J Comput Virol (2008) 4:83–100
DOI 10.1007/s11416-007-0070-0

SSTIC 2007 BEST ACADEMIC PAPERS

Windows memory forensics

Nicolas Ruff

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 2 October 2007 / Published online: 1 November 2007

Abstract This paper gives an overview of all known "live" memory collection techniques on a Windows system, and freely available memory analysis tools. Limitations and known anti-collection techniques will also be reviewed. Analysis techniques will be illustrated through some practical examples, drawn from past forensics challenges. This paper is forensics-oriented, but the information provided information will also be of interest to malware analysts fighting against stealth rootkits.

1 Introduction

"In memory only" intrusion came out from the lab to the field through the release of [Meterpreter] in 2004 [4]. Other tools (like Immunity [CANVAS] and Core [IMPACT]) [1,2] have been offering the same capability for a long time, but those tools were specialized and expensive. On the other hand, Metasploit is a freely available Open Source intrusion framework.

Whereas *nix "in memory" intrusion tools favor "syscall proxying" [3] techniques, most Windows tools seen to date rely on "in memory library (DLL) injection". This paper will focus on Windows systems exclusively.

"Classical" incident handling procedures (e.g., "pull the plug") and tools (e.g., [EnCase], [Sleuth Kit]) [51,52] are disk-oriented and rarely oriented towards "in memory" intrusion.

Intruders and malware writers are aware of this, and it is not uncommon nowadays to find encrypted malware that is decrypted "in memory only" through an externally (e.g., Web site) provided decryption key (for more information on cryptovirology, see [6–8]). In this case, "offline" analysis might be impossible if no key is available at the time of analysis.

That is why there has been a lot of interest in "live" forensics techniques in the last 3 years, starting with DFRWS 2005 [28] challenge.

This paper encompasses the following topics:

- "Live" memory collection tools;
- Analysis tools;
- "Real life" examples;
- Known anti-forensics techniques.

2 Live memory collection

Collecting "live" memory is not an easy task. By definition, this collection must be performed on a "live" system. In addition, it should maintain as small a footprint as possible on the system. A simple memory de-allocation, for instance, could trigger heap defragmentation, potentially overwriting valuable data.

Several techniques have been proposed for memory collection—none are perfect and none is suitable for all cases.

2.1 Hardware-based acquisition

One might think that this is the "silver bullet" of memory collection. The idea is to have dedicated hardware (e.g., a PCI card) in order to access physical memory through a dedicated communication port. Some players in the field of hardware memory acquisition are [Tribble] and [Komoku] [31,41].

N. Ruff (✉)
EADS-IW SE/CS, Suresnes, France
e-mail: nicolas.ruff@eads.net

Advantages:

- It has no impact on the "live" system—the OS is unaware of what is happening at the memory level.
- This solution is effective against most hiding techniques —however it is not 100% foolproof, as shown below.

We define a solution as "100% foolproof" if the attacker cannot cheat memory acquisition, even if he has access to all implementation details.

Drawbacks:

- Not available on the general market today.
- So heavily patent-protected that competitors are unlikely to show up in a near future.
- A savvy attacker could scan for specific hardware on the PCI bus.
- It is possible to "hide" parts of the system memory on the PCI bus, as demonstrated by Rutkowska (Joanna, [33]).
- The acquisition card has to be installed prior to intrusion. That is a major drawback, and that is why we need to think about other ways of collecting memory.

Rutkowska's hiding technique is based on NorthBridge chipset reprogramming. We will not go into gory details here (full paper available at Joanna [33]), as it requires a deep understanding of the PC architecture and micro-programming. It is enough to say that the PCI bus is managed by the SouthBridge chipset, and the core system (CPU + physical memory) by the NorthBridge. Communications between the PCI bus and the core system have to cross those two chipsets. For the sake of DMA access, the NorthBridge chipset can be used to "map" different views of the physical memory to peripherals.

Moreover, as virtualization gains popularity in the PC world, [IOMMU] chipsets [23] are expected to be integrated on mainstream PC motherboards, thus rendering memory hiding trivial.

Nothing more can be said about this solution, given the lack of any "off-the-shelf" product for forensic purpose. Some questions are left open, such as the handling of system activity during acquisition. If any part of the system is still running, the resulting memory data might be inconsistent.

2.2 Firewire bus

For systems that are not pre-equipped with a dedicated acquisition card, other available hardware should be considered, such as the IEEE 1394 (a.k.a. "FireWire") bus that allows direct memory access.

This technique has been initially documented as a way to hack into a system through the use of a modified [iPod] [32]. But it can be used to achieve many other goals, such as the one in which we are interested presently.

This technique is comparable to hardware-based acquisition, with some additional drawbacks:

- FireWire ports are not always available, especially on servers.
- System activity is not stopped during acquisition, resulting in potentially inconsistent data.
- This solution is not 100% foolproof, as it is still going through the PCI bus.
- Erratic behavior can be observed, especially when trying to access the [Upper Memory Area] [27].
- Direct memory access through FireWire is not enabled by default in Microsoft Windows operating system, contrary to Linux or Mac OS X.

[Adam Boileau] [35] demonstrated during Ruxcon 2006 conference that Windows access restriction is not applied to mass-storage devices, such as iPods, thus allowing direct memory access. However this *hack* is not guaranteed to be available across all Windows versions.

A Linux-oriented, forensics-targeted implementation has been proposed by [Piegdon] and [Pimenidis] [34].

Apart from FireWire, any hardware bus can potentially be used for physical memory access. The PCMCIA bus is a good candidate, as user-programmable, FPGA-based PCMCIA cards are available on the market (such as [PicoComputing]) [42].

2.3 "dd"& "nc" tools

"dd" and "nc" tools, available on G. M. Garner website [12], are tailored versions of the well-known *nix utilities.

Of all Windows-aware improvements, the one of interest to us is accessibility of the special device "\Device\PhysicalMemory" (see [17]).

Therefore, it is possible to dump the physical memory to a remote system through a simple command line such as:

```
nc -v -n -I\\.\PhysicalMemory <ip> <port>
```

This solution yields a very light footprint, with no prerequisites, and is readily available even to inexperienced collection operators.

Yet, some major drawbacks are to be found:

- Memory collection can last a long time (say several hours).
- As it is a user space (ring 3) solution, an attacker can hook several places in order to tamper with collected data.
- "\Device\PhysicalMemory" device is not available any more from user space since Windows 2003 SP1, and is not expected to be re-enabled by Microsoft in a near future.

2.4 "CrashDump" (keyboard-triggered)

Another solution that is quite unexpected at first thought is to crash the system.

When a "blue screen of death" (BSoD) occurs, the system records a crash dump file that is basically a dump of the physical memory, plus extra debugging information such as register values.

The output file, with a ".DMP" extension, is written in a Microsoft-proprietary file format, only legible to Microsoft debugging tools. However, this format has been partially reverse-engineered [18].

As one would expect, crashing Windows is quite easy. Crashes can be induced at will through a specific keyboard shortcut, when the following registry key [CrashOnCtrlScroll] is set to REG_DWORD:1 [19].

```
HKLM\SYSTEM\CurrentControlSet\Services\i8042prt\Parameters\CrashOnCtrlScroll
```

Enabling crash-control through registry

The magic key combination is "Right Ctrl + ScrollLock", pressed twice.

The CrashDump solution is the easiest and the fastest for acquiring memory of a "live" system. However, it has also many drawbacks:

- Some systems react poorly to system crashes—typically database servers. The system might be in an unstable state after crashing, which is sometimes unacceptable, even in case of a serious break-in.
- The registry setting requires a reboot to be taken into account. It must have been set prior to the incident. The ability to set this value *without reboot* will be studied later on.
- Windows stores CrashDump data into the pagefile (default "c:\pagefile.sys"). This implies that all pagefile content will be overwritten, and the pagefile size must be at least <physical memory size> + 1 MB (for status information).
- Given pagefile limits, the maximum dump file size is 2 GB. No complete dump can be obtained on a system with more than 2 GB of physical memory.

In recent Windows versions, this dump file size limit has been relaxed. As described in Knowledge Base article [Q237740] [21], using the undocumented "/MAXMEM" switch in BOOT.INI file, a dump over 2 GB in size can be generated. However, this parameter must have been set on boot. On systems with more than 4 GB of physical memory, the "/PAE" switch must have been set too.

Windows does support multiple pagefiles, up to 16 of 4 GB each. However, no publicly available forensics tool at the time of writing supports multiple pagefiles aggregation.

