Searching for Processes and Threads in Microsoft Windows Memory Dumps

By

Andreas Schuster

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2006 USA
Lafayette, IN (Aug 14th - 16th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http://dfrws.org
Searching for processes and threads in Microsoft Windows memory dumps

Andreas Schuster
Deutsche Telekom AG, Friedrich-Ebert-Allee 140, D-53113 Bonn, Germany

Abstract

Current tools to analyze memory dumps of systems running Microsoft Windows usually build on the concept of enumerating lists maintained by the kernel to keep track of processes, threads and other objects. Therefore they will frequently fail to detect objects that are already terminated or which have been hidden by Direct Kernel Object Manipulation techniques.

This article analyzes the in-memory structures which represent processes and threads. It develops search patterns which will then be used to scan the whole memory dump for traces of said objects, independent from the aforementioned lists. As demonstrated by a proof-of-concept implementation this approach could reveal hidden and terminated processes and threads, under some circumstances even after the system under examination has been rebooted.

Keywords: Digital evidence, Forensic examination, Microsoft Windows, Volatile data, Incident postmortem

1. Introduction

The physical memory of a computer running Microsoft Windows 2000 or one of its descendants contains all meta-information necessary to manage the processes that are currently executed. As Chow, Pfaff, Garfinkel and Rosenblum showed, such meta-information in kernel memory can survive periods over 14 days and longer while the system is in use (Chow et al., 2005). Despite its volatile nature kernel memory thus is a useful information source in a forensic examination.

Several methods exist to dump the physical memory’s contents to a file. Carrier and Grand (2004) provide a comprehensive description. On the Microsoft Windows platform there are two methods commonly used. Copying \Device\Physical Memory to a file with the help of dd (Garner, 2004; Nicholas Harbour, 2005) is still very popular and recommended even in the newer literature (Brown, 2005, p. 223f.). Main benefit of this method is the simple format of the resulting file: the file offset equals the absolute address in physical memory. For security reasons access to physical memory is barred from the userland from Windows Server 2003 Service Pack 1 onwards (Microsoft Corporation, March 2005).

Another popular method uses a documented registry key setting to cause a BugCheck trap on activation by a certain key sequence (Microsoft Corporation, August 2005). The BugCheck in turn causes the creation of a memory dump as configured in the system’s settings. The main disadvantage of this method is the complex format of the resulting dump which might even leave out some pages of physical memory (Schuster, 2006a). On the other hand only this format enables analysis with Microsoft’s debuggers which are quite helpful when dealing with kernel data structures.

1.1. Related work

In 2005 three programs appeared which analyze full memory dumps of systems running Microsoft Windows. Mariusz Burdach described a procedure to enumerate processes and modules and implemented it in his Windows Memory Forensics Toolkit (Burdach, 2005). Chris Betz programmed MemParser,
The kernel defines classes for processes and threads which most commonly is Intel’s IA-32 CPU architecture. According to this architecture memory is organized in pages of 4096 bytes. \(^1\) For requests of memory smaller than or equal to the page size a part of the kernel called the Memory Manager will try to find a properly sized free region within the requested pool. Such small allocations will never extend across a page boundary. If no sufficient free space is available, the Memory Manager will claim another page of memory, add it to the proper pool and assign the requested amount of memory from that page. Within a page the allocated blocks are loosely chained. Each block stores its own size and the size of the previous block. This allows to traverse the list of blocks in both directions.

Rule 1 There must be enough space preceding the current block to fit in the previous block.

Rule 2 From the start of the assumed block there must be enough space left in the current page to fit in the block.

Memory will be assigned in chunks of 32 bytes on Windows 2000 and 8 bytes on later versions. Therefore BlockSize and PreviousSize had to be extended to a width of 9 bits in transition from Windows 2000 to XP as shown in Fig. 1.

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

The PoolType is declared in the Microsoft Windows Driver Kit in the wdm.h and ntddk.h header files and documented in the Microsoft Developer Network (Microsoft Corporation, May 2005). It has to be noted that the type code stored in the POOL_HEADER is incremented by 1. Hence a value of 0x00 marks a freed allocation.

