DIGITAL INVESTIGATION 7 (2010) S48–S56

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Windows operating systems agnostic memory analysis

James Okolica, Gilbert L. Peterson*

Department of Electrical and Computer Engineering, Air Force Institute of Technology, USA

Keywords:
Memory forensics
Microsoft windows
PDB files
Operating system discovery
Processes
Registry files
Network activity

ABSTRACT

Memory analysis is an integral part of any computer forensic investigation, providing access to volatile data not found on a drive image. While memory analysis has recently made significant progress, it is still hampered by hard-coded tools that cannot generalize beyond the specific operating system and version they were developed for. This paper proposes using the debug structures embedded in memory dumps and Microsoft's program database (PDB) files to create a flexible tool that takes an arbitrary memory dump from any of the family of Windows NT operating systems and extract process, configuration, and network activity information. The debug structures and PDB files are incorporated into a memory analysis tool and tested against dumps from 32-bit Windows XP with physical address extensions (PAE) enabled and disabled, 32-bit Windows Vista with PAE enabled, and 64-bit Windows 7 systems. The results show the analysis tool is able to identify and parse an arbitrary memory dump and extract process, registry, and network communication information.

1. Introduction

Memory analysis is an integral part of effective computer forensics. Since the DFRWS memory challenge in 2005 (Digital Forensics Research Workshop, 2005), there has been significant research done in improving analysis of memory dump files (Betz, 2005; Schuster, 2006b; Walters and Petroni, 2007). Unfortunately, these techniques still rely on knowing characteristics of the operating system a priori. Furthermore, in most cases, these tools only work on a small number of operating system versions. For instance, while Volatility has extensive functionality, it only works on Microsoft Windows XP SP2 and SP3. What is needed is a tool that works on an arbitrary memory dump regardless of the operating system version and patch level.

This paper is a first step in achieving this generalized functionality. By incorporating the work of Alex Ionescu and Microsoft's program database (PDB) files (Microsoft Support) into a memory analysis tool, the tool is able to identify the operating system and version of a memory dump from the family of Microsoft NT operating systems (i.e., Windows NT4, Windows 2000, Windows Server 2003, Windows XP, Windows Vista, Windows Server 2008, and Windows 7). The tool then uses this information to locate the kernel executable and extract its globally unique identifier (GUID). With the kernel name and GUID, the tool retrieves the PDB file from Microsoft's online symbol server and uses it to enumerate the key operating system structures necessary to parse the memory dump.

The remainder of this paper presents an overview of the memory analysis work already done and a methodology for combining these different pieces of memory analysis and parsing to make a Windows agnostic tool. Finally, the paper discusses applying the resulting tool to a memory dump from a 32-bit Windows XP SP3 with physical address extensions enabled and disabled, 32-bit Windows Vista with physical

* *Corresponding author*.
E-mail addresses: jokolica@afit.edu (J. Okolica), gpeterson@afit.edu (G.L. Peterson).
1742-2876/$ – see front matter
doi:10.1016/j.diin.2010.05.007

address extensions enabled, and 64-bit Windows 7. In each case, the tool identifies the operating system version and memory layout, extract all of the process and registry information (including pages stored in page files and memory backed files), and extract network communication information.

2. Background

Live forensics examines the most volatile, and generally the most recent cyber artifacts. Process activity, configuration changes, and network communication occur constantly and by examining volatile memory, the most recent instances of each of these are captured. Furthermore, the kernel executables residing on disk may not mirror the code actually running in memory (particularly if malware programs have hooked them). Examining the operating system programs that are in memory provides the most accurate picture of what the operating system is actually doing.

Live response information investigators typically seek include:

- system data and time,
- logged on users and their authorization credentials,
- network information, connections, and status
- process information, memory and process-to-port mappings
- clipboard contents
- command history
- services, driver information
- open files and registry keys as well as hard disk images (Prosise et al., 2003).

While ideally, the method for collecting memory should not affect the operating system, if no collection method has been implemented a priori, options are limited. In these cases, the best method may be to use software tools that will impact the operating system as a part of collecting the image. There are two distinct approaches: starting a new collection process (Carvey, 2007) or inserting a collection driver into an existing kernel process. The traditional software collection method is to start a new process, such as Madiant's Memoryze, that does not use operating system application programmer interfaces (APIs) or graphical user interfaces (GUI) so that it has less system impact and is less likely to be subverted by an infected operating system. However, creating a new process still creates new process records, object tables, and device tables as well as allocates space within a portion of main memory. The alternative is adding a driver to an existing kernel process. The downside of this method is that it modifies the space for one of the processes that will be captured. This may later call into question whether other, unintended changes were made to that process' space as well, possibly tainting the results.

There are several tools that parse memory dumps and extract process information. Two of the early tools that scanned memory dumps to find processes were Chris Betz's memparser (Betz, 2005) and Andreas Schuster's ptfinder (Schuster, 2006a). In addition, Brendan Dolan-Gavitt has developed tools for extracting Windows registry information (Dolan-Gavitt, 2008). More recently, Aaron Walters and others have developed Volatility (Walters and Petroni, 2007) which in addition to finding processes and registry information, also finds the network and configuration information. Furthermore, Volatility 1.3 parses hibernation files. However, what all of these tools have in common is that they are limited to specific versions of specific operating systems, e.g., 32-bit versions of Windows XP SP2 and SP3. The reason for this is that since the data structures used by an operating system change from version to version, new versions of the software are needed each time. However, Barbarosa and Ionescu have provided a means of discovering from within a memory dump, the operating system version that was running (Barbarosa; Ionescu). We combine this with Schreiber's method for analyzing the program database files (Microsoft Support) generated when Microsoft compiles its code (Schreiber, 2001a,b) to create a Windows agnostic memory analysis tool.