2.5 "CrashDump" (EMS-triggered)

Windows 2003 does have an interesting feature called EMS (*Emergency Management Services*). If this feature is enabled at boot time (through the "/REDIRECT" switch in BOOT.INI file), a serial console is available on COM1 port.

This console has a bunch of useful features, including the "crashdump" command. Besides the boot switch, no other parameter is required—the current CrashDump configuration is used. Error code is STOP 0x000000E2, just like for the previously described methods.

2.6 "Snapshot"

A very specific case is a virtualized host, running inside [VMWare] or [Virtual PC] software [59,60].

This case is close to the best possible case for forensics investigators. Physical memory image is written to disk when "pausing" the virtual machine. In VMWare's case, it is written to a ".vmem" file. It is also easy to mount the system drive read-only and recover the pagefile.

The whole system activity (including SMM and ACPI code) is frozen during acquisition. Thus, the acquired data is fully consistent, without any adverse effect on the running system (which can be shut down cleanly thereafter, if necessary).

The "only" requirement of this method is a virtualized target. For a long time, only test and *honeypot* systems were running virtualized. In the past few years, we noticed a strong interest in virtualization technologies, and it is not uncommon nowadays to find virtualized production systems (see [24]).

2.7 Pagefile issues

On a "live" system, part of the memory is swapped out into the pagefile. Collecting the pagefile is required for complete analysis.

The default pagefile is "c:\pagefile.sys". As seen before, it is possible to move the pagefile on any local disk (even a USB key with Windows Vista), and to have up to 16 pagefiles. Fortunately, this is still uncommon in real-life cases.

When a CrashDump file is generated, the disk space used by the pagefile is reclaimed and definitely lost. Knowledge Base article [Q886429] [22] details the process. Session Manager SubSystem [25] is in charge of pagefile management. During reboot, if a CrashDump has been previously generated, the following registry key is set by the SMSS:

```
HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\MachineCrash
```

Crash notification through registry

WinLogon checks for such a key. If found, the SAVEDUMP.EXE utility, which renames PAGEFILE.SYS to MEMORY.DMP, is run.

As long as Windows is running, the pagefile is locked by the kernel. It is still possible to access this file using a specially crafted driver, or the special device "\Device\PhysicalDrive". Incidentally, this technique has been used by Joanna Rutkowska to inject unsigned drivers in Windows Vista64 memory, up to RC1 release candidate (see [36]).

Commercial tools that are reputedly able to copy the pagefile of a running system are:

- [Disk Explorer] [53];
- [Forensic Toolkit] [54];
- [X-Ways Forensics] [55];
- [iLook] [56] (free for US government officials).

Those tools must be brought onto the target system, unless they were previously installed.

However, the cheapest and easiest way to collect the pagefile is to unplug the system.

After collecting physical memory, the idea is to unplug the system and access the pagefile through hard drive extraction. Since the system is still running during physical memory collection, the physical memory + pagefile combo is definitely not coherent. However, unless the system exhibits a dearth of physical memory, most of the collected data is useable.

The next step is to merge physical memory and pagefile data into a single set of data.

Without going too deep into the details of Intel x86 architecture, physical memory is addressed through 4 KB pages (or 4 MB, in some specific cases). A directory of available pages is stored in a two-level tree, referenced by special register CR3. First level of indirection is called *Page Directory,* whereas second level is called *Page Table*.

When a physical page has been "swapped out", the corresponding *Page Table Entry* (PTE) is flagged as "invalid".

Bits	31..12	11	10	9..5	4..1	0
Value	Offset	Transition	Prototype	Protection	PFN	Valid

PTE bit format

An invalid PTE has a *Page File Number* (PFN) field pointing to one of the 16 available pagefiles, and an Offset field serving as a page index into this specific pagefile. Since Offset is a 20-bit field, and each page is 4 KB wide, a single pagefile can store up to 4 GB of data.

More information is available in [Windows Internals] book [26], page 440 and later.

Technically speaking, merging physical memory with pagefiles does not pose a significant challenge, though no freely available tool does exist today to the best of our knowledge. [FATKit] [48] is one of the commercial tools that advertizes such a capability.

2.8 Hibernation file

Hibernation (also known as "suspend to disk") is a feature that allows the whole system state to be backed up to hard drive, thus allowing the system to be frozen for a (nearly) infinite amount of time without any power source.

Windows makes use of the "c:\hiberfil.sys" file to store system state. This file is created when the feature is enabled for the first time ("power options/enable hibernation").

It is unwise to enable this feature *after* an incident, since disk space at least equal to the size of physically available memory is pre-allocated, thus potentially overwriting interesting data on disk.

However, if hibernation has previously been used on the target, this memory collection method could prove to be a useful idea. It has several advantages, such as allowing coherent physical memory + pagefile acquisition.

Some technical issues are still being researched at the time of this writing:

- The hibernation file is using an undocumented, Microsoft proprietary file format, including proprietary compression. At the time of the writing, the EADS Innovation Works research center is close to having a full understanding of this file format.
- The hibernation file stores only committed memory, and not de-allocated memory pages.

The hibernation file could prove to be useful to rootkit detection. Its usefulness for forensics investigation is yet to be demonstrated.

2.9 Alternative OS injection

Bradley Schatz developed a brand new acquisition technique along with his [BodySnatcher] [39] proof-of-concept tool. The idea is to start a new OS on the top of the existing OS. This looks like a very promising technique, which overcomes the following limitations of previously shown memory acquisition techniques:

– Fidelity: since the existing OS is completely frozen during acquisition, a coherent memory snapshot is obtained.
– Reliability: since no existing OS facility is used (except by alternative OS loader components), this solution is resistant to software data hiding.
– Software based: no specific hardware is required on the target computer.

Some identified drawbacks are:

– Loader components for the alternative OS have a significant impact on the existing OS memory.
– A non-negligible amount of memory is used by the alternative OS (e.g., 32 MB in the author's sample), thus overwriting potential data in freed areas.
– The alternative OS has to support existing hardware on the target.
– Resistance to hardware data hiding (e.g., virtualization hardware, chipset reprogramming) is limited by the capabilities of the alternative OS.
– Technical limitations are yet to be lifted up, especially the resuming of the existing OS after acquisition.

3 Deeper into the CrashDump

3.1 Dynamic reconfiguration of keyboard driver

There are at least two issues with using "CrashOnCtrlScroll" feature "in the field":

1. It requires a reboot, if the registry key has not been previously set;
2. It only works for the Intel 8042 keyboard driver, which does exclude USB keyboards, for instance.

3.1.1 Issue #1: reloading the driver

Issue #1 is not trivial to bypass.

The "CrashOnCtrlScroll" registry key is read by I8xKeyboardServiceParameters() function, which is called by I8x KeyboardStartDevice(), itself called by I8xPnP() in response to 0x1b:0x00 (IRP_MJ_PNP:IRP_MN_START_DEVICE) IOCTL. The key value is copied into a dynamically allocated memory area by the ExAllocatePoolWithTag() function.

Despite the API name ("plug-and-play"), simply unplugging and plugging the keyboard in again will not reload the driver. No system event is generated when unplugging a PS/2 keyboard.

Some ideas to force a configuration reload are the following:

1. Send an IRP to i8042prt driver.

This solution is unstable and unsupported. The "i8042prt" driver initialization sequence is pretty complex; sending a new IRP_MN_START_DEVICE or IRP_MN_STOP_DEVICE message to the driver will lead to an immediate BSoD.

Microsoft documentation states explicitly that IRP_MJ_PNP messages are "reserved for system use" and never to be used by a programmer.

2. Uninstall/reinstall driver.

"net stop i8042prt" will not succeed, since the driver cannot be stopped, paused or restarted.

A complete uninstall/reinstall of the driver has not been investigated any further. Some APIs drawn from Windows "Plug-and-Play Manager" could be used for this task, such as SetupDiRemoveDevice() and SetupDiInstallDevice().

However, we speculate that whichever method is used to uninstall the driver, the driver will be marked as "to be deleted" on next reboot since it cannot be stopped.

3. Modify the configuration key "in memory"

This is a purely intellectual challenge: finding the dynamically allocated memory area where i8042prt configuration is stored.

It may not be well known that memory allocation in kernel space can be tagged with a "pool name" of 4 characters (function ExAllocatePoolWithTag()). A driver can tag every memory block that it uses, which is of great help for debugging purpose.

