space for each process through analysis of: the kernel structures which hold the required data; the virtual memory system including how memory is mapped and translated between the physical memory and the virtual address space; and tools available for acquisition and analysis of memory. From this information, a proof-of-concept tool is developed to reconstruct the virtual address space of a process by combining a physical memory dump with the page file on the hard disk.

Acknowledgements

I would like to thank Ian Ferguson for arranging my placement with QinetiQ and for his help throughout the project; Phil Turner and all the staff at QinetiQ for making me feel welcome and for their invaluable advice; and Caroline for her endless encouragement.

This page intentionally blank

Contents

This page intentionally blank

I. Introduction

Motivation and Intent

While the methods for acquiring and examining data from non-volatile storage media such as hard disks and CD-ROMs are well documented [5,44,46], equivalent guidelines for handling volatile storage such as main memory are still somewhat lacking. Some guidelines[5] entirely disregard the collection of volatile media, instead suggesting that any device be switched off immediately to prevent loss of data on the hard disk. Others [44,46] suggest that data be collected in order of volatility: starting with the most volatile (screen contents, RAM) and ending with the least volatile (hard disks, CD-ROMs). However, these guidelines have few details showing how this collection should be performed. Furthermore, once an image of memory has been acquired there are currently no official guidelines relating to how that image should be analysed. Many investigators will run a search for text, or use a 'file carver' to look for files such as images; however, although these searches may produce some interesting pieces of text or images they will not show the context in which the recovered data was stored. For example, a strings search may recover references to illegal internet sites; but it could not say whether they were entries in the user's browser history, or in their parental control software. Although development of a fully fledged memory analysis system would be of use, there are a number currently under development as shown below. What does appear to be lacking is a tool which can amalgamate the various sources which make up a process' virtual address space. The focus of this report will therefore be to investigate what data is stored and where it can be recovered from; and examine in detail the virtual memory system in detail so that this tool can be developed.

Overview

This section begins with a description of the methods and tools used in the research phase of this report. It then continues with a short overview of the current state of memory forensics, including a summary of past and present research in the area. Section II introduces some of the structures resident in Windows memory, and how they can be analysed to find useful information. In Section III, the Windows Virtual Memory system is explained, showing how it is possible to recover information from a Windows memory image. This includes an examination of virtual and physical addresses and the paging system, and how to navigate the virtual address space. Section IV discusses the process of memory acquisition in terms of what tools can be used to acquire memory, the issues and problems arising during imaging, and possible solutions and alternative methods which may be available. In Section V current methods are examined for the analysis of what is produced during the acquisition stage. Section VI introduces the '*vtop*' proof of concept tool which reconstructs a process' virtual address space given a memory image and a page-file. Here are presented details of how the tool operates, and suggestions for how the concept could be included in further work. Finally, Section VII gives a summary of the findings of this report, and recommendations for further work in the area of memory forensics.

Methods And Tools

The main body of research required for this project was in gathering the details of the structures resident in RAM and in the memory images. This was done using a combination of: knowledge of the architecture of the system gained from [13]; debugging of crash dumps using the '*Debugging Tools for Windows*'[41] suite of utilities; manual tracing of addresses and offsets found using the debugger with '*WinHex'* [55] and a small amount of guesswork.

Once sufficient knowledge of the architecture had been acquired, it was possible to start looking for the structures, their type definitions and their offsets in memory. To do this, a test system was used: a memory image was taken using '*dd*', and this was immediately followed by forcing a crash dump using the '*CrashOnCtrlScroll*' method shown in Section IV. This ensured that the memory dump generated by '*dd*' was as similar as possible to the crash dump in order to minimise confusing discrepancies. This crash dump file was opened in *WinDbg*[41] which was able to parse it. By downloading symbols from Microsoft, the kernel structures could be examined using the 'dt' command: e.g. 'dt _eprocess' showed the structure of the _EPROCESS structure including offsets of data fields relative to the start of the structure. Provided with the address of the start of one such structure, all other connected structures could be found by reading the address of their offset and looking up their type definition using the 'dt' command.

Whenever new information was gathered from the debugger, it was confirmed by manually recovering that information from the memory dump, using *WinHex*. Offsets into a structure were verified by moving to the physical address of the structure and reading the value at that address plus the offset. At this point it was noted that the x86 architecture stores data as little-endian, and so data values had to be read in reverse-byte-order (e.g. the address 0x81291830 would appear as '30 18 29 81' in *WinHex*).

For development of the '*vtop*' proof of concept, the Python development environment '*IDLE*' was used since it allowed code fragments to be executed and tested without reloading the entire module.

When testing the impact of the acquisition tools two process viewers were used: '*Process Viewer'* [24] and '*Process Explorer*' [48]. *Process Explorer* gave more detailed information but *Process Viewer* was able to show the memory usage of a running process with faster updates.

Related Work

It is often said that computer forensics started with the work of Weitze Venema and Dan Farmer who produced '*The Coroner's Toolkit*'[12]. Even in those early days, some thought was given to memory analysis, as can be seen by the presence of '*memdump*' and '*pcat*' which can be used view and capture the state of a running Unix machine.

More recently, the advancement in memory forensics has started to gather speed. George Garner's '*dd*' utility [19] has been available for the imaging of memory under both Windows and Unix for some time, even though the analysis of the images it produces has been crude. In an attempt to overcome problems relating to software-based acquisition, Brian Carrier has developed a PCI card[7] which can read the contents of memory to an external device without altering it; however this requires prior installation. In an attempt to achieve similar results, without the requirement for hardware installation, research has been carried out by Maximillian Dornseif et al [34,35] attempting to create a tool to image memory via Firewire, making use of Direct Memory Access to minimise impact on the target machine. Although the theory of this appears sound, concrete results have not been forthcoming.

At the 2005 Digital Forensics Research Workshop Memory Challenge, where challengers were asked to analyse a dump of physical memory taken with *dd* two notable results were published: one from a team led by George Garner using their '*kntlist*' tool[17,18]; the other by Chris Betz using his own '*mem_parser*'[8,9]. Both submissions managed to extract significant amounts of information about the state of the system at the time when the dump was taken, however both are still in development and not publicly available.

Mariusz Burdach has produced a number of papers and presentations relating to memory analysis. His first[33] describes the Linux Virtual Memory system and how it can be examined using a number of freely available tools. He has also presented this paper at the Black Hat Federal 2006 security conference[31] and developed the '*Windows Memory Forensic Toolkit*'[30] which is a tool putting much of his research into practice. It is, however still in early stages of development. Jorge Mario Urrea has since expanded on the work undertaken by Burdach in his Masters Thesis[28]. This paper is similar in intent to Burdach's but focusses on the newer 2.6 Linux kernel instead.

Tobias Klein has written two publicly available tools: '*PD*'[51] which dumps a process from a running system; and '*MMP*'[52] which allows the investigator to analyse that dump under a very usable GUI. In addition, he has written a paper [49] explaining in detail how to use and interpret the output of his tools. Petroni, Walters et al are currently working on a piece of software called '*FATKit*' [1], which is an attempt to create an extensible framework for the development of further memory analysis tools. This software is not yet in the public domain but some literature is available from their website. If this software delivers what it promises, it should go some way towards focussing research into memory forensic behind a common cause. Finally, elsewhere in the Digital Forensics community, a number of personal websites and blogs maintained by security professionals and investigators continue to provide useful information and research into memory forensics [2,3,21].

This page intentionally blank

II. Windows Structures

Each program running under Windows is assigned a 'process' which instantiates the state of that program in memory. For a process to run, all data which has changed since the program was started must be maintained, and it is within the structure of the process that it is held. A list of currently running processes can be seen by invoking the Windows Task Manager, or using a third party tool such as Sysinternals' *Process Explorer*. Although running programs such as these is not a good way to view processes in a forensic environment since they will alter the contents of RAM, it does give an idea of what can be recovered from RAM about the state of the computer and the processes running on it.

Windows stores information about each process in an _EPROCESS kernel structure, and it is from these that most information can be gathered. The _EPROCESS structures for all processes are stored in the address space of the System process, so finding this structure is the first step in memory analysis. The format of the _EPROCESS structure (and selected sub-structures) in Windows XP can be found in Appendix B. This shows the name, type and offset of each structure, so once the base address of the _EPROCESS block is known, the value can be read by adding that offset and reading the value. Of the 4GB virtual address space allocated to the process, only the 1st 2GB of the virtual address space are available for the process to write to, with the 2nd 2GB used as shared system memory (although this can be changed to 3GB for the process and 1GB for the system using a boot option). This means that by looking up all virtual addresses ranging from 0x00000000 to 0x80000000 (0-2GB) a complete dump of all memory writable by that process can be produced.

