Pool Allocations as an Information Source in Windows Memory Forensics

Andreas Schuster
Deutsche Telekom AG
Group Security
Friedrich-Ebert-Allee 140
D-53113 Bonn, Germany
andreas.schuster@telekom.de

Abstract: The Microsoft Windows kernel provides a heap-like memory management, called "pools". Whenever some kernel-mode code requires an amount of memory, it is allocated from a pool. Ignoring the documented interface and searching the whole dump of physical memory for signatures of pool allocations allows the forensic examiner to gain information not only from currently active but also from freed and not yet overwritten allocations. Understanding the inner mechanics of memory pools enables an examiner to connect certain finds in memory to the originating piece of code. As an example this articles describes the steps necessary to detect traces of network activity in a memory dump.

1 Introduction

Forensic analysis of hosts commonly deals with storage media. Plenty of elaborate tools exist to analyze file systems and files. Not so when it comes to examinations of the main memory. Although it reflects the system's state, a memory image is obtained on rare occasions only. Examination procedures are usually limited to running *strings* or one of its derivatives on the dump to extract cached and unencrypted messages, pass phrases and keys. However most of the system's state like processes and their privileges or the ARP cache can not be retrieved that simply. A debugger would be the tool of choice for that.

Due to its high volatility and to avoid unnecessary pollution of the evidence it is widely accepted to copy the main memory to a file and examine it offline. Even up to date literature recommends to copy `\\.\PhysicalMemory` with *dd* or commercial tools like *X-Ways Capture*. Unfortunately the resulting 'raw' dumps can not be analyzed with Microsoft's suite of free debuggers, which relies on a proprietary dump file format.

1.1 Related Work

During the year 2005 three programs appeared on the scene and promised to provide at least some of the analysis capabilities of the debugger. The

Windows Memory Forensics Toolkit (WMFT) by MARIUSZ BURDACH [Bur05] enumerates processes and modules out of a dump file in raw format. Closely related is *MemParser* by CHRIS BETZ [Bet05]. In addition this tool evaluates a process' Program Environment Block (PEB). From this it reconstructs environment strings and a list of loaded Dynamic Link Libraries.

The Windows kernel keeps track of its objects with the help of several doubly-linked lists. For example all active processes are kept in such a list. A malicious piece of code in the kernel now could unlink a certain process from that list: That process would then become invisible for the kernel and all system monitoring tools. This hiding technique is called *Direct Kernel Object Manipulation* or in short DKOM. Obviously DKOM will hide an object from simple list-walking applications like *WMFT* and *MemParser*, too.

The by far most advanced tool of the list-walking category is *KnTList* by GEORGE M. GARNER JR. and ROBERT-JAN MORA [GM05]. It traverses and cross-checks several internal lists of the Windows kernel which reduces its susceptibility to DKOM. *KnTList* provides comprehensive lists of processes, threads, files, and other kernel objects.

As soon as an object reaches the end of its lifespan, e.g. a file is closed or a thread is terminated, the kernel removes it from the corresponding lists. From that point on its traces will be invisible to any pure list-walking tool, including *KnTList*. Hence lots of information of potential importance for a forensic investigation will remain inaccessible. The situation is comparable to a file system analysis procedure which would be unable to detect and process deleted files.

To overcome this limitation the author proposed the extension of list-walking tools by searching the memory image for process and thread objects [Sch06a]. The concept was successfully demonstrated by *PTFinder* [Sch06c].

1.2 Idea

Generating signatures for processes and threads is fairly simple. Both structures, `_EPROCESS` and `_ETHREAD`, are several hundred bytes long and contain multiple constant values at offsets distributed across the whole structure [Sch06d].

However taking this approach is generally not possible in case of smaller structures containing highly varying data. Examples are file name caches, atom tables, network interface configuration, ARP records and TCP connection status information.

Of course all these information has to be stored somewhere in kernel memory. Therefore this article analyzes the internals of the kernel's memory manager. It describes the `_POOL_HEADER`, a data structure used by the memory manager to keep track of small memory allocations. Finally the article describes a set of rules which allows to identify those allocations in a memory dump during a forensic examination. As an example this article describes how to gather all the information needed to locate and to interpret status information of TCP sockets. This enables a forensic examiner to find out on which ports a system was listening and at what time these sockets were put in listening state.

2 Memory Pools

2.1 Windows Memory Management

The Microsoft Windows NT kernel builds on memory management functionality provided by the Intel IA-32 CPU architecture. Memory is organized in pages, which usually are 4096 bytes in size. Kernel-mode routines don't allocate memory on their own but go through a set of functions which are implemented by a part of the kernel known as the Memory Manager.

The Memory Manager assigns pages to a pool. Data stored in the non-paged pool will never be paged out into the swap file. As such, the non-paged pool is a rare resource. Typical use includes data which is frequently accessed, like process and thread information, or data which can't be paged out like information needed by the Memory Manager itself.

If a requested portion of memory is larger than the page size, the Memory Manager will allocate as much pages as needed to fulfill the request. Any remaining space in the last page will be wasted. Requesting allocations larger than the page size is strongly discouraged, though.

For requests smaller than or equal to the page size the Memory Manager will try to find a properly sized free region within the requested pool. 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. This concept is widely known as Heap in userland applications.

2.2 Pool Allocations from a Programmer's View

Device drivers and the Windows kernel itself go through a set of functions to allocate memory from pools and return it later. The simplest possible routine for that is `nt!ExAllocatePool`. It accepts a pool type and allocation size. The type mainly decides whether the allocation will be made from the paged or non-paged pool. Some options allow to enforce alignment with CPU cache boundaries and to raise a bug check exception if the request fails. However these extra types are reserved for internal use by the Windows kernel. The size argument just specifies the requested amount of memory in bytes. The functions either returns a pointer to the allocated block of memory or NULL to indicate an error.

According to the documentation provided in the Microsoft Developer Network the `nt!ExAllocatePool` routine "is obsolete and is exported only for existing binaries" [Mic05a]. Programmers are asked to use `nt!ExAllocatePoolWithTag` instead. In fact `nt!ExAllocatePool` has been turned into a wrapper which just passes type and size of the request to `nt!ExAllocatePoolWithTag` and also adds a tag of "None".

The following piece of code from the Microsoft Windows Vista kernel shows a typical pool allocation sequence with a tag of "EtwQ". Please note that the tag's bytes are in reverse order.

```
00402889   push    'QwtE'          ; Tag
0040288E   push    10h             ; NumberOfBytes
00402890   push    NonPagedPool    ; PoolType
00402892   call    _ExAllocatePoolWithTag@12
```

The API also provides some other functions similar to `nt!ExAllocatePoolWithTag`:

- `nt!ExAllocatePoolSanityChecks` is undocumented and will be called only from the driver verifier's `nt!VeAllocatePoolWithTagPriority`. Among others it checks whether tags consist of printable characters only (via `nt!_ExpIsPoolTagPrintable`).
- `nt!ExAllocatePoolWithQuota` is obsolete. Like `nt!ExAllocatePool` it adds a tag of "None" and calls `nt!ExAllocatePoolWithQuotaTag`.
- `nt!ExAllocatePoolWithQuotaTag` allocates memory and charges the quota against the current process.
- `nt!ExAllocatePoolWithTagPriority` allows to specify the importance of the request.

As soon as an allocation isn't needed anymore, it should be returned into the memory pool. For this the API provides some correspondent functions:

- `nt!ExFreePool` is deprecated. In fact this is only a wrapper which calls `nt!ExFreePoolWithTag` with a PoolTag of `0x00000000`.
- `nt!ExFreePoolSanityChecks` is an undocumented function. It is called from the driver verifier only.
- `nt!ExFreePoolWithTag` is the standard way to free an allocation. The supplied tag must match the tag used to allocate the memory, otherwise the system will raise a bugcheck with an error code of `0xC2 BAD_POOL_CALLER` [Mic05c].

A similar set of allocators and deallocators exist for the so-called driver verifier. At the cost of CPU time and memory this set of routines thoroughly checks all parameters and isolates memory allocations by the driver under observation. Certain classes of drivers like the installable file system drivers also provide their own set of allocators and deallocators.