The "i8042prt" driver uses mostly "8042" and "Devi" (Device Manager) as pool tags.

Using [LiveKD] [44] debugger, it is possible to enumerate local kernel memory pools, and to find the right memory block to edit.

However we did not investigate live memory editing any further, for it has many drawbacks. It happens that memory block size and key offset into the block are Windows version dependent. A useable tool would require a database of every possible value pair, and would nevertheless be prone to false positives and potential BSoD.

In the end, no satisfactory solution has been found for issue #1. However this case does demonstrate why some Windows parameters require a reboot to be taken into account.

This issue can be fixed by Microsoft through callback notification of the "i8042prt" driver when the registry parameters are updated. While technically sound, this requires significant modification of the "i8042prt" source code.

3.1.2 Issue #2: unsupported USB keyboards

Issue #2 (unsupported USB keyboards) is crippling, especially in datacenters where not all servers have dedicated keyboards.

In the end, it seems like the best solution is not to rely on the "CrashOnCtrlScroll" feature, but to trigger a kernel bugcheck through some kind of minimalist driver. "i8042prt" implements the following code:

```
xor     ecx, ecx
push    ecx             ; BugCheckParameter4
push    ecx             ; BugCheckParameter3
push    ecx             ; BugCheckParameter2
push    ecx             ; BugCheckParameter1
push    MANUALLY_INITIATED_CRASH ; BugCheckCode = 0x0E
mov     [eax], ecx
call    ds:__imp__KeBugCheckEx@20 ; KeBugCheckEx(x,x,x,x,x)
```

Kernel bugcheck code, as implemented in i8042prt driver

This method has still one major drawback over the "CrashOnCtrlScroll" one: it requires administrative rights (or at least being able to load a driver).

SysInternals did publish a tool called "NotMyFault" that was able to trigger many kinds of different crashes. However this tool does not seem to be available anymore on SysInternals' new website at Microsoft's. Another tool with the same capability is [SystemDump] by Dmitry Vostokov [47].

3.2 CrashDump configuration

The CrashDump configuration can be edited through the following GUI path: "Control Panel/System/Advanced/Startup and Recovery/Parameters". Settings are stored in the following registry key:

```
HKLM\System\CurrentControlSet\Control\CrashControl
```

CrashDump configuration through registry

Values of interest to us are:

```
CrashDumpEnabled (REG_DWORD)
DumpFile (REG_EXPAND_SZ)
Overwrite (REG_DWORD)
```

Core registry values for CrashDump configuration

These values are documented in Knowledge Base article [20]. "Full Memory Dump" equals "CrashDumpEnabled" set to REG_DWORD:1.

Changes made through the Control Panel are taken into account immediately—they do not require a reboot. Let's see how this happens.

The "System" Control Panel applet is an executable file named "sysdm.cpl". When the Control Panel is closed, the internal function CoreDumpHandleOk() is called back. This function checks the "gfCoreDumpChanged" global variable. If the core dump configuration is to be updated, internal functions CoreDumpValidFile(), GetMemoryConfiguration() and CoreDumpGetRequiredFileSize() are called and do basic sanity checks.

Finally and critically, a NtSetSystemInformation() system call is invoked with SystemInformationClass = 34. This forces a dynamic update of the CrashDump kernel configuration.

From the kernel's point of view, the call sequence from NtSetSystemInformation() to registry keys is as follows:

- NtSetSystemInformation(),
- IoConfigureCrashDump(),
- IoInitializeCrashDump(),
- IopInitializeDCB(),
- IopReadDumpRegistry().

With all this information, it is trivial to write a command-line tool that updates the CrashDump configuration.

3.3 Pagefile configuration

The pagefile configuration can be edited through the following GUI path: "Control Panel/System/Advanced/Performance/Parameters/Advanced". Settings are stored in the following registry key:

```
HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management
```

Pagefile configuration through registry

The value that is of interest to us is:

```
PagingFiles (REG_MULTI_SZ)
```

Core registry value for pagefile configuration

This value holds a list of paging files, in following format:

```
<filename> <min size> <max size>
<filename> <min size> <max size>
...
```

Pagefile configuration registry data format (REG_MULTI_SZ)

Under the hood, the NtCreatePagingFile() API is involved.

As the registry key name suggests, SMSS (Session Manager SubSystem) process is in charge of pagefile initialization. After a configuration change, Control Panel applet calls NtCreatePagingFile() from VirtualMemCreatePagefileFromIndex() internal function.

The SeCreatePagefilePrivilege right is required for this operation.

As will be shown later on, it is possible to enumerate active pagefiles, including use percentages, through simple scripting.

3.4 Windows management instrumentation (WMI)-based configuration

Most of the previous tasks can be done using Windows native scripting tools. In my experience, most Windows users do not realize the full potential of Windows scripting tools, which are really powerful, yet user-friendly.

Focusing on WMI capabilities, the CrashDump configuration can be edited through Win32_OSRecoveryConfiguration class, such as through the following VBScript CrashDump.vbs:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\CIMV2")
Set colItems = objWMIService.ExecQuery( _
    "SELECT * FROM Win32_OSRecoveryConfiguration",,48)
For Each objItem in colItems
    Wscript.Echo "-----------------------------------"
    Wscript.Echo "Win32_OSRecoveryConfiguration instance"
    Wscript.Echo "-----------------------------------"
    Wscript.Echo "AutoReboot: " & objItem.AutoReboot
    Wscript.Echo "Caption: " & objItem.Caption
    Wscript.Echo "DebugFilePath: " & objItem.DebugFilePath
    Wscript.Echo "DebugInfoType: " & objItem.DebugInfoType
    Wscript.Echo "Description: " & objItem.Description
    Wscript.Echo "ExpandedDebugFilePath: " & objItem.ExpandedDebugFilePath
    Wscript.Echo "ExpandedMiniDumpDirectory: " & objItem.ExpandedMiniDumpDirectory
    Wscript.Echo "KernelDumpOnly: " & objItem.KernelDumpOnly
    Wscript.Echo "MiniDumpDirectory: " & objItem.MiniDumpDirectory
    Wscript.Echo "Name: " & objItem.Name
    Wscript.Echo "OverwriteExistingDebugFile: " & objItem.OverwriteExistingDebugFile
    Wscript.Echo "SendAdminAlert: " & objItem.SendAdminAlert
    Wscript.Echo "SettingID: " & objItem.SettingID
    Wscript.Echo "WriteDebugInfo: " & objItem.WriteDebugInfo
    Wscript.Echo "WriteToSystemLog: " & objItem.WriteToSystemLog
Next
```

CrashDump.vbs script for retrieving CrashDump configuration

Running on Windows XP SP2, the sample script output is:

```
C:\> cscript CrashDump.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001.

-----------------------------------
Win32_OSRecoveryConfiguration instance
-----------------------------------
AutoReboot: True
Caption:
DebugFilePath: %SystemRoot%\MEMORY.DMP
DebugInfoType: 3
Description:
ExpandedDebugFilePath: C:\WINDOWS\MEMORY.DMP
ExpandedMiniDumpDirectory: C:\WINDOWS\Minidump
KernelDumpOnly: False
MiniDumpDirectory: %SystemRoot%\Minidump
Name: Microsoft Windows XP Professional|C:\WINDOWS|\Device\Harddisk0\Partition1
OverwriteExistingDebugFile: True
SendAdminAlert: False
SettingID:
WriteDebugInfo: True
WriteToSystemLog: True
```

CrashDump.vbs sample output

The same result can be achieved through WMIC (WMI Command line utility). For some unobvious reason, WMIC aliases cannot be mapped one-to-one to WMI classes. The CrashDump configuration is accessed in WMIC using following command:

```
C:\> wmic

wmic:root\cli>recoveros list full

AutoReboot=TRUE
DebugFilePath=%SystemRoot%\MEMORY.DMP
Description=
KernelDumpOnly=FALSE
Name=Microsoft Windows XP Professional|C:\WINDOWS|\Device\Harddisk0\Partition1
OverwriteExistingDebugFile=TRUE
SendAdminAlert=FALSE
SettingID=
WriteDebugInfo=TRUE
WriteToSystemLog=TRUE
```

WMIC sample output (CrashDump configuration)

Along similar lines, the pagefile configuration is exposed through the following classes:

- Win32_PageFile,
- Win32_PageFileElementSetting,
- Win32_PageFileSetting,
- Win32_PageFileUsage.

Sample pagefile configuration read:

```
wmic:root\cli>pagefile list full

AllocatedBaseSize=2046
CurrentUsage=8
Description=C:\pagefile.sys
InstallDate=20061129104253.031250+060
Name=C:\pagefile.sys
PeakUsage=8
Status=
TempPageFile=
```

WMIC sample output (pagefile configuration)

```
wmic:root\cli>pagefileset list full

Description='pagefile.sys' @ C:\
InitialSize=2046
MaximumSize=4092
Name=C:\pagefile.sys
SettingID=pagefile.sys @ C:
```

WMIC sample output (active pagefiles)

Win32_PageFileUsage is also available through VBScripting. Here is a sample output.