Unfortunately, finding the virtual address of the System _EPROCESS is one of the harder parts, because it is different each time Windows loads. The *mem_parser* and *KnTList* tools have successfully done it, probably by searching for characteristic strings and patterns in the structure of the EPROCESS: in XP the PDB of the System process is always at 0x39000, and the '*DirectoryTableBase'* field at +0x018 holds this value. Also, the '*ImageFileName'* field at +0x174-0x183 holds the value 'System' as a 16 byte value. Therefore, the System EPROCESS should be recoverable by searching for instances of '00039000' (00 90 03 00 in raw format) and 'System ' (53 79 73 74 65 6D 00 00 in raw) at 0x15C bytes apart. This is very unlikely to produce more than one result. From here the other visible EPROCESSes can be found by following the *ActiveProcessLinks* field as shown below.

Process List

The first thing that should be recovered from the System _EPROCESS is the location of the corresponding _EPROCESS block for each of the other processes running on the system. All processes are linked together in a doubly-linked-list, so the '*ActiveProcessLinks*' field contains the virtual address of the *ActiveProcessLinks* of the next _EPROCESS at +0x088 and the previous _EPROCESS at +0x08C. All addresses with the exception of the '*DirectoryTableBase*' are given as virtual addresses which need to be translated, and this is covered in Section III. The one disadvantage of using this technique to find running processes is that it is possible for a piece of malware to 'unlink' itself from the other _EPROCESSes, rendering it invisible to process viewers and some memory analysers. To do this, the malware process alters the forward or backward links in the _EPROCESS block of its 'neighbouring' processes so that they each point to the other, missing out the process in the middle (the malware) . This process is shown in the diagram below:

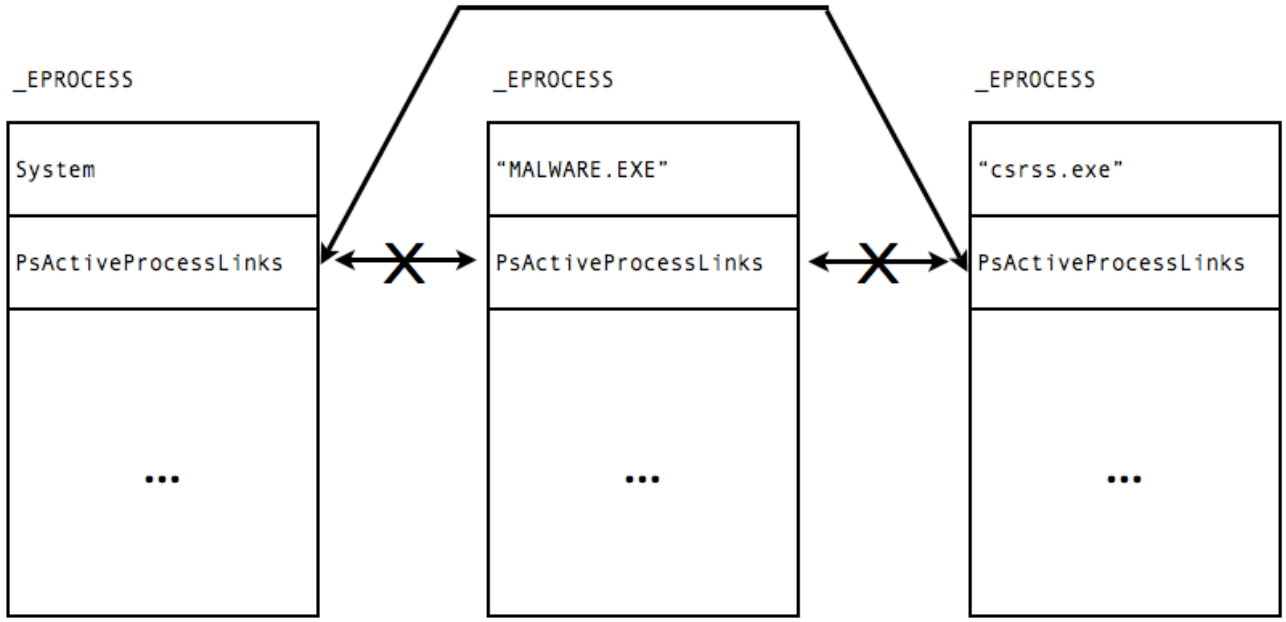

Figure 1: Direct Kernel Object Manipulation

This technique is a form of Direct Kernel Object Manipulation [13] and is one of the primary methods of data hiding. There are two possible solutions to this problem: the first is to search for the characteristics of the _EPROCESS block in the RAM image. However, this relies on educated guesswork and it is possible for false-positives to be found, although if good techniques are used the chances are reduced. The second would to parse the Windows thread queuing system to get a list of all threads, since if a thread is running it will be in a queue called the 'Dispatcher Ready Queue'. A thread is represented here by an '_ETHREAD' kernel object, which contains a pointer the the PEB of its owning process from where more information can be found. Unfortunately though, insufficient documentation for the thread scheduler is publicly available for this to be done without reverse engineering.

Basic Process Details

Once all the running processes and their process blocks have been found, basic information can be recovered for each. The following table shows some of the more useful entries:

Field Name	Offset	Description
DirectoryTableBase	+0x018	Page Directory Base *
KernelTime	+0x038	Time spent in kernel mode
UserTime	+0x03c	Time spent in user mode
UniqueProcessID	+0x084	The PID of the process
ActiveProcessLinks	+0x088	The VA of the APL field of the previous and next EPROCESS
ObjectTable	+0x0C4	VA of table of handles to objects
InheritedFromUniqueProcessID	+0x14C	The PID of the process' parent process
ImageFileName	+0x174	The name of the executable file

Table 1: Useful fields in the _EPROCESS structure

* The Page Directory Base is the physical address of the Page Directory used in virtual to physical address translation for this process, and will be discussed in Section III.

Process Environment Block

The Process Environment Block is the part of the process descriptor which resides in the process' address space. Since Windows XP is a fully 32-bit OS, each process has it's own 32 bit virtual address space, which means that addresses are only applicable to the address space of the process which is using them. Since the PEB contains the addresses of many structures resident within the process' address space, it is much more efficient to keep it here. To access the PEB, a virtual to physical address translation must be performed, using the PDB from the _EPROCESS block, and the address of the PEB which is also stored in the _EPROCESS. The PEB of the System process points to null because the _EPROCESS block is already in its correct context, and the System process is not started from an executable. Virtual to Physical address translation is discussed in Section III.

When a program is executed, a copy of the executable file is loaded in its entirety, into memory; its address is stored in the PEB at offset +0x008. This file can be carved from the memory dump and saved as a file to be run, analysed or compared with a known original.

In addition, the PEB also contains a list of modules (DLL files) the process is using, files to which it has an open handle, virtual address of the process heap, code area and shared memory. These fields and their offsets into the PEB are shown below:

Field Name	Offset	Description
ImageBaseAddress	+0x008	Virtual address of executable image
Ldr	+0x00C	VA of structure containing module list
ProcessHeap	+0x018	VA of start of Heap for this process

Table 2: Useful fields in the _PEB structure

Interpretation

Obviously all this data is useful only if it can be interpreted in a meaningful way. Starting with the Process List, this can show what programs were running at the time the memory image was taken. It is important to note that depending on the method used to acquire the memory image, some processes may be shown which are part of the acquisition process - such as '*dd*', and as noted above some processes may be intentionally hidden in which case more advanced techniques are required to list them.

Of the basic properties, the name of the process is useful because it will give an indication of what the process is, particularly if it is a suspicious-looking one. However, although the name may appear to be that of a trusted process it is possible to disguise the real identity using methods similar to DKOM shown above: if a process is loaded with the same name as a trusted process (e.g. *cssrs.exe*), and the forward and backward links of the 'real' *cssrs.exe* are modified to hide it, the 'fake' *cssrs.exe* will appear as the only entry unless more sophisticated process listing techniques are used. 'KernelTime' and 'UserTime' can be combined to show how long a process has been running, which may be of use if an alleged incident occurred before that process had been started. The 'PID' and 'PPID' can also be used to show the source of a suspicious process: child processes of '*explorer.exe*' are likely to have been started locally by a user at the keyboard or run at start-up; processes whose PPID is a network service are more likely to have been started remotely. The 'ObjectTable' field of the '_EPROCESS' points to a table apparently containing the addresses of structures describing the objects to which the process has open handles (such as files and registry keys), but documentation was not sufficient to examine this any further. Likewise the 'Ldr' field in the '_PEB' contains the address of a '_PEB_LDR_DATA' structure with details of DLL files loaded to assist the program. In some cases this may be of evidentiary value, but again there is insufficient documentation to further analyse it. Probably the most useful value in this section is the 'DirectoryTableBase', which can be used to translate virtual addresses into physical ones, as shown in the next Section.

Once the PEB has been found, more information can be recovered: namely the image file and the process heap. The image file can be carved from the memory dump by reading the contents of the virtual address specified. The length of the file can be extracted from the EXE header using the following offsets[14]:

Offset	Field
`+[0x4-0x5]`	The number of (512 byte) blocks in the file
`+[0x2-0x3]`	The number of bytes of the final block to be read (or all, if 0)

Table 3: offsets into the image file showing file size

Once the file is extracted, it can be compared to known good copies of what it appears to be by generating hashes of both. If it is not what it appears to be, it can be disassembled, tested or otherwise analysed to discover its functionality.