2.3 Searching for Pool Allocations

Windows keeps track of its pool allocations with the help of some structures of type `_POOL_DESCRIPTOR` and some lists of pages combined to form a pool. Each page then contains the allocations, each prefixed by a `_POOL_HEADER` structure.

```
+0x000 PreviousSize           : UChar
+0x001 PoolIndex              : UChar
+0x002 PoolType               : UChar
+0x003 BlockSize              : UChar
+0x004 PoolTag                : Uint4B
+0x004 AllocatorBackTraceIndex : Uint2B
+0x006 PoolTagHash            : Uint2B

+0x000 PreviousSize           : Pos 0, 9 Bits
+0x000 PoolIndex              : Pos 9, 7 Bits
+0x002 BlockSize              : Pos 0, 9 Bits
+0x002 PoolType               : Pos 9, 7 Bits
+0x004 PoolTag                : Uint4B
+0x004 AllocatorBackTraceIndex : Uint2B
+0x006 PoolTagHash            : Uint2B
```

Figure 1: Definitions of the `_POOL_HEADER` structure for Windows 2000 (top) and later versions (bottom).

As shown in figure 1 the `_POOL_HEADER` slightly varies between Microsoft Windows 2000 and later versions. The changes allow to allocate memory in smaller chunks, hence reducing the waste of precious kernel memory. For the Intel 32bit platform the chunk size is 32 for Windows 2000 and 8 bytes for Windows XP and later versions.

A pool header will not be found at an arbitrary offset, but at a multiple of the chunk size as defined before.

Rule 1 *OffsetInPage mod ChunkSize = 0*

Each `_POOL_HEADER` states the size of the whole allocation, that is the header plus the payload, in a field named *BlockSize*. As the header's size is included, *BlockSize* never can be null.

Rule 2 *BlockSize > 0*

Pool allocations smaller than a page do not cross page boundaries. Therefore there must be enough space left in the page to contain the allocation:

Rule 3 *BlockSize* $\times$ *ChunkSize + OffsetInPage = 4096*

Similarly *PreviousSize* gives the size of the preceding allocation. This allows to traverse the chain of allocations in a page in both directions.

Rule 4 *PreviousSize = 0, if the allocation starts at the beginning of a page, PreviousSize > 0 otherwise.*

There must be enough space for the previous allocation to fit in the page:

Rule 5 *PreviousSize* $\times$ *ChunkSize* $\leq$ *OffsetInPage*

Allocations must be properly chained. There's an exemption to be made for a sequence of freed allocations. In that case only the first allocation is marked as freed and their *BlockSize* is adjusted to include the subsequent freed allocations. Neither *BlockSize* nor *PoolType* is changed in the subsequent freed allocations.

Rule 6 *BlockSize*$_n$ = *PreviousSize*$_{n+1}$, *if the allocation n is not marked as free*
BlockSize$_n$ $\geq$ *PreviousSize*$_{n+1}$ *otherwise*

The *PoolType* mainly decides whether an allocation has to be kept permanently in memory or if it may be swapped out to disk to reclaim some physical memory. According to the documentation [Mic05d] some refinements exist, e.g. to enforce the alignment on cache boundaries. The *PoolType* stored in the `_POOL_HEADER` actually is incremented by 1. A type code of null indicates a freed allocation.

Rule 7 *PoolType must be in [0 .. 8, 33 .. 39].*

Rule 8 *All allocations in a page must either belong to the same pool (paged/non-paged) or be marked as free.*

The *PoolTag* associated with an allocation consists of up to four characters. According to the documentation [Mic05c] the ASCII value of each character as provided to `nt!ExAllocatePoolWithTag` must be between 0 and 127. The kernel uses the most significant bit to "protect"[1] an allocation. It has to be noted that such protection only prevents from accidental use of the same tag by third-party code.