3. Methodology

By combining work done by (Barbarosa; Dolan-Gavitt, 2008; Ionescu; Russinovich and Solomon, 2005; Schreiber, 2001a; Schuster, 2006a; Walters and Petroni, 2007), it is possible to take an arbitrary memory dump from one of the Windows NT family of operating systems (i.e., Windows NT4, Windows 2000, Windows Server 2003, Windows XP, Windows Vista, Windows Server 2008, and Windows 7) and parse it. This Windows agnostic approach provides several benefits. First, memory analysis tools no longer need to be coded to a specific operating system version and patch level; second, memory dumps that are acquired without operating system interaction (e.g., via direct memory access) may be parsed without interacting with either the operating system or a system administrator. Finally, as new versions and patch levels of operating systems are released, the existing memory analysis tools should continue to work. Fig. 1 shows the Windows agnostic memory analysis process.

First, using the work of Barbarosa and Ionescu, _DBGKD_DEBUG_DATA_HEADER64, _KDDEBUGGER _DATA64 and _DBGKD_GET_VERSION64 records are found and parsed to determine whether the dump comes from a 32-bit, 32-bit with physical address extensions enabled, or a 64-bit operating system. Using this information (Russinovich and Solomon, 2005), the kernel page directory table base is found. With this information and (Russinovich and Solomon, 2005), virtual addresses are parsed into physical addresses. Next, the base address of the kernel executable and of tcpip.sys are found from _DBGKD_DEBUG_DATA_HEADER64 directly and via PS_LOAD ED_MODULE_LIST respectively. By examining the debug section of these two portable executables (Microsoft Windows Hardware Developer Central), the globally unique identifier (GUID) and age are extracted and used to download the correct program database from Microsoft's symbol server (Microsoft Support). The PDB file is then parsed (Schreiber, 2001a), and the exported kernel data structures are extracted. With these data structures, it is possible to parse the memory dump without any hard-coded offsets (although the names of the structures (e.g., _EPROCESS) do still need to be

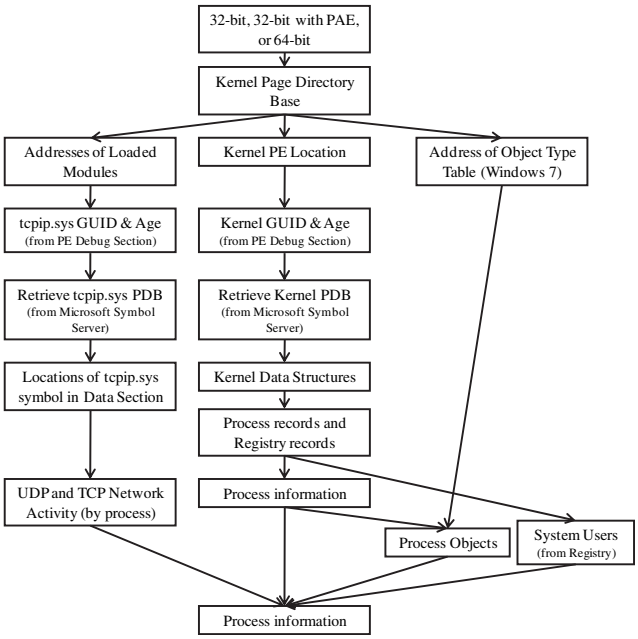

Fig. 1 – A Windows-agnostic memory analysis process flow.

hard-coded). Finally, using similar techniques, the location of the symbols in tcpip.sys is extracted from tcpip.pdb allowing the network communication data to be found within the tcpip.sys portable executable's data section. With all of this information, it is possible to extract process, registry, and configuration information using techniques discussed by Schuster (2006b), Dolan-Gavitt (2008), and Walters (Walters and Petroni, 2007) in the same way that other tools (e.g., Volatility (Walters and Petroni, 2007)) do.

The remainder of this section describes the constituent parts of the process in greater detail.

3.1. *Determining the operating system*

Memory parsing tools generally need to be given information about the memory dump (e.g., the processor type of the host machine, the operating system of the host machine (possibly including the specific service pack and patches installed), and whether physical address extensions are enabled). In Microsoft's family of Windows NT operating systems (i.e., Windows NT4, Windows 2000, Windows Server 2003, Windows XP, Windows Vista, Windows Server 2008, and Windows 7), this information is available in memory (Barbarosa; Ionescu). A key structure in the include files provided by Microsoft for developers of dynamic link libraries (DLLs) and debuggers is _DBGKD_GET_VERSION64 shown in Fig. 2.

When the kernel is running, this structure contains critical information including the base virtual address of the kernel's portable executable, a doubly linked list of the loaded modules, and whether physical address extensions are enabled. It also includes the major and minor operating system build numbers and the type of machine/processor (Microsoft Windows Hardware Developer Central). Either immediately preceding or immediately following this structure in memory is the _DBGKD_DEBUG_DATA_HEADER_64 which contains two fields, an owner tag which is the four-byte literal KDBG and the size of the _DBGKD_GET_VERSION64 structure and _KDDEBUGGER _DATA64 which contains among other things, the virtual address of all loaded modules and (in Windows 7) the location of the table of object type pointers. Our memory parser uses this information to scan a memory dump searching for KDBG followed by a four-byte field that is less than 4096 and then extracts the type of machine (i.e., 32-bit, 32-bit with PAE, or 64-bit). The operating system version, and the virtual addresses of the kernel executable, the list of loaded modules, and the object type table.

