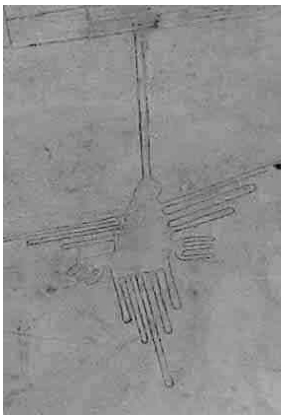


An Introduction to Windows memory forensic

Mariusz Burdach
http://forensic.seccure.net

July 9, 2005

Table of content:

1. An introduction

In this paper I address the problem of finding important kernel structures in physical memory images collected from compromised Windows machines. Steps described in this paper can be used to configure a beta version of simple toolkit: "*Windows memory forensic toolkit*" (currently, only on Linux). Using this toolkit we can simplify the process of a physical memory analysis. I would like to emphasize that procedures described in this document have to be treated as a kind of introduction to very challenging process of "Windows memory analysis".

The memory image analyzed in this paper was created by using the *netcat* tool. This tool is included in package "*Forensic Acquisition Utilities*" created by George M. Garner Jr. In fact, bit-by-bit copy image was created by using the *dd* tool. It is important because a memory image generated by coredump function has different structure and can be easily analyzed by tools like Windows Kernel Debugger. Analysis of an image created by the *dd* tool is a little bit complicated.

On compromised machine the following command was executed to create a mirror of a physical memory: **nc -v -n -l\\.\PhysicalMemory** <*ip address of destination machine*> <*port number of destination machine*>.

The physical memory was gathered from Windows 2003 Standard Edition (with no Service Packs installed).

Currently, methods used by *Windows memory forensic toolkit* can be defeated by tools using technique DKOM. This technique allows to modify kernel structures through unlinking for example: _EPROCESS block from the list of active processes. Such process is almost "invisible" but still can be executed by the Windows Scheduler. To detect hidden processes it is necessary to read other internal kernel lists. This scenario is not discussed in this document.

This document outlines techniques that allow digital investigators to extract some information from a physical memory image of compromised Windows machine.

2. Finding important symbols – offline methods

2.1 Finding strings in a memory image

The simplest way to find some important symbols in a memory image is to use the *grep* command. This technique is perfect if we have no knowledge about version of analyzed memory image. If the version of analyzed dump is known, we can install the new system, run the Windows Debugger and find interesting addresses of internal kernel structures. It is important to know that addresses of symbols are changing from version to version including Service Packs.

It is quite easy to find strings in a memory dump. There are a lot of them in an image. The biggest challenge is to select the most unique strings.

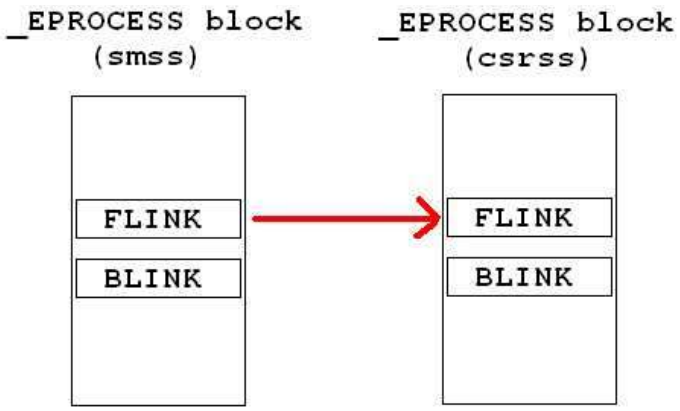
As an example let's examine processes which can be found in each MS Windows 2000/XP/2003. Processes like System, smss, csrss, lsass are active in each Windows operating system. Let's find out how many "smss" strings can be found in the memory image of Windows 2003.

```
C:\hexdump memory_image | grep smss | wc –l
3
```

It is surprising that only three "smss" strings were detected! Please keep in mint that this result can be inaccurate because such string can be broken into two lines. But after performing several tests I think that this method is still useful and allows to find proper information.