\(^1\) Large pages of 4 MB are also supported. Versions up to and including Windows 2003 employ large pages only to store the kernel binary in memory.
Rule 4 PoolType must be either free or of a non-paged class, that is \( \text{PoolType} = 0 \) or \( \text{(PoolType - 1)} \% 2 = 0 \).

It is recommended to attach a tag of four ASCII characters to each memory request. The tag should be unique for each requester, e.g., a driver. A requester might use several tags to differentiate pool usage between its routines. The tag will be stored in the pool header. Tagging allows to track back memory leaks and pool corruption to the offending driver.

Process and thread objects will be created through a call to nt!ObpAllocateObject. This function allocates the required amount of memory by calling nt!ExAllocatePoolWithTag. The pool tag will be taken from the proper OBJECT_TYPE structure, whereas the most significant bit will be set. As pool tags usually are limited to contain ASCII characters only, this might be an attempt to protect\(^2\) tags of the operating system’s objects from accidental use by third-party code. The keys for process and thread objects are “Proc” and “Thre”, hence the “protected” tags are:

Rule 5 PoolTag = 0xe36f7250 for processes. This rule does not apply to the idle process.

Rule 6 PoolTag = 0xe5726854 for threads. This rule does not apply to the idle thread.

2.2.2. OBJECT_HEADER

Each of the kernel’s objects is prefixed by an OBJECT_HEADER as shown in Fig. 2. Type points to an OBJECT_TYPE structure (see Fig. 3). This structure defines the object’s class. So obviously all instances of the same class will refer to the same OBJECT_TYPE structure.

So the Type member of the OBJECT_HEADER may be used to identify an object in memory. This instantaneously raises some new questions: What are the proper values for process and thread objects? What factors do they depend on? And how can they be derived from a memory dump?

The kernel binary exports the names of global variables containing pointers to OBJECT_TYPE structures for processes and threads. Their names are PsProcessType and PsThreadType. An examiner now could locate the kernel in the memory dump. From there he could read the export table, find the symbols mentioned before and retrieve the type pointer values.

In a similar way the offsets could be determined and tabulated for known kernel versions. Again starting at the kernel’s location in the dump file the examiner than would have to find out the version of the kernel and look up the proper offset from the table. Next he will have to add the offsets to the kernel’s position and finally retrieve the type pointer values.

However, it would not be possible to speed up the process and tabulate the pointer values themselves. Some experiments with Windows 2000 in a VMware environment have shown that the values remain constant between reboots of the same system configuration. Changing the amount of total memory available to the system affects the way Windows partitions the memory and as such causes the OBJECT_TYPE structures to be created at different addresses. The same might happen due to significant changes of the operating system’s configuration, e.g., the installation of a driver.

It would also be possible to apply the concept of searching for those two OBJECT_TYPE structures, too. The rules are:

Rule 7 \( \text{Name.Length} = 0x0e \) and \( \text{Name.MaximumLength} = 0x10 \) and \( \text{Key} = 0x636f7250 \) for the processes type.

Rule 8 \( \text{Name.Length} = 0x0e \) and \( \text{Name.MaximumLength} = 0x10 \) and \( \text{Key} = 0x65726854 \) for the thread type.

At this Length and MaximumLength are two unsigned short integers which designate the length of a UNICODE string in bytes.

There is also a third pointer value which is of great importance in forensic examinations. Whenever an object, not necessarily a process or thread, is destroyed the kernel’s nt!ObpFreeObject function sets its Type pointer to 0xbad0b0b0. The value remains constant over all kernel versions from Windows 2000 up to and including Server 2003.

Now, that the type pointer values are known, it becomes possible for them to use them for identifying process and thread structures:

Rule 9 (Type = PsProcessType) or (Type = 0xbad0b0b0) for processes.

Rule 10 (Type = PsThreadType) or (Type = 0xbad0b0b0) for threads.

Fig. 1 – Definitions of the POOL_HEADER structure in Windows 2000 (left) and later versions (right).