3.2. *Mapping virtual addresses to physical addresses*

The one value _DBGKD_GET_VERSION64 does not have is the kernel page directory table base, which is used to translate virtual addresses to physical ones. When the processor activates a process, it loads the process' page directory table base into the CR3 register and uses it to convert its virtual addresses into physical addresses. As a result, all other memory addresses used by the operating system are virtual addresses. Interestingly, although user address space is remapped by process, kernel address space is the same for all kernel processes. Therefore, finding any single page directory table base for a kernel process is sufficient to map any kernel process' virtual addresses to physical addresses.

Virtual addresses for i386 (32-bit processor) machines follows one of two formats depending on whether physical address extensions are disabled or enabled as shown in Fig. 3 (Russinovich and Solomon, 2005).

If physical address extensions are disabled, the highest ten bits are the index into an entry in the page directory table, a table composed of 32-bit words. Each entry in the page directory table points to a page table entry table. These page table entry tables are also composed of 32-bit words which

_DBGKD_DEBUG_DATA_HEADER64		
0x00	List_Entry64	List
0x10	ULong	OwnerTag
0x14	ULong	Size

_KDDEBUGGER_DATA64		
0x00	_DBGKD_DEBUG_DATA_HEADER64	Header
0x18	ULong64	KernBase
0x20	ULong64	BreakPointwithStatus
0x28	ULong64	SavedContext
0x30	UShort	ThCallBackStack
0x32	UShort	NextCallBack
0x34	UShort	FramePointer
0x36	UShort	PAEEnabled:1
…		
0x48	ULong64	PSLoadedModuleList
…		
0xA0	ULong64	OBTypeObjectType

_DBGKD_GET_VERSION64		
0x00	UShort	MajorVersion
0x02	UShort	MinorVersion
0x04	UChar	ProtocolVersion
0x05	UChar	KdSecondaryVersion
0x06	UShort	Flags
0x08	UShort	MachineType
0x0A	UChar	MaxPacketType
0x0B	UChar	MaxStateChange
0x0C	UChar	MaxManipulate
0x0D	UChar	Simulation
0x0E	UShort[]	Unused
0x10	UQuad	KernBase
0x18	UQuad	PsLoadedModuleList
0x20	UQuad	DebuggerDataList

Fig. 2 – MS Windows' debug structures.

point to specific pages in memory. When an i386 processor with physical address extensions disabled is provided with a virtual address, it starts with the page directory table base for the processor. It then takes the high ten bits of the virtual address, multiplies it by 4 (for 32-bit words) and uses that as the offset into the page directory table. At that location, it finds the physical address of the base of the relevant page table entry table. The processor then uses the next ten bits of the virtual address, multiplies it by 4 (for 32-bit words) and uses that as the offset into the page table entry table. At that location, it finds the physical address of the base of the relevant page of memory. It then uses the last 12 bits as an offset to find the specific location in physical memory. If physical address extensions are enabled, and additional level of indirection is introduced as shown in Fig. 3. In this case, the page directory table base is actual the physical address of a page directory pointer table. The high two bits of the virtual address are an offset into this page directory pointer table. The physical address in the page directory pointer table is then the base of a page directory table and the remaining process follows as above (though the indices are nine bits instead of ten). Physical address extensions change the page directory pointer table, the page directory table, and the page entry table from 32-bit words to 64-bit words (meaning indices are multiplied by eight instead of four). The ×64 architecture builds on these three levels, increasing the page directory pointer table to 512 entries and introducing a fourth level of indirection, called the page map levels as shown in Fig. 3. In the ×64 (and IA32-E) virtual memory model, the first nine bits are the page map level table index, followed by nine bits for the page directory pointer table index, followed by nine bits for the page directory table index, followed by nine bits for the page table entry table index, followed by twelve bits for the physical page offset. In each of these four cases, if the large page flag is set in the page directory entry, there is one less level of indirection as the page table entry and the page frame offset are combined to generate an offset into either a two (21 bit offset) or four megabyte page (22 bit offset).

To find the page directory bases used above, the self-referencing nature of the page directories shown in Fig. 4 is used. For instance, in 32-bit no PAE operating systems, the page directory entry that is 0xC00 from its page directory base points to the page directory base. In the case of PAE enabled, the first two (possibly three) entries of the page directory pointer table point to user space while the last entry is guaranteed to point to kernel space. As a result, the fourth entry in the page directory pointer table is the physical address of the first entry. Finally, the 64-bit pointer offset 0x68 from the page map level table base points back to the page map level table base address. While these observations are not guaranteed to work, heuristically starting at the beginning of physical memory and proceeding until the appropriate condition is found results in the page directory table base of a kernel process.

Ideally, all referenced memory exists as physical addresses in the memory dump; however, this is often not the case. The low twelve bits of the page table entries shown in Fig. 5 are flags that indicate different items of interest. One particular flag of interest is bit 0. If bit 0 is 0 than the physical page of memory referenced is invalid. This may mean that the page frame is transitioning from memory to disk (if bit 11 is set) or that the page frame is a prototype page (if bit 10 is set) or that the page frame has been paged out to disk (if bits 0, 10, and 11 are all zero).

