Windows Vista 32bits and unexported kernel symbols.

Matthieu Suiche, *Senior Security Fanatics!*
<matt@msuiche.net>
http://www.msuiche.net

January 31, 2007

Abstract: Numerous tools exist to detect Rootkits through different tricks (e.g. Rootkit Revealer [1], and so on) and some protections only work on 64-bits architecture (e.g. Patchguard [3][4]). Anyways, this one has been publicly announced as broken by *Authentium* [5].

Actually, most of these Rookits work on 32-bits architectures, more especially under Windows 2000, XP and 2003. Detection of Rookits is still a hard race between hackers and security researchers.

Although Rootkits' hiding mechanisms (e.g. SSDT, IDT, MSR, and System Structures) are still the same, there are no standalone solutions provided which can take advantage of the full range of resource that the kernel allows.

Furthermore, this article doesn't provide any introduction to Interrupts and Exceptions mechanisms. Anyways some advanced references are available in the last section.

This paper can be considered similar to my previous article about Windows Vista 64-bits [6].

Introduction: This article doesn't talk about a method to localize the SSDT, because the method published by "90210" still works under Windows Vista 32-bits. The goal of this article is to introduce two new ways of authenticity checking for the IDT and the Syscall MSRs.

The reader should notice that those tricks are also working under Windows 2000, XP and 2003. For, Windows Vista's kernel scheme is very similar to its previous version.

Prerequisites: Deprived of access to documentation and source code, we analysed Windows Vista 32-bits RTM version using an external disassembler, and the latest Debugging Tools for Windows (32-bits) to have a CPL 0 debugger compatible with Microsoft Windows Vista 32-bits. Of course, a strong assembly understanding is necessary here.

I. System Interruptions

Once upon a time near the edge of the lost kernel there was a funny function called "KiSystemStartup".

During December 2006, I published a proof of concept of IDTGuard which runs only on Windows 2000 and XP for technical reasons [7] because it uses the \PhysicalMemory object. The ingenuity of IDTGuard is to localize and to use the in-raw IDT inside the kernel as a fingerprint.

This "fingerprinting" becomes very useful when we comparing two structures by their theoretical entries. The difficulty of this trick is to rebuild correctly all the original entries.

Why? Ntoskrnl makes further self-modification after copying the in-raw IDT. Additionally, the HAL's dll (=Hardware Abstraction Layer) do also further modification using the KPCR structure to access the IDT.

Therefore, our goal can only be reached by a three-step stair.

- Find the original entry inside the kernel
- Find self modifications from Ntoskrnl.exe
- And then others from Hal.dll

Here is my decompiling of the main part of KiSystemStartup:

```
//
// Intialize the FS segment to 0x30, this segment is use inside
// the KiInitializePcr() function.
//
_asm {
    push 0x30
    pop fs
}
// [...]

//
// Returns GdtBase, IdtBase, Pcr, TssBase
// After calling this function the idt base address is stored
// into a local variable.
//
GetMachineBootPointers();

// [...]

IdtBase[DOUBLE_FAULT]->Type = 0x85;
IdtBase[DOUBLE_FAULT]->Selector = 0x50;

IdtBase[NMI_NPX_ERROR]->Type = 0x85;
IdtBase[NMI_NPX_ERROR]->Selector = 0x58;

// [...]

//
// Initialize the PCR structure
// The "IdtBase" variable will be stored into the KPCR structure,
// This is a very important part to understand how HAL get the
```

```
    // IDT base address.
    //
    KiInitializePcr(KeNbrProcessors,
                    Pcr,
                    IdtBase,
                    GdtBase,
                    TssBase,
                    &KiInitialThread,
                    &KiDoubleFaultStack);

    //  [...]

    //
    // The in-raw structure of the IDT uses a very different scheme as
    // we know. This is the most important part of the function that helps
    // us to understand how to read the in-raw IDT.
    //
    KiSwapIDT();

    //  [...]

    //
    // Copy RawIdt into memory and restore DoubleFault and NmiNpxError
    // interrupts, which have been copied before.
    //
    _asm
    {
        push dword ptr IdtBase[DOUBLE_FAULT].dw00
        push dword ptr IdtBase[DOUBLE_FAULT].dw04
        push dword ptr IdtBase[NMI_NPX_ERROR].dw00
        push dword ptr IdtBase[NMI_NPX_ERROR].dw04
    }

    //
    // This is the part where "rep movsd" opcode is copied.
    //
    RtlFillMemoryUlong(IdtBase,
                       RawIdtBase,
                       0x200);

    //
    // Restoring the Int 2 and 8.
    //
    _asm
    {
        pop dword ptr IdtBase[NMI_NPX_ERROR].dw04
        pop dword ptr IdtBase[NMI_NPX_ERROR].dw00
        pop dword ptr IdtBase[DOUBLE_FAULT].dw04
        pop dword ptr IdtBase[DOUBLE_FAULT].dw00
    }

    //  [...]
```

