Windows Vista 64bits and unexported kernel symbols.

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

January 1, 2007

Abstract: For the first Microsoft Windows Vista Beta, several articles have been published, talking about miscellaneous subjects like IT or more specifically Operating System Security (e.g. *Matthew Conover[1]*). There are numerous conflicts between ISV and Microsoft about unexported native symbols like the *IDT*, *SDT* and some *MSR*s on x64 Windows version.

However, while the Windows Vista Beta 2 beta testing, *Joanna Rutkowska[2]* showed these initiatives will not make Microsoft Windows more secure. Further, the October 25 2006, an Anti-Virus vendor called *Authentium[3]* announced publicly that Patchguard protection has been subverted.

Actually, Microsoft Windows Vista RTM (*Release to Manufacturing)* has been released but the problem for AV vendors still exists. Even if companies have told to Microsoft that building a standalone symbols importer is an easy task. AV Companies have announced to Microsoft that the decision to make these symbols as exportable won't make it easier for Rootkit's authors to access to privileged areas.

Introduction: Windows Vista x64 uses very different internal schemes compared to the x86 version. If someone reversed the x86 kernel and wanted to reverse the x64 kernel, thinking that he will find the same data at the same address, then he is wrong. Further, there are some innovations in x64 reversing like the x64 calling convention. The reader needs to know these specificities whether he doesn't want to get stopped because of a lack of understanding with pushed arguments.

This paper is covering a quick analysis of the main parts of the Microsoft Vista kernel loader to explain how it's possible to get a hand on the main native system structures, like software interruption address, SSDT address and syscall MSRs.

Prerequisites: Deprived of access to documentation and source code, we analysed Windows Vista x64 RTM version using an external disassembler, and the latest Debugging Tools for Windows (x64) to have a CPL 0 debugger compatible with Microsoft Windows Vista x64. Some knowledge of x64 assembly is needed like news operands, registers and calling convention. Of course, a fluent assembly understanding is necessary there.

I. System Interruptions

Our story start in the *KiSystemStartup()* which the prototypes seemed to be :

```
VOID KiSystemStartup(
                     PLOADER_PARAMETER_BLOCK pKeLoaderBlock);
```

```
KiSystemStartup:
                 sub     rsp, 38h
                 mov     [rsp+38h+shadow], r15
                 mov     r15, rsp
                 mov     cs:KeLoaderBlock, rcx
                 mov     rdx, [rcx+38h]
                 lea     rax, KPCR
                 test    rdx, rdx
                 cmovz   rdx, rax
                 mov     [rcx+38h], rdx   ; PKPCR
                 sub     rdx, 180h
```

As can you see the argument *pKeLoaderBlock* is stored into the exportable variable `KeLoaderBlock` located in the `ALMOSTRO` section.
For reminding the `LOADER_PARAMETER_BLOCK` structure is:

```
typedef struct _LOADER_PARAMETER_BLOC {
LIST_ENTRY                    LoadOrderListHead;          // +0x000
LIST_ENTRY                    MemoryDescriptorListHead    // +0x010
LIST_ENTRY                    BootDriverListHead;         // +0x020
UCHAR                         KernelStack;                // +0x030
PULONG64                      Prcb;                       // +0x038
UCHAR                         Process;                    // +0x040
UCHAR                         Thread;                     // +0x048
USHORT                        RegistryLength;             // +0x050
PVOID                         RegistryBase;               // +0x052
PCONFIGURATION_COMPONENT_DATA ConfigurationRoot;          // +0x060
PUCHAR                        ArcBootDeviceName;          // +0x068
PUCHAR                        ArcHalDeviceName;           // +0x070
PUCHAR                        NtBootPathName;             // +0x078
PUCHAR                        NtHalPathName;              // +0x080
PUCHAR                        LoadOptions;                // +0x088
PNLS_DATA_BLOCK               NlsData;                    // +0x090
PARC_DISK_INFORMATION         ArcDiskInformation;         // +0x098
PVOID                         OemFontFile;                // +0x0a0
PSETUP_LOADER_BLOCK           SetupLoaderBlock;           // +0x0a8
PLOADER_PARAMETER_EXTENSION   Extension;                  // +0x0b0
} LOADER_PARAMETER_BLOC, *PLOADER_PARAMETER_BLOC;
```