If the page frame has been paged to disk, then the page table entry needs to be interpreted differently as shown in Fig. 6. Bits 1 through 4 determine which page file the page frame is in (up to 16 page files are possible) and bits 12 through 32 determine the offset into the page file. If both the page file number and offset are zero, than the page table entry is referencing a "demand zero" page, i.e., a page that has been allocated and filled with zeros but which has yet to have any information stored in it.

In addition to paged to disk page table entries, there are also page table entries that are prototype page table entries. Prototype pages are pages that can be shared between two or more processes (e.g., memory that contains configuration information). Prototype page table entries have their own format shown in Fig. 7.

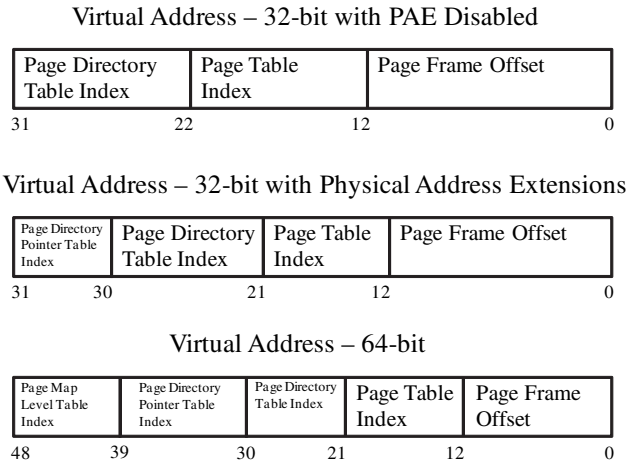

Fig. 3 – Virtual address to physical address translation.

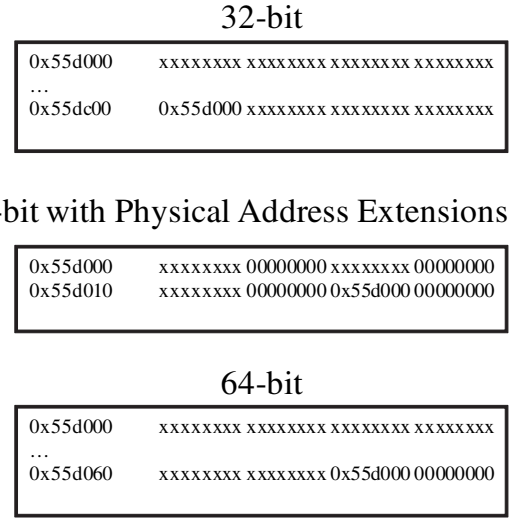

Fig. 4 – Self-referencing nature of directory table bases.

On a 32-bit operating system with PAE disabled, the prototype page table entry is contained within bits 1 through 7 (0–6) and 11 through 31 (7–27) of the page table entry and is an offset into the non-paged pool area (located at virtual address 0x*e*1000000 in 32-bit operating systems). On a 32-bit operating system with PAE enabled, the high 32 bits contain the virtual address of the page frame. Finally, if the prototype page table entry itself is invalid, then the prototype page is actually a file-backed page (i.e., a memory-mapped file). In this case, the actual memory is stored within a file stored in non-volatile memory. In the case of memory-mapped files, the prototype page table entry is actually a pointer to a subsection. Stored within the control area of the subsection, is a pointer to the file object. To determine the base offset into the memory-mapped file, the subsection base is subtracted from the prototype page table entry then multiplied by either 4 or 8 (depending on if PAE is disabled or enabled) and then added to the starting sector of the subsection. The low 12 bits of the virtual address are then added to the base offset (multiplied by 4096 to account for the page size).

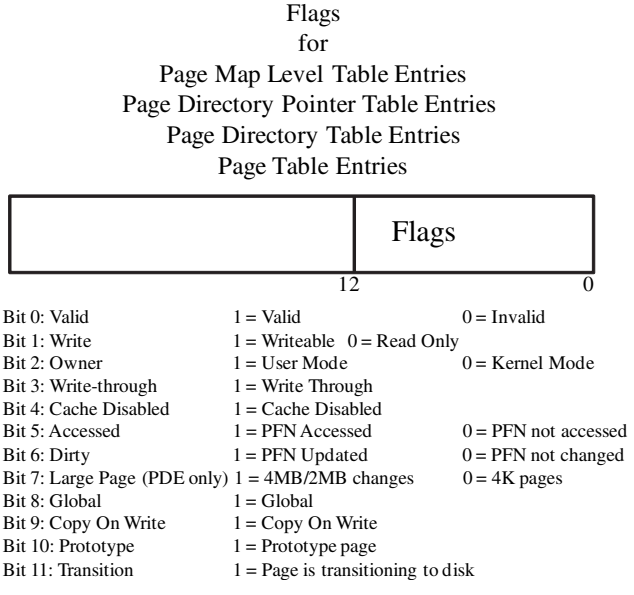

Fig. 5 – Flags for page map level, page directory pointer, page directory, and page table entries.

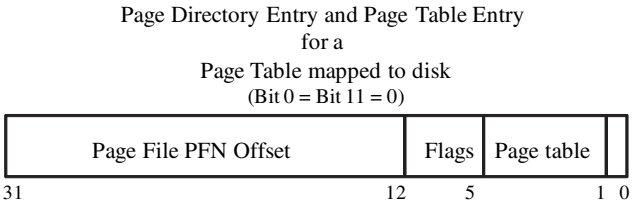

Fig. 6 – Page table entry layout for frames paged to disk.