```
AllocatedBaseSize: 512
Caption: C:\pagefile.sys
CurrentUsage: 247
Description: C:\pagefile.sys
InstallDate: 31/08/2006 13:49:34
Name: C:\pagefile.sys
PeakUsage: 319
Status:
TempPageFile:
```

VBScript sample output (active pagefiles + usage)

Unfortunately, there is no obvious way to access the hibernation configuration, either through WMIC or VBScripting. Raw access to the following registry keys seems to be the only available solution:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Power\*
```

Power management configuration through registry

Those keys are binary blobs. Hibernation configuration is held in the 7th byte of the "Heuristics" value.

Note: the POWERCFG command is quite limited as far as hibernation is concerned. The only available commands are "POWERCFG/HIBERNATE ON" and "POWERCFG/HIBERNATE OFF". There is no easy way to get the current hibernation configuration through this command.

4 Memory analysis tools

4.1 Virtual memory reconstruction

Having collected a physical memory dump is only the first step of the job. Working on a raw memory dump yields at least two challenges: finding interesting data structures, and analyzing those structures (which are mostly undocumented by Microsoft).

Intel [Pentium] manuals [15] are a good start for understanding physical memory management. Briefly, physical memory is split into 4 KB chunks called "pages". Through the use of paging, processes have a "virtual" memory view, which is unrelated to physical memory organization. On a system with 1 GB of physical memory, we have just gathered 262,144 pages in no particular order . . . Reassembling those parts is the next step of memory analysis.

Andreas [Schuster] [9] released to the public a tool called [PTFinder] [45] for this particular task. The idea behind the tool is to scan the memory dump in search of a well-known, critical system structure called EPROCESS. Given some basic sanity checks, most false positives are avoided.

As of Version 0.03.01-XP-SP2, the following checks are made against the EPROCESS structure:

- Page Directory base is not null.
- Page Directory base is aligned on a 4KB boundary.
- Thread list pointers (forward and backward) are in kernel space.
- Synchronization event are to be found at offsets 0xD8 and 0xFC.

Kernel space starts at 0x80000000 if Windows has been started without the /3 GB switch, 0xC0000000 otherwise.

Each process has an associated EPROCESS structure, holding a copy of CR3 register, which is the physical address of the process Page Directory.

A detailed description of Windows memory scanning techniques and challenges by the author of PTFinder himself can be found at [DFRWS 2006] [40].

This is a very clever approach, since it can also:

- Identify terminated processes (as long as physical memory is not reallocated).
- Identify running processes that are trying to hide through DKOM-like (Direct Kernel Object Manipulation) techniques, as long as the associated EPROCESS structure has been unlinked but not been wiped out.

This approach also has some drawbacks:

- It is not 100% foolproof—this can be an issue when used for legal purpose. But foolproof memory forensics has yet to be invented.
- It requires a thorough understanding of Windows internal structures, which are undocumented and Service Pack-dependent. Yet most structures are to be found in Microsoft [Debugging Tools] [16] and can be recovered through the "dt" command.

```
kd> .reload
Connected to Windows 2000 2195 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
...............................................................................
Loading User Symbols

Loading unloaded module list
...............

kd> dt _EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ExitStatus       : Int4B
   +0x070 LockEvent        : _KEVENT
   +0x080 LockCount        : Uint4B
   +0x088 CreateTime       : _LARGE_INTEGER
   +0x090 ExitTime         : _LARGE_INTEGER
   +0x098 LockOwner        : Ptr32 _KTHREAD
   +0x09c UniqueProcessId  : Ptr32 Void
   +0x0a0 ActiveProcessLinks : _LIST_ENTRY
   +0x0a8 QuotaPeakPoolUsage : [2] Uint4B
   +0x0b0 QuotaPoolUsage   : [2] Uint4B
   +0x0b8 PagefileUsage    : Uint4B
   +0x0bc CommitCharge     : Uint4B
   +0x0c0 PeakPagefileUsage : Uint4B
   +0x0c4 PeakVirtualSize  : Uint4B
   +0x0c8 VirtualSize      : Uint4B
   +0x0d0 Vm               : _MMSUPPORT
   [...]
```

EPROCESS structure on Windows 2000 SP4, extracted from Microsoft Kernel Debugger (KD)

[PTFinder] [45] is not alone: there are others like [Volatools] [46]. However most publicly released memory analysis tools seem to rely on EPROCESS structures.

Joe Stewart published a tool called [pmodump.pl] [50] that relies on a Windows implementation trick in order to find Page Directories: one of the Page Directory entries is self-referencing. This tool seems to produce accurate results, as well.

5 Post-intrusion forensics samples

5.1 Meterpreter evidence

Meterpreter is a "memory only" intrusion tool, part of the Metasploit project since Version 2.2 (released in August 2004). It has been designed from scratch by Jarkko Turkulainen and Matt Miller. An associated research paper is also available: [Library Injection] [30].

The basic idea is to inject a dynamically loaded library (DLL) in the target process memory space without writing anything to disk. To do so, the following exported functions of NTDLL.DLL are hooked in the target process:

- NtOpenSection(),
- NtQueryAttributesFile(),
- NtOpenFile(),
- NtCreateSection(),
- NtMapViewOfSection().

With only those five hooks, the LoadLibrary() API is tricked to load a memory area instead of a file. All these operations happen purely in user space. No kernel trick is involved.

Technical details are available in sources ("external/source/meterpreter" subdirectory of Metasploit framework). The code is of remarkably good quality and well documented.

Let's take the case of a Windows 2000 SP4 English system, successfully penetrated by a Metasploit + Meterpreter combination. After gathering a CrashDump, we are going to manually analyze the memory dump with the help of Microsoft Debugging Tools.

First step is to load the dump file into the debugger and enumerate processes.

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 81841380  SessionId: 0  Cid: 0008    Peb: 00000000  ParentCid: 0000
    DirBase: 00030000  ObjectTable: 81841e68  TableSize: 155.
    Image: System

PROCESS 816a86c0  SessionId: 0  Cid: 00a0    Peb: 7ffdf000  ParentCid: 0008
    DirBase: 0291f000  ObjectTable: 816a1c68  TableSize:  33.
    Image: SMSS.EXE

PROCESS 8168a140  SessionId: 0  Cid: 00b8    Peb: 7ffdf000  ParentCid: 00a0
    DirBase: 037fe000  ObjectTable: 8168a488  TableSize: 335.
    Image: CSRSS.EXE

PROCESS 81683820  SessionId: 0  Cid: 00b4    Peb: 7ffdf000  ParentCid: 00a0
    DirBase: 03bc3000  ObjectTable: 81683c88  TableSize: 374.
    Image: WINLOGON.EXE

PROCESS 81671020  SessionId: 0  Cid: 00e8    Peb: 7ffdf000  ParentCid: 00b4
    DirBase: 03f63000  ObjectTable: 816725e8  TableSize: 505.
    Image: SERVICES.EXE

PROCESS 816705c0  SessionId: 0  Cid: 00f4    Peb: 7ffdf000  ParentCid: 00b4
    DirBase: 03f2b000  ObjectTable: 81670b28  TableSize: 271.
    Image: LSASS.EXE

PROCESS 8164ab40  SessionId: 0  Cid: 01b4    Peb: 7ffdf000  ParentCid: 00e8
    DirBase: 04a6b000  ObjectTable: 8164ae28  TableSize: 262.
    Image: svchost.exe
[…]
```

Active processes dump through the kernel debugger

Before analyzing a specific process, we have to retrieve its execution context. Let's take for example the SVCHOST.EXE process, of PID 0x1B4, with an EPROCESS structure stored at address 0x8164ab40.