2.1.1 Defining the special offset and finding an address of the PsActiveProcessHead

To localize an address of the PsActiveProcessHead we must find addresses of some _EPROCESS blocks of active processes. It is enough to find two such blocks. But we have to be sure that blocks points at each other. Excellent candidates are smss and csrss processes. In most Windows 2000/XP/2003 the FLINK field of the _EPROCESS block of one process (smss) points to the FLINK field of the next process on the list (csrss). Described situation is shown below.



In first step we have to find the beginning of the _EPROCESS block of the smss process. After searching the "smss" string I received the following result.

```
01baa7c0    00 00 00 00 00 00 00 00 00 00 00 00 73 6d 73 73    | ……….smss|
01baa7d0    2e 65 78 65 00 00 00 00 00 00 00 00 00 00 00 00    |.exe……...|
```

It means that the physical address of the _EPROCESS block is 0x01baa678. As is described in chapter 3 the offset to the beginning of _EPROCESS block is 0x154.

For the "csrss" string the result is presented below.

```
01c18430    00 00 00 00 00 00 00 00 00 90 d1 f9 63 73 72 73    |……….csrs|
01c18440    73 2e 65 78 65 00 00 00 00 00 00 00 00 00 00 00    |s.exe……. |
```

It means that the physical address of the _EPROCESS block is 0x01c182e8.

Now, it is necessary to read values stored in the ActiveProcessLinks struct. This struct contains only two elements. The ActiveProcessLinks structure is shown below.

```
typedef struct _LIST_ENTRY ActiveProcessLinks {
DWORD Flink;
DWORD Blink; }
```

The offset from the beginning of the _EPROCESS block to the beginning of the ActiveProcessLinks is 0x88. The smss _EPROCESS block is at 0x1baa700. The csrss _EPROCESS block is at 0x1c18370. The FLINK field of the smss _EPROCESS block points to the address 0x81818370. This is the virtual address of the csrss _EPROCESS block. After removing the kernel offset from this address we have the address: 0x01818370. This is the address of the FLINK field stored in the csrss _EPROCESS block. As we can see this result is different from the result received a few steps above. This is the key point because we received an important offset. Let's call it the special offset. This offset have to be added to any address read from the memory image. This special offset is received after using the following formula: 0x1c18370 - 0x01818370 = 0x0400000. It is worth to mention that this address can be different on other versions of Windows. Also Service Packs can influence on the value of this special offset.

A simple proof-of-concept "Windows memory forensic toolkit" is added to this paper. For example the *eprocwin* tool is used to enumerate all _EPROCESS blocks from the memory image. Before compiling

this tool it is necessary set the proper value of the OFFSETW. This value is the special offset received after performing steps described above.

Now, we can find an address of the PsActiveProcessHead. In first step we have to read the address stored in the BLINK field of the smss _EPROCESS block. As we calculated above, the address of the ActiveProcessLinks is 0x1baa700. The content of this struct is shown below.

```
01baa700    70 83 81 81 70 4a 93 81 80 02 00 00 c0 14 00 00    | p…pJ…….|
```

The address 0x81934a70 points to the ActiveProcessLinks of the System _EPROCESS block. Please keep in mind that in this case the System _EPROCESS block is just before the smss _EPROCESS block.
To receive the physical address we have to add the special offset. The content of this physical address is presented below:

```
01d34a70    00 a7 7a 81 e8 ed 56 80 00 00 00 00 00 00 00 00    |..z…V……|
```

At 0x8056ede8 is the address of the PsActiveProcessHead.

A procedure of enumerating threads is omitted in this document. But each _EPROCESS block contains a field which points to list of process threads.

2.1.2 Finding a base address and an address of the PsLoadedModuleList

To find an address of the PsLoadedModuleList I use the same method as above. One important thing is that a driver's name is stored in a physical memory in the UNICODE format. For example, the "ntos" string in the memory looks as follows: 6e 00 74 00 6f 00 73 00. If we use any hex editor to open a physical image file we will see that the string for the UNICODE format looks like that: n.t.o.s.. Searching the "o.s.k.r.n " string in the image allows me to find about one hundred places in the memory image. But after simple verification only few addresses store the full name of the file - ntoskrnl.exe. The second address which store the full file name is 0x01d65400. In the driver block, the offset from the driver's name to the beginning of the block is 0x4c. At offset 0x0 is placed the address of the next driver's block. At offset 0x4 is placed the address of the previous driver's block. At offset 0x30 is placed the address of the char table which stores the name of the driver. The most important offset is 0x18. This is the address where the binary file was mapped. This address is key for our investigation, because this is the base address which has to be added to RVA's which can be extracted from the ntoskrnl.exe file. In most cases we have an access to the file system of compromised machine so files like ntoskrnl.exe or ntdll.dll can be very useful.