3.3. *Operating system structures*

Once a page directory table base (or equivalent) is found, the last item necessary to determine the operating system version is to retrieve the major and minor operating system version from the kernel's portable executable. Recall that the virtual address of the kernel's portable executable is stored in the _DBGKD_GET_VERSION64 structure. Now that a page directory table base is known for kernel space, this virtual address is converted to a physical address and the major and minor version of the operating system stored in the kernel's portable executable (Microsoft Windows Hardware Developer Central) retrieved. In addition, stored within the debug section of the kernel's portable executable is its globally unique identifier (GUID). This GUID provides the key to improving the versatility of memory parsers.

In general, memory parsers have operating system structures hard coded for the specific operating system versions/patch levels that they handle. Unfortunately, as new version/patches come out, hard-coded parsing tools may become obsolete. As part of Microsoft's compilation and linking process, Microsoft records debug information in a program database (PDB) file (Microsoft Support) including the exported structures used by the executable. Whenever Microsoft releases a patch (or operating system version), it places a copy of the PDBs for any changed executables on its symbol server. While the primary purpose of this is for use by Microsoft's own debugging tools (e.g., windbg and kdbg), these PDB files contain information that can be used by any debugging or

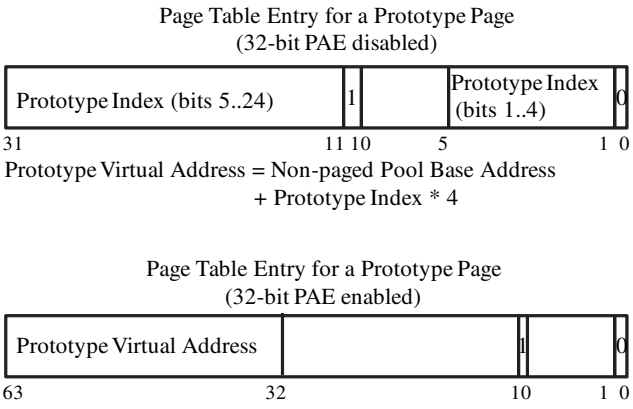

Fig. 7 – Page table entry layout for prototype pages.

memory parsing tool. To retrieve these files, a GET request is sent to the Microsoft Symbol Server with the name of the executable, the GUID and the name of the compressed program database file. Although Microsoft does not release the format of PDB files, in 2001 Schreiner provided some insight into its structure (Schreiber, 2001a). In addition, the PDB file format in use in 2001 (Schreiber, 2001a) is still valid suggesting that the PDB file format is very stable.

PDB files are structured like file systems. First, there is a root "stream" that is an index of all of the streams contained in the PDB file. Second, the file is divided into 400 byte "blocks" and streams may span multiple non-contiguous blocks. Finally, there are blocks of "obsolete" data in the file where a block had been previously allocated to a stream and then subsequently "deleted."

The PDB file begins with the PDB file header shown in Fig. 8. The first field of interest, dPageSize, tells how long each block of text is (generally 0x400). The next field of interest, RootStream is a "PDB_STREAM" data structure composed of two parts, the first being the size of the stream. Finally, the last field in the header, awRootPages, is the stream pointer (i.e., the index number of the block) that contains a list of the stream pointers containing the root stream. For instance, awRootPages may be 0x30. In that case at byte 0xC400 (0x30 × 0x400), there would be a list of stream pointers (e.g., 0x2*d*, 0x2*e*, 0x2*f*) describing where the root stream is. The root stream itself (in the above example, located at block 0x2*d*, byte 0xB400) begins with a 32-bit word defining the number of streams contained in the PDB file. This is followed by a 32-bit word defining the number of blocks contained in each stream (possibly some of these are 0 block streams). For instance, if there are seven streams in the PDB file, the root stream might begin with 0x07 0x02 0x10 0x0 0x01 0x14 0x0*a* 0x11. Immediately following the number of streams and the size (in blocks) of each stream is a list of the blocks containing each stream. In the above example, the two 32-bit words following the list would be the blocks containing stream 0; the sixteen 32-bit words following these would be the blocks containing stream 1; and the next 32-bit word would be the block containing stream 3 (since the size of stream 2 is zero).

While parsing the PDB file into streams is relatively straightforward, determining the format and purpose of the individual streams is less so. It does not appear that a specific stream number always perform the same function. However, there are a few heuristics that produce good results. For instance, the "section" stream (i.e., the stream that describes the sections in the associated portable executable) seems to always begins with the either the literal ".data" or ".text". The "structure" stream (i.e., the stream containing information about the data structures used by the associated portable executable) seems to always begin with 0x38. In addition, the "symbol" stream (i.e., the stream containing the symbols used by the portable executable) is made up of records that start with a two byte record size (safe to assume a value of less than 0x100) and a two byte literal of 0x110E. Finally, the "symbol location" stream (i.e., the stream that adjusts the location of symbols in the data section) seems to always immediately follow the section stream. While the format of the individual streams is not available, by comparing the streams with known values, some observations are made. Specifically, the records in the symbol stream shown in Fig. 9 are made up of a variable record size followed by six unknown bytes of data, followed by the 32-bit unadjusted offset of the symbol in the data section, followed by the 16-bit type of symbol, followed by the name of the symbol.

The records in the sections stream in Fig. 9 are made up of an 8-byte section name, followed by the 32-bit virtual size of the section, followed by the 32-bit virtual address of the section, followed by several more fields described in Microsoft's Portable Executable document (Microsoft Windows Hardware Developer Central). The records in the symbol location stream are a collection of 32-bit word pairs where the first word is the unadjusted offset and the second word is the adjusted offset.

Unlike the streams described above, parsing the structures stream is less straightforward. Each record in the structures

PDB Header Structures

```
#define PDB_SIGNATURE_200 \
        "Microsoft C/C++ program database 2.00\r\n\x1AJG\0"
#define PDB_SIGNATURE_TEXT 40

typedef struct _PDB_SIGNATURE {
        BYTE abSignature [PDB_SIGNATURE_TEXT+4];
} PDB_SIGNATURE;

typedef struct _PDB_STREAM {
        DWORD dStreamSize;          // in bytes, -1 = free stream
        PWORD pwStreamPages;        // array of page numbers
} PDB_STREAM,;

typedef struct _PDB_HEADER {
        PDB_SIGNATURE Signature; // PDB_SIGNATURE_200
        DWORD dPageSize;            // 0x0400, 0x0800, 0x1000
        WORD wStartPage;            // 0x0009, 0x0005, 0x0002
        WORD wFilePages;            // file size / dPageSize
        PDB_STREAM RootStream;  // stream directory
        WORD awRootPages [];      // pages containing PDB_ROOT
} PDB_HEADER;

typedef struct _PDB_ROOT {
        WORD wCount; // < PDB_STREAM_MAX
        WORD wReserved; // 0
        PDB_STREAM aStreams []; // stream #0 reserved for stream table
} PDB_ROOT;
```

Fig. 8 – PDB file header.

Structure Stream

0x00	UShort	Record Size
0x02	STRUCTURE_RECORD	

Symbol Stream

0x00	UShort	Record Size
0x02	UShort	Unknown1
0x04	ULong	Unknown2
0x08	ULong	Offset
0x0C	UShort	Type
0x0E	*Char	Symbol Name

Symbol Relocation Stream

0x00	ULong	Relocation Address
0x04	Ulong	Data Address

Section Stream

0x00	*Char	Name (not null terminated)
0x08	ULong	Virtual Size
0x0C	ULong	Virtual Address
0x10	ULong	Raw Size
0x14	ULong	Raw Pointer
0x18	ULong	Relocation Pointer
0x1C	ULong	Line Pointer
0x20	UShort	Relocation Count
0x22	UShort	Line Count
0x24	ULong	Characteristics

Fig. 9 – PDB stream structures.

stream shown in Fig. 10 is a "field" with an index that begins at zero and proceeds sequentially. This index is important because later fields make reference to earlier fields via the field index. Each field begins with a record size. This is followed by a data type [source]. The one oddity is if the offset for LF_MEMBER is 0x8004 or the size for LF_UNION, LF_STRUCTURE or LF_ARRAY is 0x8004, or the value for LF_ENUMERATE is 0x8004, then an additional 32 bits are added to the size of these structures and the next 32 bits are the offset, size, or value respectively. Additionally, LF_STRUCTURE, LF_UNION, and LF_ENUM are often defined twice, once as a placeholder (e.g., to handle self-referencing) and once with a complete structure. In this case, the first definition is a shell with a record size of zero and the second definition contains all of the relevant information including the name.

3.4. *Finding and instantiating processes in memory*

Once the memory model and the operating system data structures are known (by extracting them from the kernel's PDB file), they are used to parse the memory dump and extract configuration and process information. For instance, process information is stored in the _EPROCESS structure. The first field in _EPROCESS is itself a structure called _KPROCESS. Within _KPROCESS, the first field is another structure called _DISPATCHER_HEADER. Two fields present in the _DISPATCHER_HEADER are a two byte tag field and a two byte size field providing the size of _KPROCESS in 32-bit words (representing the size in 32-bit words is true regardless of whether the operating system is 32-bit or 64-bit). With this information, it is possible to create a signature for processes (Schuster, 2006b). By using the structures from the PDB file, the size of KPROCESS is calculated. Further, for all operating system versions through Windows 7, the type for a process is 0x03 (determined empirically through Microsoft's kernel debugger). With this information, a _DISPATCHER_HEADER template is created and populated using the above values and the structures from the PDB file. This template may then be used as a "process signature" for scanning memory with. Once the _EPROCESS structure is found, instantiating it is straightforward using the structures in the PDB file. For instance, the page directory base is stored in the _KPROCESS structure within the _EPROCESS record while a pointer to the object table is located directly in the _EPROCESS structure. Although the location of the fields within these structures is found at run-time using the PDB file, the names of these fields, e.g., _EPROCESS, is hard-coded and is assumed to remain static across all versions of the operating system (the one known exception to this is the change in Windows 7 from a pointer to _OBJECT_TYPE in the _OBJECT_HEADER field to having an index into the obObjectTypeTable). Other fields of interest include the amount of kernel and user time the process has consumed, the time the process was created, the name and unique id of the process, the process' token (which relates back to the user who created the process), the priority of the process, and the number of read, write, and other operations the process has performed. A final item of interest is the process environment block which contains, among other items, the loader table for all modules loaded by the process, the parameters the process was started with, as well as the operating system version and number of processors.

```
LF_FIELDLIST
   list of fields
LF_STRUCTURE
   UShort    Element Count
   UShort    Properties
   ULong     Field Index
   ULong     Derived
   ULong     Vshape
   UShort    Size
   Char*     Name
LF_POINTER
   ULong     Underlying Type
   ULong     Pointer Array
LF_MEMBER
   UShort    Properties
   ULong     Underlying Type
   UShort    Offset
   Char*     Name
LF_UNION
   UShort    Element Count
   UShort    Properties
   ULong     Field Index
   UShort    Size
   Char*     Name
LF_ARGLIST
   ULong     Element Count
   ULong[]   Arguments
LF_PROCEDURE
   ULong     Return Value Type
   UChar     Call Type
   UChar     Unknown
   UShort    Element Count
   ULong     Field Index
LF_ENUM
   UShort    Element Count
   UShort    Properties
   ULong     Underlying Type
   ULong     Field Index
   Char*     Name
LF_ENUMERATE
   UShort    Properties
   UShort    Value
   Char*     Name
LF_ARRAY
   ULong     Underlying Type
   ULong     Index Type
   UShort    Size
   UShort    Unknown
LF_BITFIELD
   ULong     Underlying Type
   UChar     Size
   UChar     Offset
   UShort    Unknown
```