Another important source of evidence available through the PEB is the Process Heap. This is the section of memory (a range of virtual addresses) from where segments of memory are allocated to the process when new variables are initialised. Therefore, any data stored by a program should be available in this space. However, each application organises its data differently depending on the programming language it was written in, compiler used to compile the source code, program design and the target platform against which it was compiled (in the case of Windows XP this is normally x86 but sometimes x86-64-bit or IA64). Analysing the memory of a particular process requires detailed knowledge of all of this and so writing software which can do this requires both time and access to often restricted information (such as the original source code and build environment). However, some of the tools included with *FATKit* do attest to automate at least a part of this process and so this procedure may yet prove to be possible.

This page intentionally blank

III. Windows Memory Management

All modern operating systems use some form of Virtual Memory system to allow processes access to a large address space while preventing them from gaining unnecessary and dangerous access to data belonging to other processes. The VM system used by Windows XP allows processes to access a full 4GB of virtual addresses, translating those virtual addresses into requests for real physical addresses in memory on the fly by way of a paging system.

The Paging System

To allow each process to address a full 4GB of memory without requiring excessive amounts of RAM, a paging system is used for each process. This allows only those virtual addresses which are in use to be mapped onto locations in physical memory. The paging system is organised as follows: each process keeps a Page Directory which contains 1024 references to Page Tables. Each Page Table contains 1024 Pages, which each refer to 4096 addresses. This gives 1024*1024*4096 or 4G possible addresses, while storing the whole paging structure in only 4MB. A 32 bit Virtual Address is split into 3 parts as shown below (example address 0x81291830):

Page Directory Index [0-9]	Page Table Index [10-19]	Byte Offset [20-32]
1000000100	1010010001	100000110000

Figure 2: Format of a Virtual Address

This allows a 10 bit index into the Page Directory, a 10 bit index into the Page Table and a 12 bit index into the Page. The Page Directory Index is in fact an offset from the physical address given in the Page Directory Base stored in the EPROCESS block of the process and mentioned in the previous section, and points to a Page Directory Entry containing information including the address of the Page Table. The physical address of the Page Table Entry can be found by combining the Page Directory Base of the process with the address from the PDE. Likewise, the Page is found by combining the address from the PTE with the address of the Page Table. Finally, the Byte Offset is added to that address to find the data. The specifics and exceptions of this process are discussed later.

Page Table Entries

The Page Directory is composed of Page Directory Entries, and the Page Table of Page Table Entries, both of which have effectively the same format. These contain a pointer to the required Page Table or Page, respectively as well as other useful information. They have the following structure:

Page Frame Number		V
0 19	20 30	31

Figure 3: Format of a valid Page Table (or Directory) Entry

There are three important fields: bits [1-19] contain a pointer to the location of the Page Table or Page; bit [31] is a validity flag which indicates whether or not the page is accessible in physical memory; bits [20-30] are only useful if the page is not accessible in physical memory (see below).

Page Faults and the Page-file

In some cases a process will require more memory than is available in physical RAM, either because one process requires a large amount or because there are many processes running concurrently. In this case, some of the less used pages of memory are 'swapped out' to the hard disk. When a process makes a request for that page of memory, the page is swapped back in to RAM and (if necessary) another page is swapped out. When a process requests a swapped out page, a 'page fault' is generated (also known as a 'valid page fault' and not to be confused with an 'invalid page fault' as seen in many error messages). This causes the VM system to interpret the format of the PDE or PTE differently, and look in the page-file(s) for the requested page instead. The page-file is generally called '*pagefile.sys*' and stored on the root of the primary drive, but settings should be checked to see if there are multiple pagefiles and where they are stored. The format of the page-file is very simple, and can be treated exactly the same as memory: it is simply a contiguous block of data which can be referenced by the same offsets as used for physical memory.

When a page is no longer available in memory it becomes 'invalid' and its PTE or PDE is altered accordingly. An invalid PTE or PDE will always have it's least-significant-bit set to zero, but there are a number of reasons for it being invalid each of which uses a slightly different PTE/PDE format:

- Page file: The requested page or page table has been swapped out to disk. It can be retrieved by reading [O] offset into page-file number [N] as shown below. Transition [T] and Prototype [P] fields are zero and [N] is non-zero.
- Demand Zero: The request requires a page of zeros so any such request can be satisfied by returning a '0'. Fields [O], [T], [P], [N] and [V] are all zero.
- Transition: The requested page may have been modified since the Page Table was last updated. This is the same as a valid PTE except that field [V] is zero.
- Prototype: The requested page may be shared with another process. The structure is the same as for a valid page but does not point to a real address (it points to an additional structure called the *Page Frame Number Database*) so this can be treated as an invalid address for simplicity.

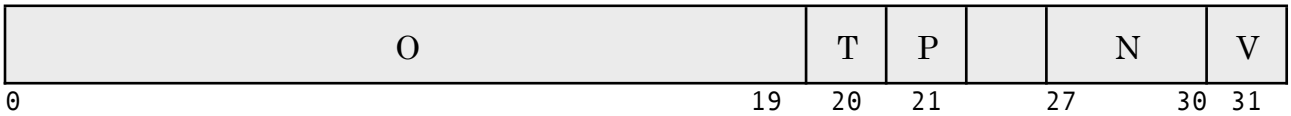

Figure 4: Structure of an invalid PTE

Address Translation

As mentioned, address translation starts with the Page Directory Base of a process, and a Virtual Address. Provided with these two values, access to a memory dump file and in some cases the page-file, an address can be translated from virtual to physical and the contents of the memory at the provided address can be found. The procedure is shown below. Bits are numbered with Most Significant Bit = 0.

To aid with the following explanation, an example will be followed with these variables (taken from a test system):

VA	PDB
0x81291830	0x00039000

1. Recover the Page Directory Index from the Virtual Address by taking the first 10 bits of the VA (1000000100 binary) and multiplying by 0x4 (since each PDE is 4 bytes), giving 0x810.
2. Add the PDI (0x810) to the PDB (0x39000), to give the physical address of the Page Directory (0x39810).
3. Read the value at that address (a 32 bit value, remembering that x86 is little-endian) to give the PDE (0x01222163).
4. The physical address of the Page Table depends on the state of certain bits in the PDE: if bit [31] is 1 then the page is valid and bits [0-19] can be read, multiplied by 0x1000 (each page is 4096 bytes) and used as the physical address in RAM. If bit 31 is 0 then the page is not accessible in physical RAM. In which case:
   - If bits [0-20] and [26-30] are zero then the PTE is a 'demand 0' so a zero page can be returned;
   - If bit [21] is one then the page is a prototype which may mean the address is shared, so for simplicity can be treated as invalid.
   - If bits [21-22] are zero then the page table has been swapped out to disk, which means bits [27-30] are taken as the page file number (normally 0), and bits [0-19] are taken as the offset into the page-file. Reading the location of that offset will give the address of the Page Table.

   In the example, the result is address 0x1222000 (from physical memory).
5. Recover the Page Table Index from the Virtual Address by taking bits [10-19] of the VA (1010010001 binary) and multiply by 0x4, giving 0xA44
6. Add the PTI to the value of the Page Table read in step 4: this gives us 0x1222A44
7. Read this address: this is the Page Table Entry (0x011F2163).
8. As with step 4, the location of the Page is dependant on the PTE: for a valid page bits [0-19] can be read, multiplied by 0x1000 and used as the physical address of the Page in RAM. Otherwise the page is invalid, in which case the same applies as

with step 4. In the example, the PTE is valid and results in the address 0x11F2000.

9. To get the physical address of the exact virtual address, add the Byte Offset, which is the given by the last 12 bits [20-31] of the virtual address. This results in 0x11F2830, which conforms to the output of the !ptov command in the kernel debugger.

This process will give the physical address referenced by a virtual address, provided that the address points to a piece of data either in the physical memory dump or the page-file.

IV. Memory Acquisition

According to published guidelines [44,46] the contents of piece of digital evidence should be collected in order from the most volatile to the least. This means that the one of the first things an incident response team should do is attempt to secure a copy of the RAM on any running machines at the scene. The problem with this though, is that the guidelines also state that no steps should be undertaken which might compromise the state of the suspect device. While these rules may be relaxed the reasoning is sound and the impact of imaging techniques must be considered.

Software Solutions

A software memory imager is one either run by the suspect computer or run using software already present on the suspect computer (such as the operating system), which is inherently problematic for a number of reasons. Firstly, some of the software must be run by the existing operating system. It is possible to modify the operating system to redirect certain procedure calls in such a way as to give erroneous or unaccountable results, and so results obtained in this way will never be entirely reliable. Secondly, some data will always be modified during the imaging process because even the most efficient memory imagers have to load data (and therefore replace existing data) in memory. If the old data is swapped out to disk then pages which are no longer in use but have not yet been overwritten will be lost. This problem can be alleviated with very small efficient memory imagers but it cannot be totally removed. At the very least, the memory imaging program will be present in the memory image, and must therefore be accounted for during any investigation. Another problem is that most software solutions run in user mode, and must share processor time with other running processes. Therefore, they will not be able to image the entire contents of RAM before another process takes over execution time from the processor, resulting in data changing during the imaging process. The final problem is that any memory imager running with user privileges may not be able to access the whole of physical memory, as was the case when using '*dd*' during testing: on most attempts to image the whole of the *PhysicalMemory* device there were 2-3 ranges of address which could not be read. These addresses must always be accounted for in an investigation, and there is always the possibility of important evidence being lost because of this. The available tools are now briefly examined in terms of:

- Background and functionality
- Typical usage
- Format of output
- Evidentiary impact
- Compatibility with suspect systems
- Applicability to situations and any other notes.

'dd'

'*dd*' is a program written by George Garner as part of the '*Forensic Acquisition Utilities*' suite. It is very popular, and is used by many incident response teams. In this regard it has been well tested, however to date there are no generally accepted methods for testing memory acquisition programs. It is a variant of the '*dd*' program commonly used to image hard disks, but can also image Windows XP memory via the '`\\.\`*PhysicalMemory*' device node.

- Typical usage:

```
D:\> dd.exe if=\\.\PhysicalMemory of=D:\dump.dd conv=noerror > D:\dump.err
```

- The output file is an exact copy of physical memory: reading offsets from the start of the generated file gave the same result as shown by the kernel debugger.
- When running the file, it exists as a single process with a consistent memory footprint of 2916kB on the test platform.
- The output file could be analysed using a hex editor, but the only tool available which can parse it is *WMFT* which is still a very early beta version. This, and the information presented here and at the DFRWS 2005 show that it should be possible to analyse in time.
- It can be run on any 'modern Windows system' apparently up to XP SP2 but not on later versions (i.e. XP x64 edition, Windows Server 2003 and Windows Vista) because *PhysicalMemory* device node is no longer accessible to user space programs [3]. A successor to *dd*, '*KnTDD'* is currently being developed by Garner, which may overcome this problem.
- It does not require any pre-installation and so can be used in incident response, although it does make use of the operating system and so results can never be entirely trustworthy.

'pmdump'

This is a tool written by Arne Vidstrom, a security researcher at the Swedish Defence Research Agency. It was designed to "dump the memory contents of a process to file without stopping the process"[4].

- Typical usage:

```
D:\> pmdump 123 process.dmp
```

- Since no analysis programs are available which can parse a memory dump of a single process (except for *MMP* which uses a proprietary format) it was not possible to confirm the format of the output. It does appear to be a raw dump of the contents of RAM after the whole address space had been paged in, since the size of the output file matched the size of the total memory usage reported by Process Viewer.
- Unlike '*dd*', a process ID must be specified which requires a process lister to be run before the image is taken, although *pmdump* can be run with the '-l' switch to do this. *pmdump* runs as a single process and has a memory footprint ranging from

~1MB up to ~10MB for a 60MB process. In addition, the memory footprint of the process being dumped increases to approximately the size of the full memory image shown by Process Viewer, which indicates that mapped files are included in the dump, and possibly pages swapped to disk. This may have implications with overwriting existing data in memory.

- *pmdump* is advertised as running on Windows NT4 and all versions since 2000, although only minimal testing has been carried out for this report on XP and SP2.
- It does not require prior installation or setup so is also suitable for incident response situations, although as before it makes use of untrusted OS components.

'Process Dumper'

'*PD*' is a tool written by Tobias Klein, an IT-security consultant at 'cirosec GmbH'. It is designed to work with another of Klein's tools, '*Memory Parser*', by dumping the whole of a process' memory space including data and code mappings. It also includes meta-data about other aspects of the state of the system, but since it is closed source it is not possible to tell exactly what.

- Typical usage:

```
D:\> pd -p 123 > process.dmp
```

- Output is directed to '*sdtout*' which means that it can be redirected to a file or network service to avoid writing to disk. The format is proprietary, but from the literature available appears to be a combination of physical memory, swap space, mapped files and meta-data.
- As with *pmdump*, a process ID must be specified but the program does not have the capacity to list running processes so another program must be run first. *PD* itself runs as a single process with a fairly small memory footprint up to 1240KB when dumping a large (60MB) process.
- It worked on the Windows XP and SP2 test system, and is advertised as working on 'Windows', but it was not possible to test which versions. Its output could be parsed by '*Memory Parser*'.
- Again, *PD* does not require prior installation and so is suitable for IR situations, but with the caveat that results will not be entirely trustworthy.

User Generated Stop Error

It is possible to force Windows to generate a 'Stop Error', which halts processing of all tasks and copies the contents of memory to a specified file [36]. The system will halt when the user enters a certain key combination.

- Usage: to allow the key combination to take effect, the following registry key must be added and the machine restarted:

```
\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i802prt\Parameters

Name: CrashOnCrtlScroll, Type: REG_DWORD, Value: 1
```

The system will then halt if the user presses the right Control key together with the Scroll Lock key twice in succession. The system must also bet set to dump to a file: by navigating to '*System Properties*' > '*Startup and Recovery: Settings'* the type of dump and location of the dump file can be set. The dump file will only be saved to local hard disks though; it is not possible to save directly to a removable device.

- This procedure generates a 'crash dump' file, which can be analysed using the '*Debugging Tools for Windows'* suite of utilities. Using these tools, many operations can be performed to recover the state of the machine at the time of the crash dump.
- Since this method halts the system, no data can be altered while the dump is in progress. In this regard, evidentiary consistency is good. However, when the method was tested it was found that a block at the beginning of the page-file equal to the size of the dump file was overwritten with the contents of the memory dump, which means that a large amount of data in the page-file is lost.
- This functionality is available on all versions of Windows since 2000 with the possible exception of XP 64-bit.
- One major drawback of this procedure is that the machine must be set up and restarted before the key combination will take effect. Obviously this not appropriate for IR situations, since the contents of memory will be lost. It could be useful though in covert operations where the investigator knows that a particular computer is at risk of being compromised. The crash dump file is very easily analysed, and produces a 'snapshot' of the state of the computer, removing the problems associated with data changing during the acquisition process. Based on the literature available from Microsoft, the functional code which triggers the stop error is within the 'i802prt' keyboard driver[36], and so it may be possible to reverse engineer this driver and discover a way to force the stop error without having to restart the machine.

'Userdump.exe'

This utility[38] provided by Microsoft allows the user to produce a process dump in format compatible with the Debugging Tools for Windows, but requires a process to generate a fatal error. Since this is hard to cause manually and different for each process, it is not very suitable for forensic use.

MS Special Administrative Console

This mode of operation is specific to Windows Server 2003[37]. When setup on a target machine, it allows users to log into the machine remotely and issue a variety of commands over a text console, one of which is 'crashdump'. This performs the same purpose as the User Generated Stop Error shown above but is better suited to network environments.

Hardware Solutions

Recently a few alternatives to software-based acquisition have been proposed. These techniques have the advantage of gaining direct access to the memory via DMA, which means that they can copy the entire contents of RAM without having to modify it in any way. In addition, they can copy it very quickly or instantaneously, which reduces the risk and effect of data changing during the imaging process.

Tribble

Brian Carrier and Joe Grand have designed and developed a prototype hardware device which can be installed in the PCI slot of a PC in advance of an incident occurring. It can be activated at any time to immediately copy the contents of the RAM to an external storage medium. It will also attempt to halt the processor of the target machine thus ensuring that no changes are made to memory during imaging. Although its requirement for being installed before the occurrence of an incident renders it largely useless for incident response, situations where an attack is suspected or covert surveillance is possible could benefit greatly from the increased reliability of the evidence it provides. One other major advantage is that the output is a raw memory dump and so should be compatible with parsers developed to be used with software solutions like '*dd*'. *Tribble* is still in development as of this writing, and so was not available for testing.

Firewire

Research carried out by Maximillian Dornseif et al has shown that it is possible to access memory of a target machine via the Firewire interface. Experiments have been carried out using Mac OS computers as analysis machines and a variety of platforms as the target, including Mac and FreeBSD. The technique makes use of the fact that Firewire connects the the internal bus via the Open Host Communication Interface, which can negotiate DMA between the connected device and memory. From the limited documentation it appears that success is dependant on the Operating System of the target machine, and how it handles filtration of Firewire traffic: the suggestion is that by sending requests formatted in a certain way, some of these filters can be overcome. Attempts to use this technology on Windows XP and 2000 computers have not yet been successful though.

Hibernation

'Hibernation' is system used by Windows to allow systems to save the contents of volatile memory to disk, allowing them to power down without permanently losing that data. It has been suggested [6] that some useful information might be available in the hibernation file, '*hiberfil.sys*' which is created whenever the computer is put into hibernation.

It is also possible that that hibernation could be a useful way to acquire the contents of memory, since no additional programs would have to be loaded. Unfortunately though, there are severe problems associated with this. Firstly, when the system is hibernated it writes data to a file, *hiberfil.sys*. If this file exists it will be overwritten, therefore the old file should first be removed. Secondly, enabling hibernation on systems where it is disabled requires extra steps to be taken, increasing the risk of

data being changed. The main problem though, is that the hibernation code for Windows XP is contained in the '*ntldr*' file which is closed source and so without thorough testing it cannot be reliably shown what the results of hibernation will be. It is also not clear exactly what data is stored: according to Microsoft [39,40] to speed up initiating and resuming from hibernation only those pages currently in use are written to disk, and they are compressed using a proprietary compression algorithm before being written. This means that the compression algorithm and file format must be discovered before the hibernation file can be read. It also means that there is no opportunity to analyse free space in memory for residual data.

If these problems could be overcome it may be feasible to use hibernation in the following way:

- Copy the *hiberfil.sys* file
- Hibernate the system
- Clone the disk
- Use the clone to resume the system, producing a system in the same state as when it was hibernated.

However, for this to be a viable option much more research is needed into the hibernation process and the format of the hibernation file.

V. Memory Analysis

Once an image of memory has been taken from a suspect device, it must be analysed so that meaningful information can be extracted from it. Up until recently, if memory was examined at all it was often only searched for text strings and common file types using file carvers such as '*foremost*' and '*scalpel*'. As mentioned previously, these methods can recover some useful pieces of information, but the context in which they the information is stored is lost. The field of memory dump analysis is newer than that of acquisition, but some tools have been developed. The only publicly available tools designed specifically for memory analysis are '*Memory Parser*' by Tobias Klein and '*WMFT*' by Mariusz Burdach, although Microsoft's '*Debuggin Tools for Windows'* is also capable in certain circumstances. In addition, three tools are currently under development but not publicly available: '*KnTList*' by George Garner and associates; '*mem_parser*' by Chris Betz and '*FATKit*' by a team based at '4tphi.net'.

The information presented so far in this report shows that at least the following tasks are possible for an automated tool:

- Show data about running processes, equivalent to the level of detail available from an advanced process lister
- Recover a copy of the image of the executable which loaded the process, so that it can be compared to the original from the disk or analysed further
- Display the memory space allocated to each process from the heap, so that this can be parsed by a system with specialist knowledge of the program or searched for text and images.
- Incorporate the page-file if one is available, allowing the full range of each process' virtual address space to be read

The available tools are discussed briefly in terms of:

- How much of the above information they can gather
- How compatible they are with acquisition techniques
- How useful they are in the field of Incident Response

Available Tools

MMP

This tool was written by Tobias Klein and designed to be used with his '*PD*' process dumper. It uses a GUI from where the user can open a dump file and have *MMP* parse it. It's original intent was to show situations where malicious code had been injected into the data area of a process. To aid in this, it shows the process' address space in terms of which areas are mapped to data and which are mapped to code. In the case of code mappings, the originating file is also shown.

- *MMP* can recover: all required information about the process; the image file; the full address space of the process and it appears to incorporate the page-file by loading the entire process space into physical memory during the dump.
- Only files generated by *PD* can be parsed, apparently because *PD* adds meta-data gathered from the target system during the acquisition process, and also possibly due it its requirement for having the whole address space in physical memory.
- In relation to incident response, this tool will not normally be applicable since it only operates on single processes. Attempting to capture all running processes would cause excessive data loss, but in situations where a specific service had been compromised (e.g. a web server), it may be more useful.

WinDbg

WinDbg is the GUI component of the *Debugging Tools for Windows*, which allow users to analyse a running system or memory dumps caused by system crashes. Commands can be entered to reveal certain aspects of the state of the system at the time of the crash, provided that a symbol database is available for the process being examined. All globally defined symbols for the operating system are available, but private symbols and those defined for user-space programs are not. Once a crash dump has been opened and parsed, there is a huge selection of commands which can be issued, the most useful of which are shown in the following table:

Command	Description
`dt [type] ([va])`	Shows the composition of the structure [type], optionally with values if the structure starts at address [va]
`!process 0 7`	Shows all available details about all running processes (change 7 to 0 for a simpler view)
`!ptov [dirbase]`	Displays the physical-virtual address mapping of the process whose PDB is specified by [dirbase]
`!vtop [dirbase] [va]`	Displays the physical address of [va] using [dirbase] as the PDB

Table 4: Useful WinDbg commands

Between them, these commands will provide most values describing the kernel space, and some information about the processes (such as the address of the image file, process heap and modules), but it is not possible to use *WinDbg* to recover the executable image file from the dump.

- *WinDbg* requires dumps to be in the 'crash dump' format generated by Windows as a result of a forced core dump. There are three ways to force a core dump, as mentioned in the previous Section.
- Of the three ways to cause a crash dump, only two are useful for forensics purposes (*Userdump.exe* requires a process to generate a fatal error but the method of causing this is process-dependent and overly complex). The '*CrashOnCtrlScroll*' method, and

issuing a 'crashdump' command over the *SAC* are both restrictive because they require the software to be set up in advance of the incident occurring. That said though, in covert analysis or situations where systems are expected to be the victim of an attack, the level of information and ease of extraction provided by the debugging tools could make this prior setup worthwhile.

Windows Memory Forensic Toolkit

This tool is under active development by Mariusz Burdach, with version 0.2 available at the time of writing. Due to its very early beta status, little of its intended functionality has been implemented and what is working is difficult to use. Despite many attempts, using a variety of values for the '*RVA*', '*offset*' and '*initproc*' variables no meaningful output was produced. This is probably due to the fact that it was developed and tested on Windows Server 2003 which stores structures (particularly the PDB) at different locations than the test system.

Tools Under Development

There are a few tools currently under development which appear to incorporate some of the findings of this report: *mem_parser* by Chris Betz and *KnTList* by George Garner which were showcased at the DFRWS 2005 Memory Challenge, and *FATKit* which is being developed by '4tphi.net'. *mem_parser* and *KnTList* appear to be similar, taking a *dd* image and recovering a variety of information including hidden processes and image files. *FATKit* takes a different approach, offering a framework and abstraction layer covering the low-level issues of memory analysis (such as address translation) which will allow further tools to be more easily developed.

This page intentionally blank

VI. The 'vtop' Proof-of-Concept

Intent

This tool was developed to prove that the virtual address space of a process could be reconstructed by combining a physical memory dump with a page-file to produce a single 4GB file containing all the memory allocated to that process.

Tools and Design

Since this was only intended to be a proof-of-concept,the normal requirements of modularity, scaleability and performance were sacrificed in the interests of development time. Python was chosen as the language because it supported all the necessary OS functions such as file handling, and it also allowed types to be implied and (fairly) easily converted which was necessary when dealing with a mixture of base 2, 10 and 16 numbers. Using the development environment '*IDLE*' allowed code to be written and executed in real-time so new functions could be tested without having to reload the whole system.

Testing and Problems

Since this program is only a proof-of-concept it was not tested as thoroughly as a production piece of software would have been. Some structures were found in the output at the expected addresses, which indicated that it basically worked even if there were some bugs present.

One of the first problems encountered was with reading large files. Due to limited experience with Python, the first version of *vtop* attempted to read the entire contents of both the memory dump and the page-file into memory, which not only took a very long time but also generated IO exceptions if the swap-file on the test system was not large enough. This was later overcome with the *seek()* function which allowed only small parts of these files to be read at a time.

A bigger problem, and one which may prevent the current implementation from being usable is that the *range()* and *xrange()* functions commonly used in Python to iterate over a *for* loop cannot handle very large ranges: *ptov* should iterate over all 2^{32} possible virtual addresses but the *xrange()* function could only generate 2^{31} iterations. One solution is to only iterate over the lower 2GB of the address space since the upper 2GB are not writable by the process anyway. It may also be possible to rewrite the *range()* function or manually set the range. A better solution would be to rewrite the program in a more appropriate language such as *C*.

By far the biggest problem though, was the difficulty in testing parts of the program. No way was found to confirm whether or not data recovered from the page file was correct, since it was not possible to discover what the results should have been. All the important data structures which could be verified (such as the _EPROCESS block or structures connected to it) are never paged to disk; only user-space processes seem to be swapped out. The only obvious way to test this would be to write a program

which writes specific data to a specific virtual address; make sure that process was paged to disk; acquire the memory and page-file without allowing the process to page back into RAM and then search at the specified address for the specified data. This would have been very difficult (assuming that fine control of the Windows VM system is possible) and time-consuming to write, and so with an already short timescale to complete the report it was not possible.