In the same way we can find other addresses in the physical memory where the drivers are allocated.

As we know, the ntoskrnl.exe is the operating system and the first module on the list of loaded modules. So at offset 0x4 is the address of the PsLoadedModuleList. The memory area which contains driver's block is shown below.

```
01d653b0    48 53 96 81 08 8c 56 80  58 70 53 80 13 00 00 00    |HS....V.XpS.....|
01d653c0    00 00 00 00 00 00 00 00  00 e0 4d 80 e6 d7 6c 80    |..........M...l.|
01d653d0    00 50 23 00 3c 00 3c 00  08 00 00 e1 18 00 18 00    |.P#.<.<.........|
01d653e0    fc 53 96 81 00 40 00 0c  01 00 00 00 00 00 00 00    |.S...@..........|
01d653f0    ff 4e 22 00 00 00 00 00  00 00 00 00 6e 00 74 00    |.N".........n.t.|
```

```
01d65400        6f 00 73 00 6b 00 72 00  6e 00 6c 00 2e 00 65 00        |o.s.k.r.n.l...e.|
01d65410        78 00 65 00 00 00 00 00  0e 00 20 0a 4d 6d 20 20        |x.e..... .Mm  |
```

At 0x80568c08 is the PsLoadedModuleList.
At 0x804de000 is the mapped ntoskrnl.exe file. This is the base address.

2.2 Analyzing kernel image and related libraries

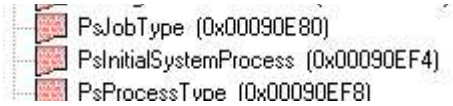
In the kernel image file (ntoskrnl.exe) there is a Debug section. Here we can find some interesting symbols. Also by disassembling functions from the ntoskrnl.exe it is possible to find interesting information about internal structures of the dumped memory image.

For tested version of Windows, an offset between RVA's of some symbols exported by the mapped operating system (the kernel image file) and the real address in the memory is 0x80000000 (kernel space) + 0x004de000 (the base address). The method of finding the base address is described in chapter 2.1.2. The formula RVA + the base address is true for symbols allocated in the system space. These symbols have to be exported by the kernel image file (ntoskrnl.exe). In the Debug section we can find all symbols with corresponding RVA's. In this document I only consider the situation where the System Space (non-PAE) is 2 GB and the page size is 4 KB. Critical kernel structures of the operating system image are mapped and available in kernel space between virtual addresses: 0x80000000 and 0xa0000000.

Now, I discuss one example of using symbol's address to find kernel structure in the image.

2.2.1 Localizing the address of the System process.

As we can see at picture below, in kernel image the PsInitialSystemProcess symbol is at RVA equal to 0x00090ef4.



After adding the base address to this RVA, we will receive the virtual address of the PsInitialSystemProcess symbol. At this address is stored the address of the _EPROCESS block for the System process.
Before we can localize the beginning to the _EPROCESS block of the System process in the memory image we have to find the address of the PsInitialSymbolProcess. To find this address it is enough to add to the RVA (0x90ef4) the base address which is 0x004de000. The result is 0x56eef4. Under this address is stored the address of the _EPROCESS block. The content of the memory image is presented below.

```
0056eef0        00 00 f4 77 e8 49 93 81  40 40 93 81 70 4e 93 81        |...w.I..@@..pN..|
0056ef00        18 0a 00 e1 02 00 00 00  00 00 00 00 00 00 00 00        |................|
```

Now, we have to add to this address (0x819349e8) the special offset 0x400000. We have to add this offset each time when we want to localize structures in the memory image which were created by the operating system. This offset is constant for each version of operating system. In this case where the Windows 2003 Standard Edition is analyzed this offset is 0x400000. There are at least two methods of finding this special offset. The first one is an address translation used by the Windows operating

system. This method is described in next chapter. The second method is based on comparing addresses. This method is described in chapter 2.1.1.