Fig. 2 – The OBJECT_HEADER structure provides information about an object’s instance.
2.2.3. DISPATCHER_HEADER

Processes and threads are synchronizable objects. As such their control structures EPROCESS and ETHREAD begin with a substructure known as DISPATCHER_HEADER (see Fig. 4).

The header contains a Type field which allows to differentiate between these object types easily. The type code for an object class is constant over the versions from Windows 2000 to Server 2003 (see Table 1).

For a given version of Windows the Size is constant for all objects of a particular kind. It tells the object’s size in units of DWORDs, that is 4 bytes. During the creation of a process (nt!KeInitializeProcess) or thread object (nt!KeInitThread), the kernel initializes Type and Size with hard-coded values. These values will not change during the object’s lifespan.

The meaning of Inserted and Absolute are unknown to the author. Based on the analysis of memory dumps obtained from several systems running different versions of Microsoft Windows from 2000 to Server 2003 both fields are always null. However, a code review of the kernel would be necessary to confirm this observation. At present it is not recommended to build a filter expression upon these two fields.

2.3. Additional checks

2.3.1. Processes

A process is represented by an EPROCESS structure. This structure varies with the version of Windows. Key-values for the rules mentioned below are given in Table 2.

The operating system provides any process with a virtual address space of its own. Some nested tables are used to map virtual addresses to the proper page frame in physical memory. DirectoryTableBase points to the beginning of the necessary structures. The Page Directory occupies a whole memory page; it is aligned at a page boundary.

- Rule 11 PageDirectoryTable! = 0
- Rule 12 PageDirectoryTable % 4096 = = 0

Every process needs at least a single thread to do the work. Control structures of threads are kept in a doubly-linked list. EPROCESS contains two pointers into this list: ThreadListHead.Flink and ThreadListHead.Blink. The control blocks pointed at are located in kernel space. Therefore their virtual address must be above 0x7fffffff.3

- Rule 13 (ThreadListHead.Flink > 0x7fffffff) and (ThreadListHead.Blink > 0x7fffffff)

- Rule 14 The structure must start with a DISPATCHER_HEADER of type 3 (process).

- Rule 15 The structure must contain a Synchronization Event # 1 at the position designated in Table 2. This rule does not apply to the Idle Process (PID 0).

- Rule 16 The structure must contain Synchronization Events # 2 and 3 at the positions designated in Table 2.

2.3.2. Threads

A thread is represented by an ETHREAD structure. Table 3 provides the offsets of variables used in the tests mentioned below.

The variable ThreadsProcess points to the EPROCESS structure of the owning process. This structure has to be located in kernel memory.

- Rule 17 ThreadProcess > 0x7fffffff. See also footnote 3. This rule does not apply to the Idle thread (PID 0).

- Rule 18 StartAddress != 0 This rule does not apply to the Idle thread (PID 0).

- Rule 19 The structure must start with a DISPATCHER_HEADER of type 6 (thread).

- Rule 20 The structure must contain a notification timer and a semaphore at the positions designated in Table 3.

2.4. Validation

The rule set described above is required to identify all processes and threads currently running. Further it is

---

3 For systems booted with the /3 GB switch the border between user and kernel space moves up to 0x0bfffffffff.
expected to find traces of now defunct processes and threads.

For validation full memory dumps were obtained from clean installations of Microsoft Windows 2000 SP4, XP, XP SP1, XP SP2 and Windows Server 2003. For reference from these dumps a list of running processes and their associated threads was obtained through the Microsoft kernel debugger. Another list was compiled through the rule set as described above. The resulting lists were compared.

As it turns out the application of the rule set it did not miss a single process or thread shown by the debugger. As expected the rule set identified more objects than the debugger. Those turned out to be:

- the Idle process and thread
- terminated processes and threads
- artifacts from a previous boot

So the rule set did not falsely identify some random data as a process or thread.

### 3. Applications

As a proof-of-concept this set of rules was implemented in a Perl script named PTfinder, short for process and thread finder. A version based only on the DISPATCHER_HEADER structure of selected objects

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Offset by Windows Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofs PageDirectoryBase</td>
<td>0x018 0x018 0x018 0x018</td>
</tr>
<tr>
<td>ofs ThreadListHead.Flink</td>
<td>0x050 0x050 0x050 0x050</td>
</tr>
<tr>
<td>ofs ThreadListHead.Blink</td>
<td>0x054 0x054 0x054 0x054</td>
</tr>
<tr>
<td>ofs PID</td>
<td>0x09c 0x084 0x084 0x084</td>
</tr>
<tr>
<td>ofs PPID</td>
<td>0x1c8 0x14c 0x14c 0x128</td>
</tr>
<tr>
<td>ofs Sync. Event #1</td>
<td>0x070 n/a n/a n/a</td>
</tr>
<tr>
<td>ofs Sync. Event #2</td>
<td>0x13c 0x0d8 0x0d8 0x0d8</td>
</tr>
<tr>
<td>ofs Sync. Event #3</td>
<td>0x164 0x0fc 0x0fc 0x224</td>
</tr>
<tr>
<td>sizeof struct</td>
<td>0x290 0x258 0x260 0x278</td>
</tr>
</tbody>
</table>

Values printed in italics do not apply to the idle process.

and some additional checks was released in March 2006 at the DFN-CERT workshop (Schuster, 2006b). This version was limited to parsing the DISPATCHER_HEADER of Microsoft Windows 2000 only. However, it is possible to adopt it to other versions of Microsoft Windows with the values given in the tables at the end of this article.

Despite its limitations PTFinder worked as expected on the memory dumps of the DFRWS 2005 Memory Analysis Challenge, on the sample dump from Jones et al. (2005) and on some suspended VMware sessions during malware examinations.

#### 3.1. Persistence of processes through a reboot

According to kntlist’s readout of the KeBootTime kernel variable from the first DFRWS image the system was booted at 2005-06-05 00:32:27Z. This matches with the start time of most system processes. However, PTfinder reveals some processes which appear to have been started prior to that time:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Image name</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-06-03</td>
<td>01:25:53Z</td>
<td>csrss.exe</td>
<td>168</td>
</tr>
<tr>
<td>2005-06-03</td>
<td>01:25:54Z</td>
<td>winlogon.exe</td>
<td>164</td>
</tr>
</tbody>
</table>

According to the description of the DFRWS challenge the system then was rebooted and the second memory dump was obtained. Kntlist indicates this happened at 2005-06-05 15:00:56Z. Again this matches the start time of most system processes. And again PTfinder shows three processes which appear to have been started earlier:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Image name</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-06-03</td>
<td>01:25:53Z</td>
<td>csrss.exe</td>
<td>168</td>
</tr>
<tr>
<td>2005-06-05</td>
<td>00:32:40Z</td>
<td>smss.exe</td>
<td>156</td>
</tr>
<tr>
<td>2005-06-05</td>
<td>00:32:43Z</td>
<td>csrss.exe</td>
<td>180</td>
</tr>
</tbody>
</table>

These processes can also be found in the first image, at a matching file offset. Note that csrss.exe has at least survived one additional boot.
While this might look odd at first, it is in accordance with Chow et al. (2005) as well as with Farmer and Venema (2004, p. 182):

Although most computers automatically zero main memory upon rebooting – many do not. This is generally independent of the operating system; for instance, motherboards fueled by Intel CPUs tend to have BIOS settings that clear main memory upon restart, but there is no requirement for this to happen.

3.2. Incident response

Interpreting lists of several hundred processes and threads could become a tedious task. To address this issue a simple visualization feature was added to PTfinder. Based on the PID and PPID as given in the EPROCESS structure a parent–child relationship between processes can be drawn. In a similar way this can be also done for threads and their owning process. This time the information is taken from PID and TID contained in the ETHREAD structure.

PTfinder can express these relations in a way suitably to be processed by the graph visualization software Graphviz (AT&T, 2005). Graphviz can render the graph into several bitmap graphic formats or into a scalable vector graphic (SVG). The latter turned out to be very helpful when viewing large graphs. SVG files can be browsed with ZGRviewer (Pietriga, 2005), which seamlessly integrates Graphviz.

This environment could help an incident responder or forensic examiner to walk down the hierarchy of processes and threads uncovered by PTfinder. This could unveil the root-cause of an incident soon. For example Fig. 5 shows a detail of the process hierarchy which was produced from the first of the DFRWS images. In the middle there are two processes spawned by lsass.exe, the Windows local security authority subsystem. LSASS is not expected to spawn processes, so this already is an alarming find. In addition the spawned processes are named after the well-known Metasploit exploit construction framework.

At least one of the two exploits obviously was successful and led to the execution of another process named UMGR32.EXE. This observation should suffice to justify any further investigation and incident response measures.

3.3. Malware analysis

The method described above does not require any conversions between virtual and physical addresses to be made. Hence it is not bound to special dump file formats. Beside raw dumps like the ones provided in the Memory Analysis Challenge it was successfully tested with Windows crash dumps (DMP) and VMware suspended sessions (VMSS).

The latter was exploited several times to analyze encrypted malware. The malware was executed in a virtual machine. As soon as it had decrypted itself and started work, the VM was suspended. The VMware session file was then examined to locate EPROCESS structures of the malware. Finally the decrypted image was extracted from the memory dump.

Fig. 5 – Visualizing the hierarchy of processes could help in finding the root-cause during incident response. This part from the DFRWS challenge shows a trojan horse launched by a Metasploit LSASS exploit.

The analysis of ETHREAD structures indicates which part of a binary was run and at what time. Depending on the usage of threads in a binary this could add to timeline information obtained from the last access time of the binary or an event log entry.

4. Conclusions and future work

The simple approach of searching for processes and threads described in this article works surprisingly well for all versions of Microsoft Windows from 2000 up to Windows Server 2003.

First tests show that Microsoft Vista is going to reuse parts of the DISPATCHER_HEADER after object creation has been completed. This change in data usage indicates that some or all of the fields used for object identification might be meaningless for normal system operations. This could allow for some modified DKOM attacks targeting the bytes used to identify the objects. One should note here that if an attacker has acquired the privileges to change the contents of the non-paged pool, he might also have got the rights to modify the in-memory code from which any sort of “protection” stems.
The POOL_HEADER could provide a basis to search for other objects beside processes and threads. Pool tags are identifiable for all of the NT kernel objects like files, registry keys and devices. Yet it has to be checked whether unique tags exist to identify other kinds of helpful information. Building a reliable filter which will be based on the small POOL_HEADER is a challenging task. Work to analyze sample memory dumps and to formulate criteria is still in progress.

After all neither the method described in herein nor its implementation in PTfinder is meant to be a complete solution of the complex task of Windows memory analysis. Differences in the result sets obtained by searching the memory for objects and by enumerating the kernel’s internal lists indicate some malicious or in another way “interesting” activity. Searching for objects is not meant to be a replacement for, but an improvement of list-walking tools like kntlist.

Acknowledgements

This work was inspired by the Digital Forensics Research Workshop 2005 and its Memory Analysis Challenge.

Eoghan Casey kindly provided event log files from the computer used to generate the memory dumps for the DFRWS 2005 Memory Analysis Challenge. This helped in analyzing the persistence of some process traces through the systems’ reboot.

Finally the author wishes to thank George M. Garner Jr. for his insightful comments and enlightening discussions about Windows kernel internals.

References


Chow Jim, Pfaff Ben, Garfinkel Tal, Rosenblum Mendel. Shredding your garbage: reducing data lifetime through secure deallo-


Microsoft Corporation, Redmond. Changes to functionality in Microsoft Windows Server 2003 Service Pack 1, Device


Schuster Andreas. PTfinder version 0.2.0 released, March 2006b. Available from: http://computer.forensikblog.de/en/2006/03/ptfinder_0_2_00.html [2006.03.02].

Andreas Schuster is a Computer Forensic Specialist with the security department of Deutsche Telekom AG since December 2003. Previously he led a commercial computer incident response team and had worked in the internet business for about seven years. Andreas had got his first computer in 1981. In order to make the most out of 1024 bytes of main memory he had to acquire low-level programming skills. Though times have significantly changed Andreas regularly falls back to low-level tools like disassemblers and hex editors when he explores the inner mechanics of an operating system or a new piece of malware.