Improvements

If the tool were to be developed beyond proof-of-concept status, it would be useful if it gave the user a few more options: specifying the PDB, input and output filenames and address range either on the command line, interactively or in a configuration file would be more user-friendly. Output format should also be configurable: currently it is set to output a single large (up to 2GB) file for the process, which may not always be appropriate. It could also output in tab-, comma- or space-delimited format showing address and data so that addresses could easily be searched for and invalid addresses would not be printed. This would result in a much smaller output file, faster run-time and in certain circumstances more easily readable output. In the longer term it would also be useful if it could list the processes present in a memory image and allow the user to select one for reconstruction. This would require a lot of extra work though, especially if it were able to find processes hidden through DKOM.

Applications

Despite its limitations, the tool could be useful in certain circumstances. If a certain process is suspected of storing a clear-text pass-phrase, illegal image or some other easily recoverable item the reconstructed address space could be searched with the knowledge that any results must have been associated with that process. It is also possible to recover the virtual address at which the item was found, which is not possible using tools such as '*PD*' or '*pmdump*'. This information could prove very useful if information about the structure of the program could be found.

VII. Outroduction

Conclusions

This report produced the following conclusions

Recoverables

- Given a physical memory dump and page-file it is possible to recover all available information about process including its image file, and reconstruct its entire virtual address space;
- Kernel structures can be parsed quite easily once the physical address of one process descriptor is found, and this can be done by searching the memory image for the strings "00900300" and "53797374656D0000" 0x15C bytes apart (in XP).
- Processes descriptors can be hidden through DKOM, making them harder find.
- Virtual Addresses are process specific but can be easily translated with a memory image and a page-file once the process descriptor is found

Tools

- *dd* remains the only usable tool for non-covert incident response, even though it relies on untrusted code and can fail to capture parts of memory. It's output is unformatted, making analysis more generic. However, the current version does not work on versions of Windows released since XP SP2, due to access restrictions on the \\.*PhysicalMemory* node.
- *PD* and *MMP* are useful for dumping and analysis of single processes, particularly cases which are suspected of being the victim of code alteration or remote code injection.
- The methods involving generation of crash-dumps are useful for covert analysis or installation in systems which are expected to come under attack. These methods halt the system so no changes can be made once the dump has started, but they do overwrite the page-file. Debugging these crash-dumps can easily yield useful information, but they are not compatible with other analysis techniques and some items such as the image file will be very hard to recover.
- When the *Tribble* PCI card becomes available it will be the most appropriate solution for covert analysis or use as a pre-emptive measure, since it does not modify the state of the machine and does not allow data to change during acquisition. Data should also be easy to analyse using forthcoming software. The downside though, is that it may be expensive and will require foresight to install in appropriate systems.
- No publicly available tool currently exists which can extract useful information from a raw memory image such as that produced by *dd*, but a number are under development: *WMFT, mem_parser* and *kntlist* will be able to extract most of the necessary information when they become available.
- *FATKit* when it is released may allow for more complex tools to be written such as parsers for memory systems of other platforms.

- It may be possible to use hibernation as a method of acquiring memory, since it appears to halt the system and dump all used pages of memory to disk. Care should be taken though for a number of reasons: data on the disk will be overwritten and should be removed first if possible; hibernation relies on proprietary code on the host system and is therefore neither fully understood or trustworthy; the result of hibernation is a *hiberfil.sys* file whose format will be hard to analyse, although using a cloned of the disk containing the file to resume the system may be a viable procedure.

Further Work

This paper has highlighted a number of areas of memory forensics which could greatly benefit from further research:

- The *FATKit* framework should allow for memory analysis tools to be developed much more easily, due to its abstraction of much of the low-level functionality. This will hopefully allow for more advanced programs to be produced which could, for example analyse data structures and variables within process space and produce visualisation systems for displaying the contents of memory in readable ways.
- Although some tools (*'KnTList'* and *'mem_parser'*) appear to be able to recover processes hidden through DKOM, it would be useful to discover how this is done, and how reliable the results are. If necessary the thread scheduler could be reverse engineered to recover the queued threads and their parent processes.
- Currently, there is no way to trigger a stop error and cause a crash dump on Windows is the system has not been set up in advance. The *'CrashOnCrtlScroll'* method uses the *i802prt.sys* keyboard driver to trigger this error, and so it may be possible to reverse engineer this driver and discover how to trigger the error without having to edit registry and restart. A tool could be developed which sent the same signal as the driver to initiate a crash dump to a specified location.
- Hibernation may prove to be a useful method of acquiring memory, but work needs to be done. The ways in which Windows fetches, compresses and stores memory are not fully understood, and it is not clear if any other operations are performed during the hibernation process (such as ending inactive processes or deleting unused memory). To do this, the *ntldr* code should be reverse engineered in an attempt to discover this. It is also not currently possible to recover data from the hibernation file due to its proprietary format, and so this would also have to be investigated.
- Firewire has been designed to access memory through DMA, which should lend it towards memory acquisition with minimal impact on the target system. Further research is required to discover whether or not this is possible on Windows XP systems and to produce a viable solution which would be suitable for incident response.

Recommendations

A number of recommendations have become clear from the work carried out for this report, applicable both to the investigators and the potential victims of crimes:

- Don't pull the plug! Large amounts of extremely useful evidence can be recovered from volatile storage, but removing the power will destroy this permanently. Although facilities for analysis may not exist at the time of the initial response, the data acquired from memory can be stored until such time as analysis techniques have matured. Also, although performing memory acquisition using software methods does damage a small amount of evidence and relies on untrusted code, this evidence is much less than the amount destroyed by removing power.
- In systems where attacks are expected, steps can be taken to make the investigation process both easier and more productive. In desktop systems where money is an issue, the '*CrashOnCtrlScroll*' registry key should be set and suitable memory dump settings applied, which will allow investigators to easily cause a crash dump and analyse it. For servers, the '*Special Administrative Console'* should be enabled so that a crashdump (or any other analysis) can be performed remotely. Once it is released, critical systems should have the *Tribble* PCI-based memory capture card fitted which allows corruption-free memory acquisition and later analysis using any available tools.

Summary

This report has shown that during the forensic process as much attention must be paid to volatile memory as is paid to the more traditional sources of evidence. It should no longer be standard practice for the entire contents of memory to be destroyed in the interests of the integrity of the data on the hard disks. Different techniques are either available now or will be available in the near future for acquiring memory depending on cost, the ability to anticipate an attack and desired output. Once a memory image has been taken it has been shown that it is relatively easy to combine with the page file to reconstruct the entire address space and map a process' virtual address onto this combined address space. There are few tools currently available which can analyse this data, but sufficient information exists for it to be done manually, and automated tools are under development. The imminent release of the *FATKit* framework will hopefully allow developers to add to this arsenal easily. It is expected that in the near future full suites of memory analysis software will be available, and the investigation of memory will be viewed with the same importance as any other areas of computer forensics.

This page intentionally blank

Appendix A: References

Aaron Walters
[1] *'FATKit: Detecting Malicious Library Injection and Upping the "Anti"'*
http://www.4tphi.net/fatkit/papers/fatkit_dll_rc3.pdf

Andreas Shuster
[2] *'Forensics Blog: Memory Analysis'*
http://computer.forensikblog.de/en/topics/windows/memory_analysis/

Arne Vidstrom
[3] *'Forensic memory dumping intricacies – PhysicalMemory, DD and caching issues'*
http://ntsecurity.nu/onmymind/2006/2006-06-01.html
[4] *'pmdump'*
http://www.ntsecurity.nu/toolbox/pmdump/

Association of Chief Police Officers (Scotland)
[5] *'Good Practice Guide for Computer Based Electronic Evidence'*
http://www.acpo.police.uk/asp/policies/Data/gpg_computer_based_evidence_v3.pdf

Brian Carrier, Joe Grand
[6] *'A hardware-based memory acquisition procedure for digital investigations'*
Digital Investigation Vol 1, Issue 1
[7] *'Tribble'*
http://www.grandideastudio.com/portfolio/index.php?id=1&prod=14

Chris Betz,
[8] *'DFRWS 2005 Challenge Report'*
http://www.dfrws.org/2005/challenge/ChrisBetz-DFRWSChallengeOverview.html
[9] *'Overview of mem_parser tool'*
http://www.dfrws.org/2005/challenge/memparser.html

CERT
[10] *'CERT® Advisory CA-2003-04 MS-SQL Server Worm'*
http://www.cert.org/advisories/CA-2003-04.html

Common Vulnerabilities and Exposures
[11] *'CVE-2004-1038'*
http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-1038

Dan Farmer, Wietse Venema,
[12] *'The Coroner's Toolkit' inc. 'memdump, pcat'*
http://www.porcupine.org/forensics/tct.html

David Solomon, Mark Russinovic
[13] *'Inside Microsoft Windows 2000, 3rd Ed.'*
Microsoft Press, ISBN 0-7356-1021-5

Delorie Software
[14] *'EXE File Format'*
http://www.delorie.com/djgpp/doc/exe/

Fool Moon Software & Security
[15] *'Windows Forensic Toolchest'*
http://www.foolmoon.net/security/wft/