To confirm that method described above is correct I used the Windows Debugger to localize the _EPROCESS block of the System process.

```
kd> ?PsActiveProcessHead
Evaluate expression: -2141786648 = 8056ede8
kd> dd 8056ede8 L2
8056ede8   81934a70 8183f9d0
kd> dd 81934a70-88 L1
819349e8 001b0003
```

```
kd>?PsInitialSystemProcess
Evaluation expression: -2141786380 = 8056eef4
kd> dd 8056eef4 L1
8056eef4 819349e8
```

As we can see the address of the _EPROCESS block of the System process is correct.

2.2.2 Address Translation

This process is performed automatically by the MMU. More details about the various types of translations are described in book ***Microsoft Windows Internals, Fourth Edition: Microsoft 2003, Windows XP, and Windows 2000***. Here only basic steps are described.

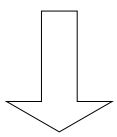
Let's start from the point, that we know values stored in the ActiveProcessLinks of the smss _EPROCESS block. The content of this part of the _EPROCESS block received from the physical memory image is shown below.

```
01baa700        70 83 81 81 70 4a 93 81 80 02 00 00 c0 14 00 00        | p…pJ…….|
```

the FLINK contains: 0x81818370
the BLINK contains: 0x81934a70

It is not enough to remove the kernel offset (0x80000000) from this address to receive the physical address. We have to translate this address manually to physical address. Windows 2003 uses a two-level page table structure to translate addresses. This is the simplest situation and the most popular. Let's analyze the BLINK address. The BLINK address in the binary format is presented as:

Page directory index



1000000110 0100110100 101001110000

Step one:
First, the page directory index will be used to locate the page table in which the virtual address is located. The physical address of the page directory base is at 0x39000 (tested on Windows 2003 and XP). To this address we have to add the page directory index which is equal to 206. Each entry in the page directory has 0x4 bytes. The beginning of the page table index is at 0x39818.

At this address we find the value 0x01c001e3 as is presented below:

```
00039810     e3 01 40 01 e3 01 80 01 e3 01 c0 01 63 21 db 01  |..@.........c!..|
```

Step two:
To this Page Director Entry address we must add the 0xffc00000 value as is shown below.

```
  0000000111 0000000000 000111100011
 +1111111111 0000000000 000000000000
-------------------------------------
  0000000111 0000000000 000000000000
```

The result is 0x1c00000

Step three:
Now, we have to add to the BLINK address the 0x3fffff as is presented below:

```
  1000000110  0100110100  101001110000
 +0000000000  1111111111  111111111111
--------------------------------------
  0000000000  0100110100  101001110000
```

The last step is to add this value to address received in step two (0x134A70 + 0x1c00000 = 1D34A70). The address 0x1d34a70 is the physical address of the BLINK field.

3. Other useful information

I consider that we have an access to the file system of compromised machine. It allows us to use original system files like ntoskrnl.exe to extract some useful information. Next, this information can be used to analyze the physical memory image.

For example the by disassembling the function PsGetProcessImageFileName we can find the offset to _EPROCESS ImageFileName field.

```
Disassembly of Debug Symbol PsGetProcessImageFileName   (0x004449B4)
;*****************************************************************************
; *** PsGetProcessImageFileName (902) ***
SYM:PsGetProcessImageFileName
0x4449B4: 8B442404       mov      eax,dword ptr [esp+0x4]
0x4449B8: 0554010000     add      eax,0x154
0x4449BD: C20400         ret      0x4
;*****************************************************************************
```

As we can see the offset from the beginning of the _EPROCESS block is 0x154.

In Debug section of ntoskrnl.exe file there is the symbol KiServiceDescriptorTable with corresponding RVA. After adding the base address we can localize a place in physical memory image where the address of the System Service Table is stored. Next, we can find addresses of internal system calls.

4. References

1. Mark E. Russinovich, David A. Solomon, “Microsoft Windows Internals, Fourth Edition: Microsoft 2003, Windows XP, and Windows 2000”.
2. Forensic Acquisition Utilities. http://users.erols.com/gmgarner/forensics/.
3. Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

Current version of “Windows memory forensic toolkit” is available at http://forensic.seccure.net.