Here, the disassembling of the main part of the GetMachineBootPointers() function.

```
    _asm {
        sgdt    [ebp+kgdt]
        mov     edi, dword ptr [ebp+kgdt+2]
        mov     cx, fs                               ; fs = 0x30
        and     cx, 0FFFCh
        movzx   ecx, cx
        add     ecx, edi                            ; @GdtEntry + 0x30
        mov     dh, [ecx+7]
        mov     dl, [ecx+4]
        shl     edx, 10h
        mov     dx, [ecx+2]
        mov     esi, edx                            ; esi = PcrBase

    // [...]

        sidt    [ebp+kidt]
        mov     eax, dword ptr [ebp+kidt+2]         ; eax = IdtBase
    }
```

And here, the decompiling of main part of KiInitializePcr().
For reminding the PCR structure looks like:

```
typedef struct _KPCR {
        KPCR_TIB Tib;                               // +0x000
        PKPCR SelfPcr;                              // +0x01C
        PKPRCB Prcb;                                // +0x020
        KIRQL Irql;                                 // +0x024
        ULONG IRR;                                  // +0x028
        ULONG IrrActive;                            // +0x02C
        ULONG IDR;                                  // +0x030
        PVOID KdVersionBlock;                       // +0x034
        PKIDTENTRY IDT;                             // +0x038
        PKGDTENTRY GDT;                             // +0x03C
        PKTSSENTRY TSS;                             // +0x040
        USHORT MajorVersion;                        // +0x044
        USHORT MinorVersion;                        // +0x046
        KAFFINITY SetMember;                        // +0x048
        ULONG StallScaleFactor;                     // +0x04C
        UCHAR SpareUnused;                          // +0x050
        UCHAR Number;                               // +0x051
        UCHAR Spare0;                               // +0x052
        UCHAR SecondLevelCacheAssociativity;        // +0x053
        UINT VdmAlert;                              // +0x054
        UINT KernelReserved[14];                    // +0x058
        UINT SecondLevelCacheSize;                  // +0x090
        UINT HalReserved[16];                       // +0x094
        UINT InterruptMode;                         // +0x0d4
        UCHAR Spare1;                               // +0x0d8
        UINT KernelReserved2[17];                   // +0x0dc
        KPRCB PrcbData;                             // +0x120
} KPCR, *PKPCR;
```

Here is the decompiling.

```
        // [...]

        //
        // PCR structure Initialization
        //
        Pcr.GDT = GDTBase;          // mov     [eax+3Ch], ecx
        Pcr.IDT = IDTBase;          // mov     [eax+38h], ecx
        Pcr.TSS = TSSBase;          // mov     [eax+40h], ecx

        // [...]
```

1. ***Step one:*** *Flight to the original IDT entries.*

This is the part of the code we have to look for while scanning the code. It's an inline function so it's very easy to recognize a "rep movsd" opcode, which has only one occurrence is the whole kernel.

```
    //
    // This is the part where "rep movsd" opcode is copied.
    //
    RtlFillMemoryUlong(IdtBase,
                       RawIdtBase,
                       0x200);
```

Then, we look for this part of code :

```
8B 45 XX             mov     edi, [ebp+IdtBase]
BE F4 A1 70 00       mov     esi, offset RawIdtBase
B9 00 08 00 00       mov     ecx, 2048
C1 E9 02             shr     ecx, 2
F3 A5                rep movsd
```

These red bytes are unchanged since Windows 2000, then with this 10bytes signature it's very easy to get a pointer to "mov esi, offset RawIdtBase".
And then, we can easily recover the dword "0x0070A1F4" which is a pointer to the RawIdtBase.

```
RawIdtBase          dd offset _KiTrap00
                    dd 88E00h
                    dd offset _KiTrap01
                    dd 88E00h
                    dd offset _KiTrap02
                    dd 88E00h
                    dd offset _KiTrap03
                    dd 8EE00h
                    [...]
                    dd offset _KiGetTickCount
                    dd 8EE00h
                    dd offset _KiCallbackReturn
                    dd 8EE00h
                    dd offset _KiRaiseAssertion
                    dd 8EE00h
                    dd offset _KiDebugService  ; 0x2D
                    dd 8EE00h
                    dd offset _KiSystemService ; 0x2E
                    dd 8EE00h
                    [...]
```

As you can see, there are some very interesting interrupts like: all x86 exception interrupts and more especially the 0x2D and 0x2E interrupts.