George M. Garner, Robert-Jan Mora
[16] *'Response to specific questions posed by the DFRWS 2005 Memory Challenge'*
http://www.dfrws.org/2005/challenge/RossettoeCioccolato-Responses.pdf

George M. Garner

[17] *'Preliminary Analysis of 2005 DFRWS Forensic Challenge'*
http://www.dfrws.org/2005/challenge/rossettoecioccolato-DFRWSChallengeOverview.pdf
[18] *'kntlist'*
http://www.dfrws.org/2005/challenge/kntlist.html
[19] *'Forensic Acquisition Utilities'*
http://users.erols.com/gmgarner/forensics/

Harlan Carvey
[20] *'Forensic Server Project'*
http://www.windows-ir.com/fsp.html
[21] *'Windows Incident Response' (blog) relevant entries at:*
http://windowsir.blogspot.com/2005_06_01_windowsir_archive.html

Help Net Security
[22] *'Interview with Arne Vidstrom, technical editor of the "Hacknotes Windows Portable Security Reference"'*
http://www.net-security.org/article.php?id=579

'Holy_Father'
[23] *'Invisibility on NT boxes, How to become unseen on Windows NT'*
http://vx.netlux.org/lib/vhf00.html

Igor Nys
[24] *'Process Viewer'*
http://www.teamcti.com/pview/prcview.htm

Ing. M.F. Breeuwsma
[25] *'Forensic imaging of embedded systems using JTAG (boundary-scan)'*
Digital Investigation Vol 3, Issue 1

Joe Grand
[26] *'pdd: Memory Imaging and Forensic Analysis of Palm OS Devices'*
http://www.grandideastudio.com/files/security/mobile/pdd_palm_forensics.pdf

Joe Stewart
[27] *'pmodump'*
http://www.lurhq.com/truman/

Jorge Mario Urrea
[28] *'An Analysis of Linux RAM Forensics'*
http://cisr.nps.edu/downloads/theses/06thesis_urrea.pdf

Mariusz Burdach
[29] *'Digital forensics of the physical memory'*
http://forensic.seccure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf
[30] *'An introduction to Windows memory forensic'*
http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf
[31] *'Finding Digital Evidence In Physical Memory'*
http://blackhat.com/presentations/bh-federal-06/BH-Fed-06-Burdach/bh-fed-06-burdach-up.pdf
Presented at "Black Hat Federal 2006 Briefings", Washington, DC. January 2006
[32] *'Physical Memory Analysis'*
http://forensic.seccure.net/pdf/mburdach_physical_memory_analysis.pdf
Presented at "Innovations in Digital Forensic Practice", Washington, DC. March 2006
[33] *'Forensic Analysis of a Live Linux System' (2 parts)*
http://www.securityfocus.com/infocus/1769
http://www.securityfocus.com/infocus/1773

Maximillian Dornseif
[34] *'0wned by an iPod'*
http://md.hudora.de/presentations/firewire/PacSec2004.pdf
Presented at PacSec/Core04, Tokyo, Japan, Nov 2004

Michael Becher, Maximillian Dornseif, Christian N. Klein
[35] *'FireWire: All Your Memory Are Belong To Us'*

http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf
Presented at CanSecWest/Core05, Vancouver, May 2005

Microsoft
[36] *'Windows feature allows a Memory.dmp file to be manually generated with the keyboard'*
http://support.microsoft.com/kb/244139/
[37] *'Special Administrative Console (SAC) and SAC Commands'*
http://technet2.microsoft.com/WindowsServer/f/?en/library/2acd37af-5439-4789-924c-14e1040cf5a01033.mspx
[38] *'How to use the Userdump.exe tool to create a dump file'*
http://support.microsoft.com/?kbid=241215
[39] *'Fast System Startup for PCs Running Windows XP'*
http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/Fast%20System%20Startup%20for%20PCs%20Running%20Windows.doc
[40] *'Kernel Enhancements for Windows XP'*
http://www.microsoft.com/whdc/driver/kernel/xp_kernel.mspx
[41] *'Debugging Tools for Windows'*
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Monty McDougal
[42] *'Live Forensics on a Windows System: Using Windows Forensic Toolchest'*
http://www.foolmoon.net/cgi-bin/down.pl?ID=13
[43] *'Windows Forensic Toolchest'*
http://www.foolmoon.net/cgi-bin/down.pl?ID=7

National Institute of Standards and Technology
[44] *'Special Publication 800-61: Computer Security Incident Handling Guide'*
http://csrc.nist.gov/publications/nistpubs/800-61/sp800-61.pdf

'Quinn "The Eskimo!"'
[45] *'Firestarter'*
http://www.quinn.echidna.id.au/Quinn/WWW/Hacks.html

RFC
[46] *'RFC3227: Guidelines for Evidence Collection and Archiving'*
http://rfc.net/rfc3227.txt

SysInternals Inc
[47] *'LiveKD overview'*
http://www.sysinternals.com/Utilities/LiveKd.html
[48] *'Process Explorer'*
http://www.sysinternals.com/Utilities/ProcessExplorer.html

Tobias Klein
[49] *'Process Dump Analyses'*
http://www.trapkit.de/papers/ProcessDumpAnalyses_v1.0_20060722.zip
[50] *'All your private keys are belong to us'*
http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf
[51] *'Process Dumper (PD)'*
http://www.trapkit.de/research/forensic/pd/index.html
[52] *'Memory Parser (MMP)'*
http://www.trapkit.de/research/forensic/mmp/index.html

UnxUtils
[53] *'Native Win32 ports of some GNU utilities'*
http://unxutils.sourceforge.net/

Wetstone Tech.
[54] *'Livewire Investigator'*
http://www.wetstonetech.com/catalog/item/1104418/2347979.htm

X-Ways Software Technology AG
[55] *'WinHex'*
http://www.x-ways.net/winhex/index-d.html