```
kd> .process 8164ab40
Implicit process is now 8164ab40
```

Changing process context (from the kernel debugger point of view)
Note: this does not affect the current process from the operating system point of view

From now on, it is possible to dig into that process context. Let's display a (shortened) list of loaded libraries, using PEB (Process Environment Block) information.

```
kd> !peb
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 71f40 . 8fa28
    Ldr.InLoadOrderModuleList: 71ec0 . 8fa18
    Ldr.InMemoryOrderModuleList: 71ec8 . 8fa20
            Base TimeStamp                     Module
         1000000 3814ad86 Oct 25 21:20:38 1999 C:\WINNT\system32\svchost.exe
        77f80000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\ntdll.dll
[…]
        775a0000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\CLBCATQ.DLL
        10000000 439e49c1 Dec 13 05:10:41 2005 C:\WINNT\system32\metsrv.dll
          c30000 4435ac71 Apr 07 02:04:01 2006 C:\WINNT\system32\ext635732.dll
    SubSystemData:     0
    ProcessHeap:       70000
    ProcessParameters: 20000
        WindowTitle:  'C:\WINNT\system32\svchost.exe'
        ImageFile:    'C:\WINNT\system32\svchost.exe'
        CommandLine:  'C:\WINNT\system32\svchost -k rpcss'
        DllPath:
'C:\WINNT\system32;.;C:\WINNT\system32;C:\WINNT\system;C:\WINNT;C:\WINNT\system32;C:\WIN N
T;C:\WINNT\System32\Wbem'
        Environment:  0x10000
```

Dumping the Process Environment Block from the current debugger process context

Detailed information can be recovered with the help of "!dlls" command.

```
kd> !dlls

0x00071ec0: C:\WINNT\system32\svchost.exe
      Base   0x01000000  EntryPoint  0x010010b8  Size        0x00005000
      Flags  0x00005000  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_LOAD_IN_PROGRESS
             LDRP_ENTRY_PROCESSED

0x00071f30: C:\WINNT\system32\ntdll.dll
      Base   0x77f80000  EntryPoint  0x00000000  Size        0x0007b000
      Flags  0x00004004  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED

[…]

0x000bfbb8: C:\WINNT\system32\CLBCATQ.DLL
      Base   0x775a0000  EntryPoint  0x7760f150  Size        0x00086000
      Flags  0x000c4004  LoadCount   0x00000001  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_DONT_CALL_FOR_THREADS
             LDRP_PROCESS_ATTACH_CALLED

0x0009caf0: C:\WINNT\system32\metsrv.dll
      Base   0x10000000  EntryPoint  0x10004a73  Size        0x00013000
      Flags  0x002c4004  LoadCount   0x00000002  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_DONT_CALL_FOR_THREADS
             LDRP_PROCESS_ATTACH_CALLED
             LDRP_IMAGE_NOT_AT_BASE

0x0008fa18: C:\WINNT\system32\ext635732.dll
      Base   0x00c30000  EntryPoint  0x00c36068  Size        0x00023000
      Flags  0x00284004  LoadCount   0x00000001  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED
             LDRP_IMAGE_NOT_AT_BASE
```

Dumping loaded DLLs from the current debugger process context

The last two DLLs are part of Meterpreter. They are easily spotted by their names.

In this example, we went directly to the result. In a "real life" investigation, identifying the faulting process is a much more tedious task. Meterpreter libraries could also have been renamed. Microsoft Debugging Tools quickly show their limitations:

- Terminated processes are not easily available—and in most cases, remote exploitation of a security flaw has killed the faulting process.
- Complex tasks, such as enumerating libraries in each process context, require the use of scripting. Microsoft Debugging Tools PERL-like scripting language is very awkward; a Python-like high-level interface would simplify matters considerably.

Here is a sample script, drawn from [Dump Analysis] website [13]:

```
$$
$$ Enumerating processes
$$

r $t0 = nt!PsActiveProcessHead

.for (r $t1 = poi(@$t0); (@$t1 != 0) & (@$t1 != @$t0); r $t1 = poi(@$t1))
{
  .catch {
      r? $t2 = #CONTAINING_RECORD(@$t1, nt!_EPROCESS, ActiveProcessLinks);
      .process @$t2
      .reload
      !peb
  }
}
```

WinDbg script to gather PEBs from all running processes

This script is close to unintelligible to a non-specialist. Note the ".catch" directive, which is required to avoid debugger close in case of a script error.

This script outputs the following log:

```
kd> $$><script.txt
Implicit process is now 81841380
Loading Kernel Symbols
...............................................................................
Loading User Symbols

Loading unloaded module list
.................
PEB at 00000000
    *** unable to read PEB
Implicit process is now 816a86c0
Loading Kernel Symbols
...............................................................................
Loading User Symbols
...
Loading unloaded module list
.................
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         48580000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 161f40 . 1627a0
    Ldr.InLoadOrderModuleList: 161ec0 . 162790
    Ldr.InMemoryOrderModuleList: 161ec8 . 162798
            Base TimeStamp                     Module
        48580000 3d5cebca Aug 16 14:10:50 2002 \SystemRoot\System32\smss.exe
        77f80000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\ntdll.dll
        68010000 3ef27500 Jun 20 04:44:16 2003 C:\WINNT\System32\sfcfiles.dll
    SubSystemData:     0
    ProcessHeap:       160000
    ProcessParameters: 110000
        WindowTitle:  '(null)'
        ImageFile:    '\SystemRoot\System32\smss.exe'
        CommandLine:  '\SystemRoot\System32\smss.exe'
        DllPath:      'C:\WINNT\System32'
        Environment:  0x100000
Implicit process is now 8168a140
Loading Kernel Symbols
...............................................................................
Loading User Symbols
........
Loading unloaded module list
.................
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         5FFF0000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 161f40 . 162fb0
    Ldr.InLoadOrderModuleList: 161ec0 . 163188
    Ldr.InMemoryOrderModuleList: 161ec8 . 163190
            Base TimeStamp                     Module
        5fff0000 3ef2750b Jun 20 04:44:27 2003 \??\C:\WINNT\system32\csrss.exe
        77f80000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\ntdll.dll
        5ff90000 3ef274e9 Jun 20 04:43:53 2003 C:\WINNT\system32\CSRSRV.dll
        5ffa0000 3ef274e6 Jun 20 04:43:50 2003 C:\WINNT\system32\basesrv.dll
        5ffb0000 3ef27505 Jun 20 04:44:21 2003 C:\WINNT\system32\winsrv.dll
        77e10000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\USER32.DLL
        7c4e0000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\KERNEL32.DLL
        77f40000 3ef274dc Jun 20 04:43:40 2003 C:\WINNT\system32\GDI32.DLL
    SubSystemData:     0
    ProcessHeap:       160000
    ProcessParameters: 110000
        WindowTitle:  '(null)'
        ImageFile:    '\??\C:\WINNT\system32\csrss.exe'
        CommandLine:  'C:\WINNT\system32\csrss.exe ObjectDirectory=\Windows
SharedSection=1024,3072,512 Windows=On SubSystemType=Windows ServerDll=basesrv,1
ServerDll=winsrv:UserServerDllInitialization,3
ServerDll=winsrv:ConServerDllInitialization,2 ProfileControl=Off MaxRequestThreads=16'
        DllPath:
'C:\WINNT\system32;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem'
        Environment:  0x100000

[…]
```

WinDbg script output

– It would have been nice to be able to *dump* each loaded binary, for integrity checking purposes. Microsoft Debugging Tools only offer basic primitives such as memory read. A full *dumper* could be designed as a debugger plug-in—however, writing plug-ins is a very tedious task given the absence of documentation and the complexity of the

API. Just imagine SDK samples not compiling cleanly by default, requiring additional #defines.