After having analysed the KiSystemStartup() and KiSwapIDT(), we can rebuild the structure of the in-raw IDT.

```
typedef struct _KIDT_RAW_ENTRY32 {
    union {
    ULONG Offset;
        struct {
        USHORT OffsetLow;
```

```
            USHORT OffsetHigh;
            };
        };
        UCHAR Reserved;
        UCHAR Type:4;
        UCHAR Always0:1;
        UCHAR Dpl:2;
        UCHAR Present:1;
        UCHAR Selector;
} *PKIDT_RAW_ENTRY32, KIDT_RAW_ENTRY32;
```

2. ***Step two****: A boat near the self-modification.*

These two following interrupts are different from others exception interrupts because they are task gates and not interrupt gates. That’s why they don’t have any offset.

```
    IdtBase[DOUBLE_FAULT]->Type = 0x85;
    IdtBase[DOUBLE_FAULT]->Selector = 0x50;

    IdtBase[NMI_NPX_ERROR]->Type = 0x85;
    IdtBase[NMI_NPX_ERROR]->Selector = 0x58;
```

3. ***Step three****: Rocket to the HAL*

The difference between HAL and Ntoskrnl is that HAL uses KPCR structure to get the IDT entry base address.

For reminding, in kernel-land FS points to the Processor Control Region (KPCR) structure. This structure can be found at the hard address 0xFFDFFF00. Anyways, kernel programmers would rather use FS:0x1C which points to the SelfPCR structure’s member.

The problem is that HAL exists in six different versions. This involves that the methods to access to the IDT are different from each others.

- "Standard PC", Non-ACPI PIC HAL (Hal.dll)
- "MPS Uniprocessor PC", Non-ACPI APIC UP HAL (Halapic.dll)
- "MPS Multiprocessor PC", Non-ACPI APIC MP HAL (Halmps.dll)
- "Advanced Configuration and Power Interface (ACPI) PC", ACPI PIC HAL (Halacpi.dll)
- "ACPI Uniprocessor PC", ACPI APIC UP HAL (Halaacpi.dll)
- "ACPI Multiprocessor PC", ACPI APIC MP HAL (Halmacpi.dll)

There are three different ways to access to the IDT Entry without the SIDT opcode.

The first one consists to accessing the SelfPCR structure’s member.

```
         mov r00, dword ptr fs:0x1C
         mov r01, [r00+0x38]
```

The second one directly reads the IDT Entry member.

```
         mov r00, dword ptr fs:0x38
```

And the last one uses an untypical scheme to access the KPCR by using its hard address instead of FS:0x1C.
Therefore, we have a part of code like this.

```
mov r00, dword ptr ds:0xFFDFF000
         mov r01, [r00+0x38]
```

Or more directly like these following:

```
mov r00, dword ptr ds:0xFFDFF038h
```

For instance, with the HalpMcaCurrentProcessorSetTSS() (halmacpi.dll) function which modify the 18th interrupt.

```
mov     ecx, large fs:38h
lea     eax, [ecx+144]
mov     byte ptr [eax+5], 85h
mov     word ptr [eax+2], 0A0h
```

Although the most common way to initialize interrupts in the Vista’s HAL is the following:

```
mov     edx, large fs:1Ch
mov     edx, [edx+38h]
movzx   eax, al
shl     eax, 3
mov     ecx, offset HalpHpetRolloverInterrupt
mov     edi, ecx
shr     ecx, 10h
mov     [edx+eax+6], cx
mov     [eax+ecx], di
```

II. Syscall

Once upon a time near the edge of the lost kernel there was a funny function called "KiLoadFastSyscallMachineSpecificRegisters".

To call a native function Windows uses ntdll.dll to switch from CPL3 to CPL0.
This switch is done by the *SYSENTER* opcode. Metasploit published a full listing for system call table index, available here [9].

After referring into the Intel instructions handbook [10], we note these following notes:

```
SYSENTER—Fast System Call
      Executes a fast call to a level 0 system procedure or routine.
      SYSENTER is a companion instruction to SYSEXIT. The instruction is
      optimized to provide the maximum performance for system calls from
      user code running at privilege level 3 to operating system or
      executive procedures running at privilege level 0.
      Prior to executing the SYSENTER instruction, software must specify
      the privilege level 0 code segment and code entry point, and the
      privilege level 0 stack segment and stack pointer by writing values
      to the following MSRs:
            • IA32_SYSENTER_CS — Contains a 32-bit value, of which the
            lower 16 bits are the segment selector for the privilege level
            0 code segment. This value is also used to compute the segment
            selector of the privilege level 0 stack segment.
            • IA32_SYSENTER_EIP — Contains the 32-bit offset into the
            privilege level 0 code segment to the first instruction of the
            selected operating procedure or routine.
            • IA32_SYSENTER_ESP — Contains the 32-bit stack pointer for the
            privilege level 0 stack.
      These MSRs can be read from and written to using RDMSR/WRMSR.
      Register addresses are listed in Table 4-6. The addresses are defined
      to remain fixed for future Intel 64 and IA-32 processors.
```

For remaining a 32bits switch looks like:

```
Zwxxxxxxxxxxxxx proc near
                                   mov eax, FunctionIndex
                                   mov edx, ppKiFastSystemCall
                                   call dword ptr ds:[edx]
                                   retn 0xXX
Zwxxxxxxxxxxxxx endp

KiFastSystemCall proc near
                                   mov     edx, esp
                                   sysenter
KiFastSystemCall endp
```

We first notice that Zw* functions move function's index into eax and then jump near KiFastSystemCall.