Fig. 10 – PDB structures.

3.5. *Finding and instantiating configuration information in memory*

While finding configuration management "hives" (i.e., registry entries) in memory is also done with a signature key, the instantiation is less straightforward (Dolan-Gavitt, 2008). The configuration manager is composed of several "hives" with each hive having a specific purpose. For instance, under Windows XP SP2, there are hives for the NTUser, UsrClass, currently logged in user, LocalService user, NetworkService user, template user ("default"), Security Account Manager (SAM), SYSTEM, SECURITY, SOFTWARE, and two volatile hives that have no on-disk representation (HARDWARE (hardware installed on the particular machine) and REGISTRY (a header hive that provides a unified namespace)) (Dolan-Gavitt, 2008). Each of these configuration management hives (_CMHIVE) has a field named signature with a value of 0x*bee0bee*0. With the signature, it is possible to find potential configuration manager hives in memory (and then remove any false positives by examining their structures). Once these hives are found, they still need to be parsed. Hives are broken down into fixed length 0x1000 byte bins with variable lengths cells within them. The cells are generally of one of two types: key nodes and value nodes. Key nodes provide the directory structure while value nodes provide the values for the configuration keys. References to cells in a bin are made using a cell index. The high bit of the cell index indicates whether the cell index's main storage is stable (on-disk) or volatile (only in memory). The next 10 bits are the directory index and work the hive's map to point to a hive table. Bits 12–20 are the table index and provide an offset into the table found using the directory index. Finally, the low 12 bits are the cell offset for the hive table found previously (Dolan-Gavitt, 2008).

Key nodes have a name followed by either values for the count of subkeys and a cell index containing the LF records (an

array of cell indices and subkey abbreviations) or a list of values (which also is a count followed by cell indices). Consider, for example, translating a process token into a user name. First the SOFTWARE hive is located. Then the key node tree is traversed to find the value list for Microsoft/Windows NT/Current Version/Profile List. At each level (starting from the root index), a location of the sublist is found. Each entry in the sublist points to a key node with a name. From the SOFTWARE hive, the first sublist is traversed until a key node with the name Microsoft is found. Then the sublist at that key node is traversed until Windows NT is found. This is repeated until the Profile List key node is found. At this point, the sublist is traversed looking for a name that matches a string representation of the token. Once this key node is found, the sublist is traversed to find the ProfileImagePath key node. This key node has a value list with (in this case) a single value for the home directory of that token. In general the final subdirectory of that path should correspond to the name of the user who started the process.

3.6. *Finding and instantiating network activity in memory*

In addition to containing the data structures of the executables, the PDB files also contain the symbols used by an executable. This is particularly important since the Windows operating system kernel does not directly handle communication. Windows uses tcpip.sys (a portable executable) to handle its TCP/IP communication. Furthermore, the data structures (i.e., symbols) used to store the communication activity are not exported. Instead, the names and structures of the relevant data structures must be determined via reverse engineering. For instance, in Windows XP, the symbol _AddrObjTable is a table of process IDs and TCP connections while _TCBTable is a table of process IDs and UDP connections. The structure of these symbols (along with the symbols for their sizes, _AddrObjTableSize and _MaxHashTableSize respectively) must be found by manually examining the tcpip.sys portable executable.

Once these symbols are known, the memory analysis tool uses the PDB file for tcpip.sys to find the location of these symbols within the tcpip.sys executable resident in the memory dump. Within the PDB file, there are two streams used to calculate the location of the symbols. The first, provides a list of the symbols along with their offsets. The second, is an adjustment for these offsets. This second stream is a table where each entry has two values. The first is a virtual address (relative to the image base address) and the second is the sum of the offset and the relative virtual address of the data section (found in the section stream). Once the adjusted locations of these symbols within the portable executable resident in memory is known, the memory tool extracts the local and remote socket addresses as well as the process they are associated with.

4. Analysis of results

The analysis tool developed following the process in Fig. 1 parses a memory dump (with associated page files and potentially memory-mapped files) from a Windows NT family of operating systems to provide information on user accounts, the Windows Registry, and running processes. The tool outputs system, process, registry, and user information in a standalone tool that runs without API calls or high level language interpreters.