```
kd> d 0x01000000
01000000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
01000010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
01000020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
01000030  00 00 00 00 00 00 00 00-00 00 00 00 c0 00 00 00  ................
01000040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
01000050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
01000060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
01000070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

METSRV.DLL library header

```
kd> d 0xc30000
00c30000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
00c30010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
00c30020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00c30030  00 00 00 00 00 00 00 00-00 00 00 00 f8 00 00 00  ................
00c30040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
00c30050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
00c30060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
00c30070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

EXT635732.DLL library header

We can conclude that Microsoft Debugging Tools are the most "universal" tools, since they work with any version of Windows. But they are also the hardest to use for forensic analysis, as we will see by comparison to other tools.

5.2 Securitech challenge 2005

The [Securitech] challenge [29] is a renowned French security challenge, held annually in June. In 2005, Kostya Kortchinsky submitted a memory analysis challenge as challenge #16. The objectives were to recover the following elements from a physical memory dump:

1. *Microsoft security bulletin identifier of the exploited flaw on the target (e.g., MS01-123);*
2. *source IP address of the attack;*
3. *Windows local administrator password on the target;*
4. *output of the "validnivo" program, that has been uploaded and executed on the target.*

The target was running Windows 2000 SP4.

This challenge was held in optimal conditions for the analyst, since the pagefile was deactivated and physical memory was collected by the means of a VMWare snapshot. Physical memory size was 256 MB.

However, the filesystem was not available as part of the challenge, thus raising the bar for question #4. Rebuilding the binary file was possible through the network queue or the remaining memory of the terminated process.

[PTFinder] [45] gives the following information.

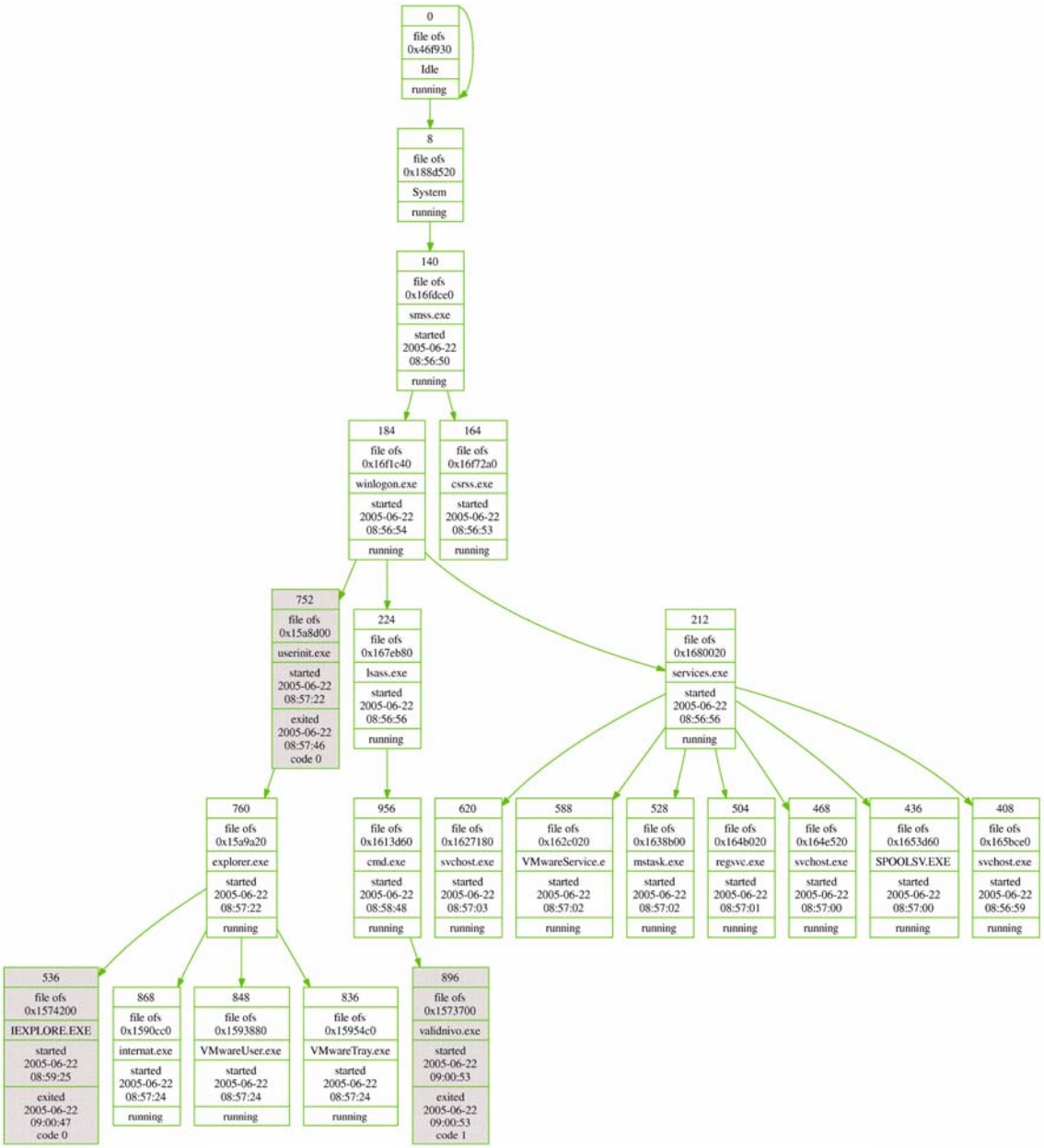
```
C:\>perl ptfinder_w2k.pl --nothreads --dotfile output-no-thread.dot ..\win2000pro.vmem

No.  Type PID    TID    Time created        Time exited         Offset     PDB        Remarks
---- ---- ------ ------ ------------------- ------------------- ---------- ---------- --------------
   1 Proc      0                                                0x0046f930 0x00030000 Idle
   2 Proc    896        2005-06-22 09:00:53 2005-06-22 09:00:53 0x01573700 0x0a8c6000 validnivo.exe
   3 Proc    536        2005-06-22 08:59:25 2005-06-22 09:00:47 0x01574200 0x093e7000 IEXPLORE.EXE
   4 Proc    868        2005-06-22 08:57:24                     0x01590cc0 0x071cf000 internat.exe
   5 Proc    848        2005-06-22 08:57:24                     0x01593880 0x06d3f000 VMwareUser.exe
   6 Proc    836        2005-06-22 08:57:24                     0x015954c0 0x06c73000 VMwareTray.exe
   7 Proc    752        2005-06-22 08:57:22 2005-06-22 08:57:46 0x015a8d00 0x06086000 userinit.exe
   8 Proc    760        2005-06-22 08:57:22                     0x015a9a20 0x06092000 explorer.exe
   9 Proc    956        2005-06-22 08:58:48                     0x01613d60 0x07de1000 cmd.exe
  10 Proc    620        2005-06-22 08:57:03                     0x01627180 0x04ac5000 svchost.exe
  11 Proc    588        2005-06-22 08:57:02                     0x0162c020 0x04b3c000 VMwareService.exe
  12 Proc    528        2005-06-22 08:57:02                     0x01638b00 0x047b8000 mstask.exe
  13 Proc    504        2005-06-22 08:57:01                     0x0164b020 0x04575000 regsvc.exe
  14 Proc    468        2005-06-22 08:57:00                     0x0164e520 0x040dc000 svchost.exe
  15 Proc    436        2005-06-22 08:57:00                     0x01653d60 0x04211000 SPOOLSV.EXE
  16 Proc    408        2005-06-22 08:56:59                     0x0165bce0 0x040c9000 svchost.exe
  17 Proc    224        2005-06-22 08:56:56                     0x0167eb80 0x03cd8000 lsass.exe
  18 Proc    212        2005-06-22 08:56:56                     0x01680020 0x03d0f000 services.exe
  19 Proc    184        2005-06-22 08:56:54                     0x016f1c40 0x03a97000 winlogon.exe
  20 Proc    164        2005-06-22 08:56:53                     0x016f72a0 0x037d2000 csrss.exe
  21 Proc    140        2005-06-22 08:56:50                     0x016fdce0 0x029c6000 smss.exe
  22 Proc      8                                                0x0188d520 0x00030000 System
```

PTFinder tool running against Securitech Challenge (text view)

[PTFinder] [45] also gives a visual view of the process tree, based on Parent PID (PPID). Grayed blocks are terminated processes. VALIDNIVO.EXE process stands out, with CMD.EXE as a parent, which in turn has LSASS.EXE as a parent (which is extremely suspicious, since all user processes are expected to be launched from EXPLORER.EXE).

MemDump will run successfully against parent CMD.EXE with the following result:



PTFinder tool running against Securitech Challenge (graph view)

The VALIDNIVO.EXE process and its Page Directory have been found. Unfortunately, the process itself cannot be reconstructed with the MemDump tool, since the Page Directory has been damaged.

```
C:\> perl memdump.pl win2000pro.vmem 0x07de1000
Reading page directory at file offset 0x7de1000... done.