Furthermore, this is the KiFastSystemCall function which executes the SYSENTER opcode to enter in CPL 0 from CPL 3.

That’s another way used by Rootkits’ authors who don’t want to patch the SSDT because of the easiness of being detected and restored with tools like SDTRestore.

As we can see above in the Intel documentation, there are three important MSR registers to initialize (IA32_SYSENTER_CS, IA32_SYSENTER_EIP, and IA32_SYSENTER_ESP).

Let’s take a look at theses structures and constants.

```
#define IA32_SYSENTER_CS 0x00000174
#define IA32_SYSENTER_EIP 0x00000175
#define IA32_SYSENTER_ESP 0x00000176

//
// IA32_SYSENTER_CS
//
typedef struct _SEP_SEL {
    USHORT Segment;
    USHORT scratch;
    ULONG ignored;
} SEP_SEL, *PSEP_SEL;

//
// IA32_SYSENTER_EIP
//
typedef struct _SEP_EIP {
    ULONG TargetEIP;
    ULONG ignored;
} SEP_EIP, *PSEP_EIP;

//
// IA32_SYSENTER_ESP
//
typedef struct _SEP_ESP {
    ULONG TargetESP;
    ULONG ignored;
} SEP_ESP, *PSEP_ESP;
```

And here the decompiling of the broadcasted function pushed to KeIpiGenericCall inside the KiRestoreFastSyscallReturnState function.

```
ULONG_PTR
KiLoadFastSyscallMachineSpecificRegisters(
                                          IN ULONG_PTR  Argument)
{
    ULONG Unknow;

    if (KiFastSystemCallIsIA32)
    {
        _asm mov Unknow, dword ptr fs:[0x20]
        WRMSR(IA32_SYSENTER_CS, 0x08, 0x00);
        WRMSR(IA32_SYSENTER_EIP, KiFastCallEntry, 0x00);
        WRMSR(IA32_SYSENTER_ESP, Unknow.u1988, 0x00);
    }
}
```

As you can see, there is a series of three calls towards WRMSR function. Therefore, it becomes very easy to build a signature to localize this part of code.

```
6A 00                  push    0
6A 08                  push    8
68 74 01 00 00         push    174h            ; IA32_SYSENTER_CS
E8 XX XX XX XX         call    _WRMSR@12       ; WRMSR(x,x,x)
6A 00                  push    0
68 YY YY YY YY         push    offset _KiFastCallEntry
68 76 01 00 00         push    176h            ; IA32_SYSENTER_EIP
E8 XX XX XX XX         call    _WRMSR@12       ; WRMSR(x,x,x)
6A 00                  push    0
FF B6 88 19 00 00      push    dword ptr [esi+1988h]
68 75 01 00 00         push    175h            ; IA32_SYSENTER_ESP
E8 XX XX XX XX         call    _WRMSR@12       ; WRMSR(x,x,x)
```

The red bytes are constants and unchanged bytes, the green bytes are what we look for and the uncolored bytes are submitted to modifications.

There are some registers pushed and used except ESI. Thus, we are able to build a 26bytes length signature to find out the address of KiFastCallEntry.

References and further informations

[1] Bryce Cogswell and Mark Russinovich (November 2006), RootkitRevealer v1.71
http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx

[3] Microsoft (June 2006), Patching Policy for x64-Based Systems
http://www.microsoft.com/whdc/driver/kernel/64bitpatching.mspx

[4] Authentium (October 2006), Microsoft Patchguard
http://blogs.authentium.com/sharp/?p=12

[5] Uninformed (December, 2006), Subverting PatchGuard Version 2
http://www.uninformed.org/?v=6&a=1

[6] Matthieu Suiche (January 2007), Windows Vista 64-bits and unexported kernel symbols.
http://www.msuiche.net/papers/Windows_Vista_64bits_and_unexported_kernel_symbols.pdf

[7] Microsoft, Device\PhysicalMemory Object
http://technet2.microsoft.com/WindowsServer/en/library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true

[8] Microsoft Press, Windows Internals Fourth Edition
Chapter 3, Trap dispatching

[9] Metasploit, Windows System Call Table (NT/2000/XP/2003/Vista)
http://www.metasploit.com/users/opcode/syscalls.html

[10] Intel, SYENTER
IA-32 Intel Architecture Software Developer's Manual. Volume 2B