Rule 9 *(PoolTag and 0x00808080) = 0*

This sort of protection is used by `nt!ObpAllocateObject`, a routine which allocates resources during object instantiation. In order to calculate the *PoolTag* it takes the *Key* from the object's type declaration and sets the most significant bit:

[1]Microsoft's kernel debugger marks such pool allocations as "protected".

```
004D7BD4 CheckForTag:
004D7BD4   cmp     edi, esi
004D7BD6   mov     eax, 'TjbO' ; default pool tag
004D7BDB   jz      short AllocateMemory
004D7BDD   mov     eax, [edi+_OBJECT_TYPE.Key]
004D7BE3 AllocateMemory:
004D7BE3   or      eax, 80000000h
004D7BE8   push    eax ; Tag
004D7BE9   mov     eax, [ebp+arg_10]
004D7BEC   add     ecx, eax
004D7BEE   push    ecx ; NumberOfBytes
004D7BEF   push    edx ; PoolType
004D7BF0   call    _ExAllocatePoolWithTag@12
```

3 Applications in Memory Forensics

3.1 Attributing Data to Code Fragments

Extracting sequences of printable characters with the help of *strings* or a similar tool is still a common method to analyze a memory dump. The problem here is to attribute a suspect string to a certain routine. If the suspect string is found inside of a pool allocation, the *PoolTag* can be helpful in this task.

The first step would be to find the driver file that uses the tag. Microsoft describes [Mic04] how to search for files containing a certain *PoolTag* with the *findstr* utility or the search applet. If the resulting set of files is too large, the article recommends to put the letter "h" in front of the tag.

The letter "h" maps to an ASCII value of `0x68`. In machine language this is interpreted as "push" instruction. So when following Microsoft's advice one could miss a call of the allocation routine whenever the pool tag is not passed as an immediate operand, e.g. in a register.

As stated above, on little-endian machines like the Intel Pentium the programmer has to provide the pool tag in reverse order. However some programmers prefer to let the computer do the conversion at run-time. The following sample has been taken from a Windows Vista kernel. Note the usage of the "bswap" instruction to reverse the tag:

```
0047DFEF   mov     eax, 'Hvlm' ; normal orientation
0047DFF4   bswap   eax ; byte reversal
0047DFF6   push    eax ; Tag
0047DFF7   push    edi ; NumberOfBytes
0047DFF8   push    esi ; PoolType
0047DFF9   call    _ExAllocatePoolWithTag@12
```

So for forensic purposes files should be searched for the *PoolTag* in normal and reversed byte order. If the "protected" bit is set in the memory image, the file should also be searched for the *PoolTag* with the most significant bit in both possible states.

Several well-known pool tags and their origin are documented in a file named `pooltag.txt` which ships with the Device Driver Kit (DDK) or Windows Driver Development Kit (WDK).

3.2 Finding Data of Known Functions

Sometimes it is difficult to locate data structures in a memory dump which belong to a known routine. Usually a kernel debugger would be the right tool for this task. However memory dumps are still commonly obtained in a "raw" format [Bro05, p. 223f.] by *dd* or with commercial tools like *X-Ways Capture*. These dumps are lacking some CPU state information required by the debugger. Manually following the extensive data structures of the kernel's memory manager would consume too much time. So an alternate procedure is needed to locate the data.

If the requested data is stored in a memory pool, searching for allocation could be an option. First the proper *PoolTag* needs to be determined from the code. Microsoft recommends that "each allocation code path should use a unique pool tag to help debuggers and verifiers identify the code path" [Mic05b]. If against this recommendation the same tag is used throughout several places then additional information like the affected pool (paged or non-paged) or a fixed allocation size will be helpful in locating the proper pool allocations.

3.3 A Use Case: Retrieving TCP/IP Socket Activity

An example shall illustrate this procedure: TCP/IP sockets in an listening state could be an indicator for potentially unwanted system activity. For example, an exploit might have spawned a listener bound to a privileged command shell. Windows versions from 2000 to 2003 handle sockets through the *Transport Device Interface* (TDI). Some research reveals that the TDI function `tcpip!TdiOpenAddress` is responsible for socket creation and setup. This function is implemented in the `tcpip.sys` driver file. The following code fragments were taken from said function as implemented in Microsoft Windows XP SP2. They will help to identify and interpret pool allocations containing information about network sockets:

```
0001CD5D   push    NormalPagePriority
0001CD5F   push    'APCT'          ; Tag
0001CD64   push    360             ; NumberOfBytes
0001CD69   push    NonPagedPool
0001CD6B   call    ds:_ExAllocatePoolWithTagPriority@16
...
```

```
0001CF4D   mov     eax, [ebp+var_LocalAddress]
0001CF50   mov     [esi+44], eax
0001CF53   mov     al, byte ptr [ebp+arg_Protocol]
0001CF56   mov     [esi+50], al
...
0001CF5C   mov     [esi+48], di    ; LocalPort
...
0001CF76   call    _PsGetCurrentProcessId@0
0001CF7B   mov     [esi+328], eax
0001CF81   lea     eax, [esi+344]  ; CurrentTime
0001CF87   push    eax
0001CF88   call    ds:_KeQuerySystemTime@4
```