C:\> type 0x7de1000.map
 virt. addr.  file offset     size
------------ ------------ --------
     0x10000            0   0x1000
     0x20000       0x1000   0x1000
    0x12e000       0x2000   0x1000
    0x12f000       0x3000   0x1000
    0x130000       0x4000   0x1000
[…]
  0x4ad00000      0x12000   0x1000
  0x4ad01000      0x13000   0x1000
  0x4ad04000      0x14000   0x1000
[…]
  0x7ffde000      0x8c000   0x1000
  0x7ffdf000      0x8d000   0x1000
  0x7ffe0000      0x8e000   0x1000
  0xc0000000      0x8f000   0x1000
  0xc0001000      0x90000   0x1000
  0xc012b000      0x91000   0x1000
[…]
```

MemDump tool running against Securitech Challenge

"0x7de1000.mem" file stores the whole committed memory of the CMD.EXE process, including executable file, libraries, dynamically allocated memory and kernel related data. The process could be wholly reconstructed from that file, given the usual process dumping issues (such as Import Table reconstruction).

It is quite easy to spot that this CMD.EXE process was a "Metasploit Courtesy Shell" (default window title of a Metasploit-created command shell).

```
C:\> strings -o 0x7de1000.mem | grep -i metasploit
5504:Metasploit Courtesy Shell (TM)
31696:Metasploit Courtesy Shell (TM)
37368:Metasploit Courtesy Shell (TM)
50844:Metasploit Courtesy Shell (TM) - v
29288:Metasploit Courtesy Shell (TM)
```

Locating the Metasploit shell with grep

The MemParser tool, by Chris Betz, gives same results than [PTFinder] [45].

```
C:\> memparser.exe win2000pro.vmem

MemParser v1.3 Chris Betz, (c) 2005
No process list loaded.
In Windows 2000 Mode
Options:
1:      Load the process list
<enter>:        Quit
1

Searching for processes in memory dump
00%--05%--10%--15%--20%--25%--30%--35%--40%--45%--50%--55%--60%--65%--70%--75%--80%--85% -
--90%--95%--100%
Enumerating process structures.
Sorting processes by PID
Checking for processes hidden by detachment from process link-list or processes no longer
active
Searching for all threads.
MemParser v1.3 Chris Betz, (c) 2005
Process List:
 Proc#          PPID          PID            InProcList             Name:
Threads:
   0             0             0            Yes                    Idle
   1             0             8            Yes                    System
   2             8           140            Yes                    smss.exe
   3           140           164            Yes                    csrss.exe
   4           140           184            Yes                    winlogon.exe
   5           184           212            Yes                    services.exe
   6           184           224            Yes                    lsass.exe
   7           212           408            Yes                    svchost.exe
   8           212           436            Yes                    SPOOLSV.EXE
   9           212           468            Yes                    svchost.exe
  10           212           504            Yes                    regsvc.exe
  11           212           528            Yes                    mstask.exe
  12           212           588            Yes                    VMwareService.e
  13           212           620            Yes                    svchost.exe
  14           752           760            Yes                    explorer.exe
  15           760           836            Yes                    VMwareTray.exe
  16           760           848            Yes                    VMwareUser.exe
  17           760           868            Yes                    internat.exe
  18           224           956            Yes                    cmd.exe
  19             0         29718073             No
  20             0         29718073             No
  21             0         29718073             No
In Windows 2000 Mode
Options:
#:      Select a process
s:      Show System Information
<enter>:        Quit
18