To test the tool's functionality, memory dumps are generated from 32-bit Windows XP with PAE enabled and disabled, from 32-bit Windows Vista with PAE enabled, and from 64-bit Windows 7. The output is then compared with output from Microsoft's netstat and SysInternals psinfo, pslist, logonsessions, handles, and listdlls utilities (Russinovich). The system information examined includes operating system version, number of processors, and number of processes. The process information examined includes process creator, files opened, registry keys accessed, modules loaded, and network activity. Several application programs are started on the machine including Internet Explorer, Word, PowerPoint, Visual Studio, Calculator, Kernel Debugger, and two command line shells. One of the command line shells is hidden by the FUTo rootkit (Silberman, 2006).

Test results demonstrate that the memory analysis tool provides the same or equivalent information to the information provided by the SysInternal utilities. In addition, in the case of Windows XP, the Windows-agnostic tool provides the same information provided by Volatility (since Volatility is limited to Windows XP, it could not be tested on Windows Vista or Windows 7). In all cases, the operating system version, processor count, process count, user IDs, loaded modules, files, registry keys, and network activity matched.

While these tests were performed only on a subset of the Windows NT operating systems, it should be straightforward to extend the Windows-agnostic tool to all Windows NT operating systems. There is, however, one issue. Since the names of structures are still hard-coded, any changes to the names of the variables would require changing the memory analysis tool. For instance, the memory analysis tool assumes there is a structure called _EPROCESS that has process information and that there is a structure contained within _EPROCESS called DirectoryTableBase that contains the page directory base for a process. If in a future version of a Windows operating system, Microsoft changes the names of the structures, the tool will have no way of knowing what the new names are. In fact, this did occur when Microsoft released Windows 7 and changed the method of associating an object type with an object from a pointer within the _OBJECT_HEADER record to an index into the Object Type Array pointed to by obpObjectTypeTable.

Data structure names have also changed in each operating system release of tcpip.sys. While in the case of kernel objects, the new structure names are generally published making any required coding changes straightforward, this is not the case for tcpip.sys. Changes in tcpip.sys are not publically available and can only be found by reverse engineering the new system module. This means that extracting network activity across operating systems continues to require manual operating system specific coding.

5. Conclusions and future work

By incorporating the debug structures and PDB files, memory analysis tools can handle a much larger range of operating

systems. Tests performed on 32-bit Windows XP with and without physical address extensions, 32-bit Windows Vista with physical address extensions enabled and 64-bit Windows 7 show that Windows agnostic memory analysis tools can provide the same level of detail as the current state of Windows specific memory analysis tools with one exception. Given the degree that the data structures in tcpip.sys change from operating system version to operating system version, memory analysis tools need to remain operating system specific for network communications.

While greater understanding of the PDB file structures is needed, these same techniques should extend to searching for malware in the portable executables found in memory dumps. With this additional functionality, even memory-resident malware becomes visible to the forensic investigator. This provides access to a two new classes of malware: (1) malware that is downloaded only after a stub is executed and (2) malware that is packed (encrypted) and only unpacks (decrypts) when it is loaded into memory.

Finally, there is no reason that these techniques have to be limited to memory dumps. By incorporating them either into system modules or into an underlying hypervisor, these tools can function as sensors for intrusion detection systems.

REFERENCES

Barbarosa E. (Opcode), Finding some non-exported kernel variables in Windows XP, http://www.rootkit.com/vault/Opc0de/GetVarXP.pdf.

Betz C. memparser, http://sourceforge.net/projects/memparser; 2005.

Carvey H. Windows forensic analysis. Syngress; 2007.

Digital Forensics Research Workshop. DFRWS 2005 forensic challenge – memory analysis, http://www.dfrws.org/2005/challenge/index.shtml; 2005. Accessed February 19, 2010.

Dolan-Gavitt B. Forensic analysis of the windows registry in memory. In: Proceedings of the 2008 Digital Forensic Research Workshop (DFRWS); 2008. p. 26–32.

Ionescu A. Getting Kernel variables from KdVersionBlock, Part2, http://www.rootkit.com/newsread.php?newsid=153.

Mandiant. Memoryze, http://www.mandiant.com/software/memoryze.htm. Accessed August 15, 2009.

Microsoft Support. Description of .PDB and of the .DBG files, http://support.microsoft.com/kb/121366.

Microsoft Windows Hardware Developer Central. Microsoft portable executable and common object file format specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

Prosise C, Mandia K, Pepe M. Incident response & computer forensics. 2nd ed. McGraw-Hill/Osborne; 2003.

Russinovich M, Solomon D. Microsoft Windows internals. 4th ed. Microsoft Press; 2005.

Russinovich M. SysInternals suite, http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx. Accessed August 15, 2009.

Schreiber S. Undocumented Windows 2000 secrets: a programmer's cookbook. Addison Wesley, http://www.informit.com/articles/article.aspx?p=22685; 2001a.

Schreiber S. Undocumented Windows 2000 secrets: a programmer's cookbook. Addison Wesley, http://undocumented.rawol.com/; 2001b.

Schuster A. PTfinder, http://computer.forensikblog.de/en/2006/03/ptfinder_0_2_00.html; March 2, 2006a.

Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. In: Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS); 2006b. p. 10–6.

Silberman P. FUTo, http://www.uninformed.org/?v=3&a=7&t=sumry; Jan, 2006.

Walters A, Petroni N. Volatools: integrating volatile memory forensics into the digital investigation process. Blackhat Hat DC 2007, www.blackhat.com/presentations/bh-dc./bh-dc-07-Walters-WP.pdf; 2007.