This page intentionally blank

Appendix B: The '_EPROCESS' Kernel Structure
(In Windows XP, including _KPROCESS)

```
+0x000 Pcb              : _KPROCESS
       +0x000 Header           : _DISPATCHER_HEADER
       +0x010 ProfileListHead  : _LIST_ENTRY
       +0x018 DirectoryTableBase : [2] Uint4B
       +0x020 LdtDescriptor    : _KGDTENTRY
       +0x028 Int21Descriptor  : _KIDTENTRY
       +0x030 IopmOffset       : Uint2B
       +0x032 Iopl             : UChar
       +0x033 Unused           : UChar
       +0x034 ActiveProcessors : Uint4B
       +0x038 KernelTime       : Uint4B
       +0x03c UserTime         : Uint4B
       +0x040 ReadyListHead    : _LIST_ENTRY
       +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
       +0x04c VdmTrapcHandler  : Ptr32 Void
       +0x050 ThreadListHead   : _LIST_ENTRY
       +0x058 ProcessLock      : Uint4B
       +0x05c Affinity         : Uint4B
       +0x060 StackCount       : Uint2B
       +0x062 BasePriority     : Char
       +0x063 ThreadQuantum    : Char
       +0x064 AutoAlignment    : UChar
       +0x065 State            : UChar
       +0x066 ThreadSeed       : UChar
       +0x067 DisableBoost     : UChar
       +0x068 PowerState       : UChar
       +0x069 DisableQuantum   : UChar
       +0x06a IdealNode        : UChar
       +0x06b Flags            : _KEXECUTE_OPTIONS
       +0x06b ExecuteOptions   : UChar
+0x06c ProcessLock      : _EX_PUSH_LOCK
+0x070 CreateTime       : _LARGE_INTEGER
+0x078 ExitTime         : _LARGE_INTEGER
+0x080 RundownProtect   : _EX_RUNDOWN_REF
+0x084 UniqueProcessId  : Ptr32 Void
+0x088 ActiveProcessLinks : _LIST_ENTRY
+0x090 QuotaUsage       : [3] Uint4B
+0x09c QuotaPeak        : [3] Uint4B
+0x0a8 CommitCharge     : Uint4B
+0x0ac PeakVirtualSize  : Uint4B
+0x0b0 VirtualSize      : Uint4B
+0x0b4 SessionProcessLinks : _LIST_ENTRY
+0x0bc DebugPort        : Ptr32 Void
+0x0c0 ExceptionPort    : Ptr32 Void
+0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
+0x0c8 Token            : _EX_FAST_REF
+0x0cc WorkingSetLock   : _FAST_MUTEX
+0x0ec WorkingSetPage   : Uint4B
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock   : Uint4B
+0x114 ForkInProgress   : Ptr32 _ETHREAD
+0x118 HardwareTrigger  : Uint4B
+0x11c VadRoot          : Ptr32 Void
+0x120 VadHint          : Ptr32 Void
+0x124 CloneRoot        : Ptr32 Void
+0x128 NumberOfPrivatePages : Uint4B
+0x12c NumberOfLockedPages : Uint4B
+0x130 Win32Process     : Ptr32 Void
+0x134 Job              : Ptr32 _EJOB
+0x138 SectionObject    : Ptr32 Void
+0x13c SectionBaseAddress : Ptr32 Void
+0x140 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
+0x144 WorkingSetWatch  : Ptr32 _PAGEFAULT_HISTORY
+0x148 Win32WindowStation : Ptr32 Void
+0x14c InheritedFromUniqueProcessId : Ptr32 Void
+0x150 LdtInformation   : Ptr32 Void
```

```
+0x154 VadFreeHint      : Ptr32 Void
+0x158 VdmObjects       : Ptr32 Void
+0x15c DeviceMap        : Ptr32 Void
+0x160 PhysicalVadList  : _LIST_ENTRY
+0x168 PageDirectoryPte : _HARDWARE_PTE
+0x168 Filler           : Uint8B
+0x170 Session          : Ptr32 Void
+0x174 ImageFileName    : [16] UChar
+0x184 JobLinks         : _LIST_ENTRY
+0x18c LockedPagesList  : Ptr32 Void
+0x190 ThreadListHead   : _LIST_ENTRY
+0x198 SecurityPort     : Ptr32 Void
+0x19c PaeTop           : Ptr32 Void
+0x1a0 ActiveThreads    : Uint4B
+0x1a4 GrantedAccess    : Uint4B
+0x1a8 DefaultHardErrorProcessing : Uint4B
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 Peb              : Ptr32 _PEB
+0x1b4 PrefetchTrace    : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER
+0x1c0 WriteOperationCount : _LARGE_INTEGER
+0x1c8 OtherOperationCount : _LARGE_INTEGER
+0x1d0 ReadTransferCount : _LARGE_INTEGER
+0x1d8 WriteTransferCount : _LARGE_INTEGER
+0x1e0 OtherTransferCount : _LARGE_INTEGER
+0x1e8 CommitChargeLimit : Uint4B
+0x1ec CommitChargePeak : Uint4B
+0x1f0 AweInfo          : Ptr32 Void
+0x1f4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm               : _MMSUPPORT
+0x238 LastFaultCount   : Uint4B
+0x23c ModifiedPageCount : Uint4B
+0x240 NumberOfVads     : Uint4B
+0x244 JobStatus        : Uint4B
+0x248 Flags            : Uint4B
+0x248 CreateReported   : Pos 0, 1 Bit
+0x248 NoDebugInherit   : Pos 1, 1 Bit
+0x248 ProcessExiting   : Pos 2, 1 Bit
+0x248 ProcessDelete    : Pos 3, 1 Bit
+0x248 Wow64SplitPages  : Pos 4, 1 Bit
+0x248 VmDeleted        : Pos 5, 1 Bit
+0x248 OutswapEnabled   : Pos 6, 1 Bit
+0x248 Outswapped       : Pos 7, 1 Bit
+0x248 ForkFailed       : Pos 8, 1 Bit
+0x248 HasPhysicalVad   : Pos 9, 1 Bit
+0x248 AddressSpaceInitialized : Pos 10, 2 Bits
+0x248 SetTimerResolution : Pos 12, 1 Bit
+0x248 BreakOnTermination : Pos 13, 1 Bit
+0x248 SessionCreationUnderway : Pos 14, 1 Bit
+0x248 WriteWatch       : Pos 15, 1 Bit
+0x248 ProcessInSession : Pos 16, 1 Bit
+0x248 OverrideAddressSpace : Pos 17, 1 Bit
+0x248 HasAddressSpace  : Pos 18, 1 Bit
+0x248 LaunchPrefetched : Pos 19, 1 Bit
+0x248 InjectInpageErrors : Pos 20, 1 Bit
+0x248 VmTopDown        : Pos 21, 1 Bit
+0x248 Unused3          : Pos 22, 1 Bit
+0x248 Unused4          : Pos 23, 1 Bit
+0x248 VdmAllowed       : Pos 24, 1 Bit
+0x248 Unused           : Pos 25, 5 Bits
+0x248 Unused1          : Pos 30, 1 Bit
+0x248 Unused2          : Pos 31, 1 Bit
+0x24c ExitStatus       : Int4B
+0x250 NextPageColor    : Uint2B
+0x252 SubSystemMinorVersion : UChar
+0x253 SubSystemMajorVersion : UChar
+0x252 SubSystemVersion : Uint2B
+0x254 PriorityClass    : UChar
+0x255 WorkingSetAcquiredUnsafe : UChar
+0x258 Cookie           : Uint4B
```

Appendix C: Code Listing for '*ptov*' tool

```
#######################################################################
# 'vtop.py version 0.1 alpha                                          #
# Nicholas Maclean in completion of MSc Forensic Informatics 2006   #
#reconstruct the virtual address space of a process given it's PDB  #
#######################################################################

import struct
# --DEFINITIONS
def hextobin(hex):              #convert 8 char 'hex' to 32b binary
    hex = str(hex)
    if hex[0:2] == '0x':        #strip the '0x'
        hex = hex[2:]           #...
    if hex[len(hex)-1] == 'L':  #and the trailing 'L'
        hex = hex[0:len(hex)-1] #...
    padding = 8 - len(hex)      #make hex up to 8 chars
    hex = (padding*'0') + hex   #...
    s=''
    t={'0':'0000','1':'0001','2':'0010','3':'0011',
       '4':'0100','5':'0101','6':'0110','7':'0111',
       '8':'1000','9':'1001','A':'1010','B':'1011',
       'C':'1100','D':'1101','E':'1110','F':'1111',
       'a':'1010','b':'1011','c':'1100','d':'1101',
       'e':'1110','f':'1111'}   #lookup table for char->bin
    for pos in range(len(hex)):
        s+=t[hex[pos]]
    return s

# --UGLY GLOBAL CONSTANTS
#
outfilename='C:\ptov.out'       # file to write to
dump_name='C:\loaded.dd'        # physical memory dump
pf_name='C:\pagefile.sys.copy'  # pagefile
PDB   = '03b04000'              # physical address of PDB
start = '261ff0'                # VA to start reconstructing from (hex)
stop  = '262001'                # VA to reconstruct up to (hex)
                                # can handle up to 231 - 1

# --BEGIN...
#
dump = file(dump_name, 'rb')    # open files
pf   = file(pf_name, 'rb')      #...
out  = file(outfilename, 'wb')  #...

for pos in (xrange(int(start,16), int(stop,16), 1)):
                                  #for each byte...
                                  #   (could read as addressees using
                                  #   skip = 4)
    VA = hex(pos)                 #convert to hex
    pdibin = hextobin(VA)[0:10]   #split the VA...
    ptibin = hextobin(VA)[10:20]  #...
    bibin  = hextobin(VA)[20:32]  #...

    pdi = int(pdibin, 2)*4        #offset PDB->PDE
    pti = int(ptibin, 2)*4        #offset PDE->PTE
    bi  = int(bibin, 2)           #offset page->byte

    pd = int(PDB,16) + pdi        #phys. address of page directory
    dump.seek(pd)
    pde = '%0x' % struct.unpack('<I', dump.read(4))[0]
        #read the pde as a big-endian long int (32 bit)
    pdebin = hextobin(pde)        #bitmap of PDE
```

```
    #find page table source
    if pdebin[26:31]+pdebin[0:20] == '0'*21:
        pt_source = ''              #demand 0
    elif pdebin[21]+pdebin[31] == '10':
        pt_source = ''              #prototype PDE, ignore for now (FIX)
    elif pdebin[31] == '1' or pdebin[20:21] == '11':
        pt_source = dump            #table should be in the dump
    else:
        pt_source = pf              #table should be in the page-file

    #get page table
    if pt_source == '':             #pde is demand 0 or prototype
        ptebin = '0'*32             #return 0 page
    else:
        pt = (int(pdebin[0:20],2)*4096)+pti #phys. address of page table
        pt_source.seek(pt)
        pte = '%0x' % struct.unpack('<I', pt_source.read(4))
            #read the pte as a little-endian long int (32 bit)
        ptebin = hextobin(pte)              #bitmap of PTE
    offset = (int(ptebin[0:20],2)*4096)+bi  #offset from page to data

    #find page source
    if pt_source == '': #the pte was demand 0 or prototype (or invalid)
        char = '0'      #so return a 0 (change to '' to skip invalids)
    else:
        if ptebin[26:31]+ptebin[0:20] == '0'*21: #demand 0
            pg_source = ''                       #return 0 page
        elif ptebin[21]+ptebin[31] == '10':      #prototype PDE
            pg_source = ''                       #return 0 page
        elif ptebin[31] == '1' or pdebin[20:21] == '11':
            pg_source = dump                     #page should be in dump
        else:
            pg_source = pf                       #page should be in page-file

    #get data
    if pg_source == '':
        char = '0'          #change '0' to '' to not write invalid addresses
    else:
        pg_source.seek(offset)
        char = struct.unpack('<c', pg_source.read(1))[0]
                            #read char as little-endian 1 byte char
    out.write('%c' % char)  #write byte to file as char

out.close()     #close files
pf.close()      #...
dump.close()    #...
```