So obviously during examination one would have to search for pool allocations labeled "TCPA" and a size of 368 bytes (360 bytes for the payload and 8 for the _POOL_HEADER). These allocations will reside in the non-paged pool.

To demonstrate the ability to recover even defunct connection objects, a clean installation of Microsoft Windows XP SP2 was run in VMware 5.5.1. A netcat listener was started on TCP port 666, simulating the typical aftermath of a compromise. After a while netcat was terminated without a connection being made. The VMware session was suspended, causing VMware to store the emulated physical memory into a VMEM file. This file then was searched for pool allocations matching the aforementioned conditions, leading to the following results.

```
192.168.186.128:138/UDP, PID=4, 2006-07-17 22:08:47
0.0.0.0:135/TCP, PID=800, 2006-07-17 22:08:40
0.0.0.0:0/IGMP, PID=884, 2006-07-17 22:08:49
0.0.0.0:0/GRE, PID=4, 2006-07-17 22:08:51
0.0.0.0:1029/UDP, PID=948, 2006-07-17 22:09:46
127.0.0.1:1025/TCP, PID=1508, 2006-07-17 22:08:51
0.0.0.0:666/TCP, PID=1448, 2006-07-17 22:11:15 (defunct)
192.168.186.128:139/TCP, PID=4, 2006-07-17 22:08:47
192.168.186.128:137/UDP, PID=4, 2006-07-17 22:08:47
127.0.0.1:1028/UDP, PID=884, 2006-07-17 22:08:54
0.0.0.0:1026/TCP, PID=4, 2006-07-17 22:08:51
0.0.0.0:445/TCP, PID=4, 2006-07-17 22:08:27
0.0.0.0:445/UDP, PID=4, 2006-07-17 22:08:27
127.0.0.1:1027/UDP, PID=884, 2006-07-17 22:08:54
```

The listening socket belonging to the netcat process is clearly visible in the middle of the list. Its Unique Process ID (PID) could be confirmed by *PTFinder*. The pool allocation was already marked as "free", but had not been reused yet. In a similar way one could also monitor TCP connections.

Though no TCP connections were established deliberately in the course of this experiment, there were three connections observed with endpoints in address space registered to

Microsoft and Akamai:

```
192.168.186.128:1037 -> 213.253.9.70:80, PID=884
192.168.186.128:1038 -> 213.253.9.70:80, PID=884
192.168.186.128:1039 -> 64.4.21.93:80, PID=884
```

The associated Unique Process ID belongs to an instance of `svchost.exe`. This behaviour most likely was caused by the Windows Update service.

4 Conclusions and Future Work

As a proof of concept the rules defined in section 2.3 were implemented in a Perl script named *PoolFinder* [Sch06b]. So far this script was used to search for memory pool allocations in memory dumps obtained from Windows versions from 2000 to Vista.

The script is capable of locating freed and unclaimed allocations. While a debugger shows consecutive freed allocations as aggregate, *PoolFinder* identifies each single block. This enabled the author to locate and identify traces of a process, which was not visible to a debugger and *PTFinder*.

During analysis of the DFRWS images also the persistence of data through a reboot was observed. *PoolFinder* found allocations which were written during a prior run of Windows. Obviously this would be impossible with a debugger because the memory region would be marked as unused in the currently running instance of Windows.

An examiner working with pool allocations should be aware that pool tags were mainly introduced in order to debug memory allocation errors in driver code. Programmers are free in what pool tags they use. There is no registry for pool tag identifiers, neither at Microsoft nor at another central location nor on the running instance of Windows. So ambiguities in the use of pool tags might occur accidentally or intentionally.

Microsoft teaches how to misuse some well-known pool tags to conceal allocations in PatchGuard for Windows x64. PatchGuard also varies the size of allocations made in an attempt to complicate detection of its data structures in the non-paged pool [sS05].

This article explained how to structure and parse kernel memory of systems running Microsoft Windows. Further research could build upon this description in order to to detect suspicious events through statistics of pool allocations in comparison with baselines obtained from pristine systems.

As CHOW, PFAFF, GARFINKEL AND ROSENBLUM showed some information in kernel memory can survive periods over 14 days and longer while the system is used normally [CPGR05]. For their article they observed the decay of network buffers filled with a certain "marker" byte sequence over time. This kind of experiment should be carried on with, considering different pool types, allocation sizes and system load profiles, e.g. web server, domain controller and desktop computer. The results would help in determining response time limits for an effective incident response.

References

[Bet05] Chris Betz. MemParser. August 2005. Online. http://www.dfrws.org/2005/challenge/memparser.html (2005-12-15).

[Bro05] Christopher L. T. Brown. *Computer Evidence: Collection & Preservation*. Charles River Media, Hingham, MA, September 2005.

[Bur05] Mariusz Burdach. Windows Memory Forensic Toolkit. July 2005. Online. http://forensic.seccure.net/tools/wmft.tar.gz (2005-12-15).

[CPGR05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proc. 14th USENIX Security Symposium*, August 2005. Online. http://footstool.stanford.edu/~jchow/papers/usenixsec05/secdealloc-usenix05.pdf (2006-04-20).

[GM05] George M. Garner and Robert-Jan Mora. kntlist. August 2005. Online. http://www.dfrws.org/2005/challenge/kntlist.html (2005-12-15).

[Mic04] Microsoft Corporation, Redmond. *How to find pool tags that are used by third-party drivers*, June 2004. Online. http://support.microsoft.com/kb/298102/en-us (2006-12-15).

[Mic05a] Microsoft Corporation, Redmond. *ExAllocatePool*, May 2005. Online. http://msdn.microsoft.com/library/en-us/Kernel_r/hh/Kernel_r/k102_02ff5510-3d96-4a15-a0da-5da56e14b1b8.xml.asp (2005-12-15).

[Mic05b] Microsoft Corporation, Redmond. *ExAllocatePoolWithTag*, May 2005. Online. http://msdn.microsoft.com/library/en-us/Kernel_r/hh/Kernel_r/k102_13ab2d7e-dd96-4474-bf27-59ee9b7d84d6.xml.asp (2005-12-15).

[Mic05c] Microsoft Corporation, Redmond. *ExFreePoolWithTag*, May 2005. Online. http://msdn.microsoft.com/library/en-us/Kernel_r/hh/Kernel_r/k102_03ac2997-acff-40b6-a110-718261627130.xml.asp (2005-12-15).

[Mic05d] Microsoft Corporation, Redmond. *POOL_TYPE*, May 2005. Online. http://msdn.microsoft.com/library/en-us/Kernel_r/hh/Kernel_r/k112_90446d42-0e73-4da3-a3df-27efe3daa67b.xml.asp (2006-04-09).

[Sch06a] Andreas Schuster. Forensische Analyse des Arbeitsspeichers am Beispiel von Microsoft Windows 2000. In Christian Paulsen, editor, *13. DFN-CERT Workshop "Sicherheit in vernetzten Systemen"*, pages I1–20, Hamburg, March 2006. DFN-CERT GmbH.

[Sch06b] Andreas Schuster. PoolFinder version 1.0.0 released. October 2006. Online. http://computer.forensikblog.de/en/2006/10/poolfinder_1_0_0.html (2006-10-10).

[Sch06c] Andreas Schuster. PTfinder version 0.2.00 released. March 2006. Online. http://computer.forensikblog.de/en/2006/03/ptfinder_0_2_00.html (2006-03-02).

[Sch06d] Andreas Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Investigation*, 3(Supplement 1):10–16, September 2006. `doi:10.1016/j.diin.2006.06.010`.

[sS05] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, 3, December 2005. Online. `http://www.uninformed.org/?v=3&a=3&t=txt` (2006-04-09).