956: cmd.exe selected:
1       Dump Process Memory (No System Memory Included) to Disk
2       Dump Process Memory (Including System Memory Space) to Disk
3       Dump Process Strings (No System Memory Included) to Disk
4       Dump Process Strings (Including System Memory Space) to Disk (Takes a long time)
5       Display Process Environment Information
6       Display all DLLs loaded by process
<enter>:        quit
5
Process Environment Information:
        Executable File:        C:\WINNT\system32\cmd.exe
        Command Line:   cmd.exe
        Window Title:   Metasploit Courtesy Shell (TM)
        Desktop Info:   WinSta0\Default
        Shell Info:
        Runtime Data:
        Dll Path:
C:\WINNT\system32;.;C:\WINNT\system32;C:\WINNT\system;C:\WINNT;C:\WINNT\System32
```

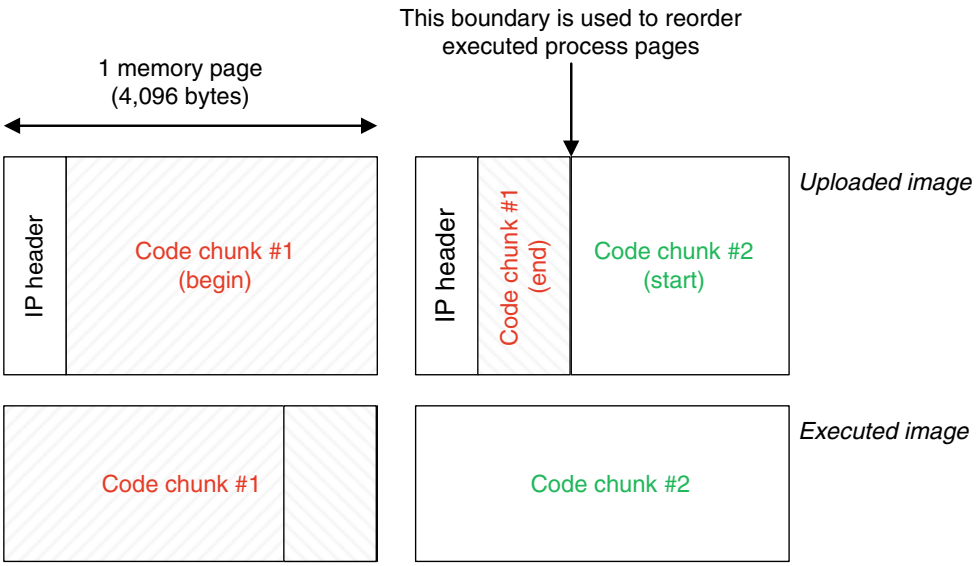
MemParser tool running against Securitech Challenge

In the end, the rebuilding of VALIDNIVO.EXE process has been worked out "by hand".

Since the same binary data has been uploaded and executed, every piece of data lies in memory at least twice. Given a different alignment of network buffers and executed binary file, it was possible to manually rejoin every piece of data. The whole binary was about 10 pages of 4 KB each.

Two new tools were designed during the challenge:

- [MemParser] [49], available on SourceForge, which combines PTFinder and MemDump functions in a single tool. Written in C, this tool is much faster than equivalent PERL scripts available from Andreas Schuster.
- [KnTTools] [58], which is commercially available today.



Manual reconstruction of VALIDNIVO.EXE: how to assemble pieces together

This challenge is a good example of "real life" issues when dealing with dead processes.

5.3 DFRWS 2005 challenge

The DFRWS 2005 challenge principles were similar to the Securitech Challenge. The Target system was also Windows 2000. The following questions were posed:

1. *What hidden processes were running on the system, and how were they hidden?*
2. *What other evidence of the intrusion can be extracted from the memory dumps?*
3. *Why did "plist.exe" and "fport.exe" not work on the compromized system?*
4. *Was the intruder specifically seeking Professor Goatboy's research materials?*
5. *Did the intruder obtain the Professor's research?*
6. *What computer was the intrusion launched from?*
7. *Is there any indication of who the intruder might be?*

All user-contributed solutions are available on the web site of [DFRWS] challenge [28]. The general techniques stay the same, but the published papers are extremely detailed and cannot be summed up easily in this article.

6 Counter measures

Memory forensics is still a very young field of research. However some people have already focused on counter-analysis techniques.

The field of memory hiding techniques has been largely explored by rootkit authors, since anti-rootkits are mostly based on live memory scanning.

Hardware anti-analysis, such as NorthBridge reconfiguration, has already been described before. Given the complexity of the ever-evolving PC architecture, other hardware-based anti-analysis techniques are expected to be seen in a near future, such as firmware reprogramming or [SMM] code [37].

Software based anti-analysis techniques are numerous and well-known:

- A trivial but still efficient technique is to block kernel space communications, such as "\Device\PhysicalMemory" or driver loading.

This will hamper most software-based analysis tools, such as "dd". Of course, a single forgotten entry point is enough to bypass the protection. However it can still delay memory collection, if the collection agent is not skilled enough or is not expected to deviate from standard procedures.

– In rootkit history, Direct Kernel Object Manipulation (DKOM) is the oldest known hiding technique. It consists in unlinking carefully chosen objects (such as an EPROCESS token) from kernel-managed lists.

This technique is also the easiest to spot: a thread that does not belong to any process is not common on a sane system.

– Another technique is to create a new thread in an existing process.

In user space, EXPLORER.EXE and IEXPLORE.EXE are common targets. In kernel space, NULL.SYS driver is often targeted. Malicious code can be injected into a dynamically allocated memory area or inside code cavities (which is stealthier—see [5]).

– "Split TLBs" is another very powerful technique that is available for code hiding.

This technique has been used for years by PaX protection for Linux. The basic idea is to desynchronize code and data Translation Lookaside Buffers (TLBs)—which are basically caches for virtual address translation. When such caches are not in sync, read and execute accesses at the same virtual address will yield different results. A functional proof-of-concept for Windows does exist: [Shadow Walker] [38].

With such a protection in place, "dd"-like tools are fooled: read memory is not consistent with actually executed code. The easiest way to get around the protection is to have a driver flush caches before any page read operation.

Given all these advanced hiding techniques available "off the shelf", Meterpreter seems to be the easiest "memory only" intrusion tool to detect.

7 Conclusion

Despite intense research activity, live physical memory collection and analysis on Windows operating system is still in early stages of practical deployment.

No 100% foolproof collection technique has been invented to date—even hardware acquisition cards may be fooled by data hiding. A tradeoff between data coherency, target availability, timeframe for memory acquisition has to be determined on a case-by-case basis.

Given a random, out-of-the-box system, options for the collection operator are quite limited: access to "PhysicalMemory" device, or CrashDump. Both require driver installations on the target.

Publicly and freely available tools for the analyst are also limited and require customization (e.g., hard coded addresses). It is not generally known how advanced much commercial and government-restricted tools are and what capabilities they possess. The most "mysterious" tool of them all may be [WOLF] (Windows OnLine Forensics) by Microsoft [57].

Nevertheless, memory analysis is still able to recover valuable information that would have been otherwise "wiped out" by the classical "power off" forensics procedure. Known available tools for "memory only" intrusion, such as the Meterpreter, can be easily spotted.

It is our opinion that no forensics analyst may disregard live memory analysis. It should be performed as a complement to traditional disk-based analysis and will become more and more valuable as available tools grow more sophisticated and forthcoming tools are run against older memory dumps.

References

Instrusion

1. Immunity [CANVAS] http://www.immunitysec.com/products-canvas.shtml
2. Core [IMPACT] http://www.coresecurity.com/products/coreimpact/index.php
3. [Syscall Proxying] http://www.coresecurity.com/files/files/11/SyscallProxying.pdf
4. Metasploit's [Meterpreter] http://www.metasploit.com/projects/Framework/docs/meterpreter.pdf
5. Ultimate way to hide [rootkit] https://www.rootkit.com/newsread.php?newsid=648

Cryptovirology

6. Fred [Raynal] "Malicious cryptography" *Part one:* http://www.securityfocus.com/infocus/1865 *Part two*: http://www.securityfocus.com/infocus/1866
7. Éric [Filiol] "Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the Bradley virus" Proceedings of the 14th EICAR Conference, 2005 http://papers.weburb.dk/archive/00000136/01/eicar05final.pdf
8. [Malicious Cryptography] http://www.cryptovirology.com/

Blogs

9. Andreas [Schuster] http://computer.forensikblog.de/en/
10. Windows Incident Response http://windowsir.blogspot.com/
11. Mariusz Burdach http://forensic.seccure.net/
12. George M. [Garner] http://users.erols.com/gmgarner/forensics/
13. [Dump Analysis] http://www.dumpanalysis.org/
14. Alexandre Garaud http://c4rtman.blogspot.com/

Documentations

15. [Pentium] Intel® 64 and IA-32 Architectures Software Developer's Manuals http://www.intel.com/products/processor/manuals/index.htm

16. [Debugging Tools] for Windows http://www.microsoft.com/whdc/devtools/debugging/default.mspx
17. \Device\[PhysicalMemory] http://technet2.microsoft.com/WindowsServer/en/library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true
18. [DMP] File Structure http://computer.forensikblog.de/en/2006/03/dmp_file_structure.html
19. [CrashOnCtrlScroll] Windows feature lets you generate a memory dump file by using the keyboard http://support.microsoft.com/kb/244139
20. [Q254649] Overview of memory dump file options for Windows Server 2003, Windows XP, and Windows 2000 http://support.microsoft.com/kb/254649
21. [Q237740] How to overcome the 4,095 MB paging file size limit in Windows http://support.microsoft.com/kb/237740
22. [Q886429] What to consider when you configure a new location for memory dump files in Windows Server 2003 http://support.microsoft.com/kb/886429
23. [IOMMU] http://en.wikipedia.org/wiki/IOMMU
24. Virtualization Services [Market] to Reach $11.7 Billion by 2011, According to IDC http://www.idc.com/getdoc.jsp?containerId=prUS20778407
25. [SMSS] Session Manager SubSystem http://en.wikipedia.org/wiki/Session_Manager_Subsystem Mark E. Russinovich and David A. Solomon
26. [Windows Internals], 4th edn. http://www.microsoft.com/mspress/books/6710.aspx
27. [Upper Memory Area] Memory dumping over FireWire—UMA issues http://ntsecurity.nu/onmymind/2006/2006-09-02.html

Challenges

28. [DFRWS] 2005 Challenge http://www.dfrws.org/2005/challenge/index.html
29. [Securitech] 2005, Challenge 16 http://www.challenge-securitech.com/archives/2005/displaylevel.php?level=21

Conference Materials

30. Remote [Library Injection] http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf
31. [Tribble] "A Hardware-Based Memory Acquisition Procedure for Digital Investigations" http://www.digital-evidence.org/papers/tribble-preprint.pdf
32. [iPod] "Firewire—all your memory are belong to us" http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf
33. Joanna [Rutkowska] "Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)" http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt
34. David [Piegdon] and Lexi [Pimenidis] "Targeting Physically Addressable Memory" http://david.piegdon.de/papers/SEAT1394-svn-r432-paper.pdf
35. [Adam Boileau] "Hit by a Bus: Physical Access Attacks with Firewire" http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf
36. Joanna Rutkowska [Subverting Vista Kernel] http://invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt
37. Loïc Duflot [SMM] Security Issues Related to Pentium System Management Mode http://cansecwest.com/slides06/csw06-duflot.ppt
38. Sherri Sparks, Jamie Butler [Shadow Walker]: Raising the Bar for Rootkit Detection http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf
39. Bradley Schatz [BodySnatcher]: Towards reliable volatile memory acquisition by software https://www.dfrws.org/2007/proceedings/p126-schatz.pdf
40. Andreas Schuster [DFRWS 2006] Searching for processes and threads in Microsoft Windows memory dumps http://dfrws.org/2006/proceedings/2-Schuster.pdf

Specialized companies

41. [Komoku] http://www.komoku.com/
42. [PicoComputing] http://www.picocomputing.com/
43. [Lexfo] http://www.lexfo.fr/

Free tools

44. [LiveKD] http://www.microsoft.com/technet/sysinternals/SystemInformation/LiveKd.mspx
45. [PTFinder] 0.3.0 http://computer.forensikblog.de/en/2006/09/ptfinder_0_3_00.html
46. [Volatools] http://www.komoku.com/forensics/basic.html
47. [SystemDump] http://citrite.org/blogs/dmitryv/2006/09/12/new-systemdump-tool/
48. [FATKit] http://www.4tphi.net/fatkit/
49. [MemParser] http://sourceforge.net/projects/memparser
50. [pmodump.pl] and the Truman Project http://www.secureworks.com/research/tools/truman.html

Commercial and or private forensics tools

51. Guidance Software: [EnCase] Forensics http://www.guidancesoftware.com/products/ef_index.asp
52. The [Sleuth Kit] & Autopsy: Digital Investigation Tools for Linux and other Unixes http://www.sleuthkit.org/
53. [Disk Explorer] http://www.runtime.org/
54. [Forensic Toolkit] http://www.accessdata.com/catalog/partdetail.aspx?partno=11000
55. [X-Ways Forensics] http://www.x-ways.net/forensics/index-m.html
56. [iLook] http://www.ilook-forensics.org/
57. [WOLF] http://blogs.technet.com/robert_hensing/archive/2005/01/17/354471.aspx
58. [KnTTools] http://users.erols.com/gmgarner/KnTTools/

Other software

59. [VMWare] http://www.vmware.com/
60. Microsoft [Virtual PC] http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx