# The Impact Of Microsoft Windows Pool Allocation Strategies On Memory Forensics

*By*

**Andreas Schuster**

*From the proceedings of*

The Digital Forensic Research Conference

**DFRWS 2008 USA**

Baltimore, MD (Aug 11th - 13th)

available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

# The impact of Microsoft Windows pool allocation strategies on memory forensics

*Andreas Schuster*

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

*Keywords:*
Microsoft Windows
Volatile data
Pool memory
Process Persistence

ABSTRACT

An image of a computer's physical memory can provide a forensic examiner with a wealth of information. A small area of system memory, the nonpaged pool, contains lots of information about currently and formerly active processes. As this paper shows, more than 90% of such information can be retrieved even 24 h after process termination under optimum conditions.

Great care must be taken as the acquisition process usually affects the memory contents to be acquired. In order minimize the impact on volatile data, this paper for the first time analyzes the pool allocation mechanism of the Microsoft Windows operating system. It describes a test arrangement, which allows to obtain a time series of physical memory images, while it also reduces the effect on the observed operating system.

Using this environment it was found that allocations from the nonpaged pool are reused based on their size and a last in-first out schedule. In addition, a passive memory compaction strategy may apply. So, the creation of a new object is likely to eradicate the evidence of an object of the same class that was destructed just before. The paper concludes with a discussion of the implications for incident response procedures, forensic examinations, and the creation of forensic tools.

## 1. Introduction

Over the last years, physical memory has gained weight as an information source in computer forensics. Comprehensive and unique information about a system's state can be extracted from an image of its main memory.

In addition to the current state, it is possible to derive a lot of information about a system's past from the memory dump. Among other things there is timestamped information about processes, threads and network activity.

The research presented in this paper was stimulated by an oddity: The author was creating sample images for a course on Windows memory forensics. For one of the images, a process was hidden by means of the FU Rootkit (2005). FU is controlled through a console program. Immediately after the control program had terminated, a memory image was obtained by copying from the `\\.\PhysicalMemory` device. As expected, the "hidden" process was still visible to forensic tools. But the control program's `_EPROCESS` structure could not be found. So while the effect of the manipulation was clearly visible, the evidence about the malicious tool was missing. The author suspected the control program destroyed its own `_EPROCESS` block to foil detection. However, no evidence to prove this theory could be found during a thorough examination of the binary.

From this observation arose some questions:

(1) How long can an `_EPROCESS` structure persist in system pool memory after the process has been terminated?

E-mail address: andreas.schuster@telekom.de
1742-2876/$ – see front matter © 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2008.05.007

(2) What mechanisms affect the persistence of freed allocations in pool memory?
(3) What are the implications on forensic memory acquisition and incident response tools?

The answers to these questions could become the key to designing better incident response procedures and forensic memory acquisition tools.

### 1.1. *Related work*

Farmer and Venema (2005) measured the decay of freed memory on FreeBSD 4.1 and ReadHat 6.2 Linux. They found that after ''some 10 min, about 90% of the monitored memory was changed''.

Chow et al. (2005) recognized that Microsoft Windows employs an Idle Thread that continually fills freed pages with zeros. They forced Windows to load marked and timestamped data into kernel memory by sending it through a local TCP/IP connection. After 14 days of ''everyday work'', 3 MB from initially 4 MB of marked data were still accessible.

Walters and Petroni (2007) measured the effect of idle activity on the memory of Microsoft Windows XP with Service Pack 2. The test system was running in a virtual machine. Only system services, a virus scanner and the screen saver were active during the tests. The authors found that after 15 h about 85% of 512 MB RAM were unchanged (73% at 256 MB RAM).

After building the baseline, Walters and Petroni (2007) repeated the tests for dd and the Windows Forensic Toolchest (WFT) from the Helix CD. At 512 MB of RAM, dd changed about 10% and WFT about 30%.

Solomon et al. (2007) researched the persistence of data in the userland portion of the virtual address space on Microsoft Windows XP with Service Pack 2. They imposed 100, 1000 and 1500 concurrent probe processes on their test system. Each probe marks memory pages in userland with a unique string and a timestamp. According to the authors, ''the majority of pages persisted for less than 5 min[utes] with single pages only lasting longer''.

In case of a text editor the userpace would hold a copy of the program binary. It would also contain the user's text in the editor's internal representation. According to Solomon et al. (2007) this information can be expected to deplete within 5 min after the text editor has been terminated.

But there's more information about the text editor available: where the program binary was stored on the disk, when it was launched and by what user ID, when it was terminated, how many bytes it sent through IO-channels, and so on. All of this information is stored in kernel memory. This paper shall answer the question, whether a similar 5-min time barrier applies to that data, too.

### 1.2. *Pool allocations*

The Microsoft Windows kernel is based on an object oriented design. Kernel objects are represented by portions of data that range from 10s to 100s of bytes in size. Together with other small blocks of data that are used by the kernel, these are kept in special areas of kernel memory, called *pools*. On an average Windows XP desktop computer, the pools contain between 40,000 and 80,000 allocations, depending on uptime and level of system activity.

There are mainly two different types of pools. The *paged pool* can hold up to 491 MiB[1] of data (Russinovich and Salomon, 2005). As any other virtual memory, parts of this pool can be moved (paged) between physical memory and the page file on demand.

In contrast, the *nonpaged pool* is permanently kept in physical memory. With a maximum size of 256 MiB[2] it is significantly smaller than the paged pool (Russinovich and Salomon, 2005). The nonpaged pool is reserved for data that either need to be accessed frequently or that need to be accessed at times where the paging mechanism is not available.

Data that describe process and thread objects are kept in the nonpaged pool. So, the rest of this paper will refer to the nonpaged pool only. This greatly simplifies the experimental work as there is no need to deal with the complex mechanisms of paging. On the other hand, it will not confine the understanding of the kernel's pool allocation strategies.

The nonpaged pool consists of two areas in the virtual address space. Their bounds are marked by the global kernel variables `MmNonPagedPoolStart` and `MmNonPagedPoolEnd0` for the lower area and `MmNonPagedPoolExpansionStart` and `MmNonPagedPoolEnd` for the upper area, respectively. The total size is reported by `MmSizeOfNonPagedPoolInBytes`.

There is a whole family of allocator functions, with `ExAllocatePoolWithTag` (Microsoft Corporation, 2008) being the most commonly used one. For each allocator there is also a matching deallocator. Subsystems may derive their own set of functions, like the File System Runtime (FSRtl) does.

The returned block of memory will be aligned on an 8 byte boundary (16 bytes for 64 bit versions of Microsoft Windows). Also, the allocation will be prefixed by a `_POOL_HEADER` structure. The definition for 32 bit hardware platforms is shown in Fig. 1. There are only slight differences to 64 bit platforms.

## 2. Method

Some experiments shall improve the understanding of allocation strategies. The experimentation environment should support the observation of short-term and long-term behavior of allocations in the nonpaged pool. Experiments should be reproducible. Consequently the acquisition process is required to affect the system to the lowest possible extent.

### 2.1. *Environment*

The environment will *not* reflect real-life conditions. In contrary, it tries to avoid any unneeded background activity in order to provide an unobstructed, microscopic view at the activities that are associated with allocation and deallocation of pool memory.

[1] For 32 bit versions of Microsoft Windows 2000 and XP, 650 MiB for Windows Server 2003.

[2] For 32 bit versions of Microsoft Windows, 128 MiB if booted with the/3GB switch.

```
kd> dt -v -b _POOL_HEADER
struct _POOL_HEADER, 9 elements, 0x8 bytes
   +0x000 PreviousSize     : Bitfield Pos 0, 9 Bits
   +0x000 PoolIndex        : Bitfield Pos 9, 7 Bits
   +0x002 BlockSize        : Bitfield Pos 0, 9 Bits
   +0x002 PoolType         : Bitfield Pos 9, 7 Bits
   +0x000 Ulong1           : Uint4B
   +0x004 ProcessBilled    : Ptr32 to _EPROCESS
   +0x004 PoolTag          : Uint4B
   +0x004 AllocatorBackTraceIndex : Uint2B
   +0x006 PoolTagHash      : Uint2B
```

**Fig. 1 – _POOL_HEADER structure of 32 bit versions of Microsoft Windows.**

The experimentation environment was built upon the VMware Workstation 6.0.2 (VMware, Inc., 2008) virtual machine monitor. VMware allows the creation of a snapshot at any time. The resulting `.vmem` file provides an image of the emulated physical memory. At this, the memory acquisition process does not affect the observed system.

Unfortunately, the Microsoft debugger is unable to handle raw memory images like those contained in a VMware snapshot. One can still chose to execute the debugger inside the examined system. However, one of the major drawbacks of this method is that the debugger creates new threads to execute certain commands. So, this method is very likely to interfere with the primary object of investigation. Therefore the best procedure is to restore the snapshot after a debug command was executed.

The VM was configured to use 128 MiB of RAM. This helped to keep the resulting memory images small, while there was still enough memory to keep the guest operating system from frequent paging.

A subdirectory of the host system was made accessible to the VM by means of VMware's "shared folders". The VM was granted write access to the shared folder. While this required several network services, it allowed to modify the probe binary and the exchange of log data without modifying the VM's configuration and state.

Finally Microsoft Windows XP with Service Pack 2 was installed in the VM. Unneeded system services were disabled in order to reduce unwanted background activity and memory usage. The event log files, which are mapped into a region of shared memory, were trimmed to their allowed minimum size to conserve memory.

The shared folder was permanently mapped as a network drive. A command console was opened. Its working directory was then changed to the shared folder.

Microsoft Debugging Tools for Windows (Microsoft Corporation, 2007) were installed and the required symbol files were downloaded to the guest. The debugger was launched and a local debugging session was established.

The whole configuration was then saved as the baseline snapshot. There are 17 processes[3] and 190 threads active. As reported by the `!xpoolmap 0` extension command of the Microsoft debugger, the nonpaged pool uses 703 out of 12,748 allocated pages (utilization 5.5%). Background processes caused the creation of about 16 threads per hour.

[3] Counting system and idle as one process.

## 2.2. *Probe*

A small program was created that recursively spawns itself for a specified number of instances. Every instance then records its instance number and process ID in a log file. Relevant parts of the probe's code are shown in Fig. 2.

The last instance would then wait and prompt the operator to press "Enter". Usually an additional snapshot will be created during the break.

The probe was implemented in Perl and transformed into an executable by means of Perl2Exe (IndigoSTAR Software, 2007). Two slightly different versions were created. For the first version the probe was tuned into an EXE file, while the stub Perl interpreter and supporting modules were provided as DLL files.

For the second version all those file were linked into a single self-extracting executable. Upon execution this file creates a temporary directory and then splits and unpacks itself into the same set of files as in the first version. After execution, these files and the directory are then deleted automatically. In effect, this version of the probe causes some additional file system activity.

## 2.3. *Experiments*

For each of the following experiments a script for the Windows command processor was created. Such a script mainly contains invocations of the probe, whereas a suitable number of instances are passed as an option. For a sample script see Fig. 3.

The script and the probe binary were copied to the shared folder. Prior to every experiment the first snapshot was restored, thus bringing the environment into a known and reproducible state. The script was then executed by the command interpreter.

Each instance of the probe will cause the creation of a new process along with a single thread. The `_EPROCESS` structures of interest can be identified by means of PTFinder (Schuster, 2006a, 2007a). Thus the pool allocations are "marked" by an `_EPROCESS` structure, the `ImageFileName` of the probe binary and the recorded process ID. They are also "timestamped", because the operating system maintains create and exit timestamps in the `_EPROCESS` structure. Results were then cross-checked with Volatility (Walters and Petroni, 2008) in `psscan` mode to ensure that no defunct processes and threads were missed. Both programs, PTFinder and Volatility, scan memory images for traces of processes and threads. However, there are slightly differences in the signatures used. No differences were detected.

Where necessary, changes to single pool allocations were tracked by means of PoolFinder and its companion tools (Schuster, 2006b, 2007b).

## 3. Results

## 3.1. *Persistence of object data*

In order to observe the persistence of objects in the nonpaged pool, 100 processes were created. This took about 80 s. During

```
#!/usr/bin/perl

# MemLoader.pl

sub do_spawn_self {
   my @args;

   push @args, $0;
   # mandatory parameters
   push @args, '--instances', $opt_instances;
   push @args, '--current', $current_instance + 1;
   # optional parameters
   push @args, ($opt_interactive)? '--interactive' : '--
       nointeractive';
   if ($opt_logfile ne '') {
      push @args, '--logfile', $opt_logfile;
   };

   # execute child
   system(@args);
};

# =====================================================
# main routine

printf "Instance␣%d␣of␣%d,␣PID␣%d␣(0x%x)\n",
   $current_instance, $opt_instances,
   $$, $$;

if ($current_instance >= $opt_instances) {
   if ($opt_interactive) {
      prompt('x', 'All␣instances␣created.␣Press␣ENTER␣to␣
          continue...', '', 'ok');
   };
} else {
   &do_spawn_self();
};

if ($current_instance == 1) {
   print "done.\n";

   if ($opt_logfile) {
      open(LOG, ">>", $opt_logfile);
      printf LOG "#␣DONE\n\n";
      close(LOG);
   };
};
```

**Fig. 2 – Selected parts of the probe's code.**

the next 5 min the system could calm down, while the processes were in their idle loop waiting for user input. A first snapshot (pre) was taken for reference.

Now the processes were terminated in reverse order, no. 100 through no. 1. This took about 4 min and 4 s. As soon as the last process terminated, a timer was started and the first snapshot (0 m) in the time series was created. Additional snapshots were obtained at 1, 5, 15, 30 and 60 min as well as 24 h from there.

For comparison the same procedure was repeated using the second version of the probe, which causes more file system activity. Process creation took twice as long (160 s). About 5 min were needed to terminate the processes.

The memory images were scanned for process objects by means of PTFinder and the (defunct) probe processes were counted. The results are shown in Table 1.

From the 10 artifacts that were overwritten during the first 24 h, eight were overwritten by `_ETHREAD` structures associated with background processes (SYSTEM and various instances of `svchost.exe`), one was overwritten by network related data and one was overwritten by data of unknown origin.

In case of the provoked file system activity the numbers are slightly different: seven former process objects were overwritten by file system related data. In three cases the allocations now contain NTFS MFT entries for some of the probe's DLL files. One allocation was overwritten by a virtual address descriptor. Another allocation was related to network activity. Finally, three allocations were overwritten by `_ETHREAD` structures that could be linked to background activity.

The experiment shows that a significant part of freed allocations from the nonpaged pool can persist for 24 h or

```
@ECHO OFF
REM ****************************************
REM Test #2
REM
REM Launch 3 processes, then another one.
REM Which EPROCESS gets overwritten?
REM ****************************************

SET PROG=memloader
SET LOG=--logfile test2.log

ECHO creating processes 1-3
%PROG% %LOG% --instances 3 --interactive
ECHO.
ECHO creating process 4
%PROG% %LOG% --instances 1 --interactive
ECHO.
ECHO Test 2 completed.
```

**Fig. 3 – A sample command script.**

even longer, if system load permits. No intentional cleaning could be observed. This is in contrast to the Idle Thread cleaning up unused pages that were in use by the userland, as reported in Solomon et al. (2007).

However, background activity like the aforementioned threads of the local security authority subsystem, the system process, and the various instances of svchost.exe will overwrite information about defunct processes and threads over time.

## 3.2. *Re-allocation strategy*

The previous experiment still doesn't explain the initial observation that a rootkit's control program could not be found in a memory image at least once during a number of trials. In order to better understand the effect, the situation was reproduced in the controlled environment.

Three probe processes were launched and a snapshot was taken while they were all running in parallel. The processes were then terminated in reverse order and again a snapshot was taken. Now a fourth probe was launched and another snapshot was obtained while it was running.

The results are shown in Table 2. For every process the kernel assigns a new unique process ID (PID). Also each page directory is created at a different physical address.

**Table 1 – Number of identifiable process objects over time, with and without file system activity**

| Time | Without file system activity | With file system activity |
|---|---|---|
| Pre | 100 | 100 |
| 0 m | 97 | 93 |
| 1 m | 97 | 88 |
| 5 m | 93 | 88 |
| 15 m | 93 | 88 |
| 30 m | 93 | 88 |
| 60 m | 93 | 88 |
| 24 h | 90 | 88 |

**Table 2 – Re-allocation of process objects**

| Probe | PID | EPROCESS | Page directory |
|---|---|---|---|
| 1 | 464 | 0x04c9a020 | 0x06bf1000 |
| 2 | 492 | 0x04878da0 | 0x01876000 |
| 3 | 500 | 0x01082da0 | 0x04b9f000 |
| 4 | 540 | 0x04c9a020 | 0x039f9000 |

Interestingly the _EPROCESS structure for probe no. 4 is stored in the same location where the structure describing probe no. 1 – the one that was launched first and terminated last – was kept before. This could be repeated for several times, with the _EPROCESS ending up in the same place every time.

In order to explain this behavior, one needs to understand another important data structure, the _POOL_DESCRIPTOR (see Fig. 4). The pool descriptor helps the system to keep track of reusable blocks in a pool. There is only a single descriptor serving the nonpaged pool, which is appointed by the global kernel variable NonPagedPoolDescriptor.

The pool descriptor provides some basic information about the pool type (paged or nonpaged) and the pool's utilization. By far the largest part consists of an array of 512 list heads.

Whenever an allocation is freed and returned into the pool, the system checks if one of the neighboring blocks is marked as free, too. In that case, the pool manager merges the blocks.

Now the block size is divided by the granularity[4] and decremented by one, giving the number of the appropriate list head. The list head is adjusted to point to the newly freed allocation. Also the beginning of the allocation's payload area is overwritten by the former list head pointers, thus forming a doubly linked list of equally sized free blocks (SoBeIt, 2005; Johnson, 2007).

Whenever the kernel has to serve a new allocation request, it will return the address from the proper list head. The base objects like "Process" and "Thread" are of a fixed size. Therefore the _EPROCESS structure of a newly created process will overwrite the object data of a process that has been terminated just before.

If the kernel cannot fulfill a request because there is no free block of a matching size, it will try a larger block size. Hence the kernel will search the next list heads. This explains why _EPROCESS blocks (640 bytes, list head no. 79) are overwritten on occasion by slightly smaller _ETHREAD structures (632 bytes, list head no. 78).

## 3.3. *Memory compaction*

Two observations are still worth mentioning. During the first experiment, the count of used pages *decreased* from 703 (the baseline) to 697. This could be reproduced by freshly booting the baseline image (748 pages used) and repeatedly creating and terminating processes for a while.

The contiguous, low level of activity seemingly provided a stimulus to compact the nonpaged pool after the extremely busy bootstrap phase.

[4] Eight bytes for 32 bit versions from Windows XP on.

```
kd> dt -v -b _POOL_DESCRIPTOR
struct _POOL_DESCRIPTOR, 11 elements, 0x1028 bytes
   +0x000 PoolType         : Enum _POOL_TYPE
   +0x004 PoolIndex        : Uint4B
   +0x008 RunningAllocs    : Uint4B
   +0x00c RunningDeAllocs  : Uint4B
   +0x010 TotalPages       : Uint4B
   +0x014 TotalBigPages    : Uint4B
   +0x018 Threshold        : Uint4B
   +0x01c LockAddress      : Ptr32 to
   +0x020 PendingFrees     : Ptr32 to
   +0x024 PendingFreeDepth : Int4B
   +0x028 ListHeads        : (512 elements)
      +0x000 Flink            : Ptr32 to _LIST_ENTRY
      +0x004 Blink            : Ptr32 to _LIST_ENTRY
```

**Fig. 4 – _POOL_DESCRIPTOR structure of 32 bit versions of Microsoft Windows.**

But what means are there to compact pool memory? An active strategy would allow the pool manager to move allocations around within a page or even across page boundaries. Sure, this would be an effective strategy, similar to defragmenting a file system. However, unlike a file system, there is no single handle (like the file name) to an allocated portion of memory.

Whenever memory is allocated from the pool, the pool manager just returns a pointer to the beginning of the assigned block. The application then accesses its assigned memory directly through this pointer. The application is free to store the pointer in as many places as it likes to. So, there is no way for the pool manager to update the pointer. Also, there is no protocol that allows the pool manager to notify the application about an address change. Therefore, an active strategy seems to be highly unlikely.

A slight variation of the second experiment indicated an alternate way to compact the pool memory: Again three processes were created and then terminated. Then, three new processes were created. Only two of the `_EPROCESS` structures were stored in locations that hold similar structures before. Though there was a suitable free block, the pool manager assigned another block at a higher address.

The repeated execution of this experiment with higher process counts indicates that the pool manager tends to assign blocks at lower addresses in the first part of the nonpaged pool and at higher addresses from the nonpaged pool extension, respectively. This passive strategy creates an area of free memory in the middle of the pool. Even the largest requests could be served from this area, when needed. Also, completely unused pages could in that area be returned to the memory manager to free physical memory.

## 4. Conclusions

Unlike the userland portion of memory, system pool memory can persist for a long time – probably for as long as the system process is running.

The system re-allocates blocks of memory primarily based on their size. If no matching block is found, the system will search for larger ones. If there are multiple free blocks of the same size, they are reused according to a ''last in–first out'' rule.

Incident response toolkits may run batches of dozens of tools, some probably in parallel. Thus the impact on the free lists that are usually populated by remains of processes and threads might be devastating. It is advisable to carefully check the resource consumption of tools prior to their first usage in the field.

Forensic software that is expected to execute on the system under examination should use as little system resources as possible. This does not apply to the memory footprint in userland alone, but especially to threads, network sockets, files and other resources that leave traces in system memory pools.

Large organizations are likely to deploy agent-based incident response and forensic tools. They enable the IT security staff to make an instant connection to a suspect machine, examine its state and obtain a forensic image of the hard disk drive and the main memory. Agents will generally be installed prior to an incident and then run in the background. So they could provide the easiest solution to the problems of system memory re-allocation: During system boot, the agent could pre-allocate a reasonable amount of system resources, e.g. create threads. In case of an incident the required resources would then be readily available, without interfering with leftovers of any suspicious activity.

On the other side, forensic agents need to execute code at the system privilege level and communicate over the network. So, a security vulnerability in the agent's code could expose the monitored host to a remote compromise. In addition, a knowledgeable attacker may detect the agent and respond accordingly.

As mentioned earlier, the Microsoft Windows kernel overwrites the beginning of a block after it has been returned into the pool. This could cause any tool that was not specifically designed to handle those situations to malfunction or to produce misleading results. If, for instance, the Microsoft debugger is used to interpret a terminated process object, it is likely to display a ridiculously high handle count in the `_OBJECT_HEADER`. This happens, because the true handle count was overwritten with a pointer to the next free block. Any forensic examiner who analyzes Windows memory images should be aware of this. Also authors of dedicated memory forensic tools should be aware of the situation and design their tools carefully.

Further research is needed to fully understand the compaction strategy that is pursued by the pool manager. Also, an unusually calm system was examined in order to gain an uncluttered view on the pool manager's internals. The rate of process artifacts that persisted over a period of 24 h therefore indicate a theoretical upper bound, which should not be expected under real-world conditions. However, the experiment could be repeated under a continuous artificial load in order to simulate real-world conditions.

## REFERENCES

Chow Jim, Pfaff Ben, Garfinkel Tal, Rosenblum Mendel. Shredding your garbage: reducing data lifetime through secure deallocation. In: Proc. 14th USENIX security symposium, <http://footstool.stanford.edu/jchow/papers/usenixsec05/secdealloc-usenix05.pdf>; August 2005 [accessed 01.03.08].

Farmer Dan, Venema Wietse. Forensic discovery. Upper Saddle River, NJ, USA: Addison Wesley Professional; 2005.

fuzen_op. FU Rootkit, <https://www.rootkit.com/project.php?id=12>; 2005 [accessed 01.01.08].

IndigoSTAR Software. Perl2Exe. Mississauga, Ontario, Canada: IndigoSTAR, <http://www.indigostar.com/perl2exe.htm>; August 2007 [accessed 01.03.08].

Johnson Richard. Memory allocator attack and defense. In: ToorCon Seattle, <http://seattle.toorcon.org/talks/richardjohnson.pptx>; May 2007 [accessed 01.03.08].

Microsoft Corporation. Debugging tools for Windows, 32 Bit, Version 6.8.4.0. Redmond, WA, USA: Microsoft Corporation, <http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.8.4.0.msi>; October 2007 [accessed 01.03.08].

Microsoft Corporation. ExAllocatePoolWithTag. Redmond: Microsoft Corporation, <http://msdn2.microsoft.com/en-us/library/ms796989.aspx>; 2008 [accessed 01.03.08].

Russinovich Mark E, Salomon David A. Windows internals. Redmond, WA, USA: Microsoft Press; 2005. 4th ed.

Schuster Andreas. Searching for processes and threads in Microsoft Windows memory dumps. Digital Investigation September 2006a;3(suppl. 1):10–6. doi:10.1016/j.diin.2006.06.010.

Schuster Andreas. Pool allocations as an information source in windows memory forensics. In: Göbel Oliver, Schadt Dirk, Frings Sandra, Hase Hardo, Günther Detlef, Nedon Jens, editors. IT-incident management & IT-forensics – IMF 2006. Lecture notes in informatics, vol. P-97; 18 October 2006b. p. 104–15.

Schuster Andreas. PTFinder Version 0.3.05, <http://computer.forensikblog.de/en/2007/11/ptfinder_0_3_05.html>; November 2007a [accessed 01.03.08].

Schuster Andreas. PoolTools Version 1.3.0, <http://computer.forensikblog.de/en/2007/11/pooltools_1_3_0.html>; November 2007b [accessed 01.03.08].

SoBeIt. How to exploit Windows kernel memory pool. In: XCon, <http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_SoBeIt.pdf>; August 2005 [accessed 01.03.08].

Solomon Jason, Huebner Ewa, Bem Derek, Szeżynska Magdalena. User data persistence in physical memory. Digital Investigation 2007;4(2):68–72. doi:10.1016/j.diin.2007.03.002.

VMware, Inc. VMware products. Palo Alto, CA, USA: VMware, Inc., <http://www.vmware.com/products/>; 2008 [accessed 01.03.08].

Walters AAron, Petroni Nick L. Volatools: integrating volatile memory forensics into the digital investigation process. In: BlackHat DC, <http://www.cs.umd.edu/npetroni/papers/bh-fed-07-walters.pdf>; February 2007 [accessed 01.03.08].

Walters Aaron, Petroni Nick L. The volatility framework: volatile memory artifact extraction utility framework, <https://www.volatilesystems.com/VolatileWeb/volatility.gsp>; 2008 [accessed 28.02.08].

**Andreas Schuster** is a Senior Computer Forensic Examiner 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. He 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. Through times have significantly changed he 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.