The beginning of the function just fixes the `PKPCR` value to *KeLoaderBlock.Prcb*.

```
                 mov     [rdx+18h], rdx   ; PKPCR-0x180
                 mov     [rdx+20h], r10   ; PKPCR
                 mov     r8, cr0
                 mov     [rdx+1C0h], r8   ; CR0
                 mov     r8, cr2
                 mov     [rdx+1C8h], r8   ; CR2
                 mov     r8, cr3
                 mov     [rdx+1D0h], r8   ; CR3
                 mov     r8, cr4
```

```
                 mov     [rdx+1D8h], r8  ; CR4
                 sgdt    qword ptr [rdx+216h]
                 mov     r8, [rdx+218h]
                 mov     [rdx], r8
                 sidt    qword ptr [rdx+226h]
                 mov     r9, [rdx+228h]
                 mov     [rdx+38h], r9
                 str     word ptr [rdx+230h]
                 sldt    word ptr [rdx+232h]
                 mov     dword ptr [rdx+180h], 1F80h
                 ldmxcsr dword ptr [rdx+180h]
```

These following registers/tables values are stored into the structure pointer by rdx.

- CR (=Control Registers)
- TR (=Task Register)
- GDT (=Global Descriptor Table)
- IDT (=Interrupt Descriptor Table)
- LDT (=Local Descriptor Table*)*

```
                 mov     eax, edx
                 shr     rdx, 32
                 mov     ecx, 0C0000101h ; GS_BASE
                 wrmsr
                 mov     ecx, 0C0000102h ; KERNEL_GS_BASE
                 wrmsr
```

The *RDX* register is going to be stored in a MSR identified by `GS_BASE` and `KERNEL_GS_BASE` constants.

Some instructions later, the function *KiInitializeBootStructures()* is called. His prototype seemed to be like the following:

```
VOID KiInitializeBootStructures(
                      PLOADER_PARAMETER_BLOCK pKeLoaderBlock);
```

After reading the function we see that mapped *IDT* Base address is obtained in 2 lines of code:

```
                 mov     rsi, gs:18h
[...]
                 mov     pMmIdtEntry, [rsi+38h]
```

In fact, these 2 lines of code represents a 13 lines tricks of internal structure initialization:

```
                 mov     cs:KeLoaderBlock, rcx
                 mov     rdx, [rcx+38h]
                 lea     rax, KPCR
                 test    rdx, rdx
                 cmovz   rdx, rax
                 mov     [rcx+38h], rdx  ; PKPCR
                 mov     r10, rdx
                 sub     rdx, 180h
                 mov     [rdx+18h], rdx
                 mov     [rdx+20h], r10  ; PKPCR
[...]
                 sidt    qword ptr [rdx+226h]
                 mov     r9, [rdx+228h]
```

```
mov     [rdx+38h], r9
```

Where `rdx+0x18`, is a pointer to `gs:[0x18]` and `rdx+0x38` a pointer to the mapped Idt.

Note: We see that in theory `gs:[0x18]` should be equal to `GS_BASE` so `gs:[0x38]` should point to mapped *IDT*.

All of the following lines are used to copy *System Interrupt* to mapped memory. Here, the copy procedure is initialized.

```
lea     r11, (KxUnexpectedInterrupt0+1)
xor     r10d, r10d
lea     r12, (KiInterruptInitTable+8)
lea     r9, KxUnexpectedInterrupt0
lea     r8, [pMmIdtEntry+4]
sub     r11, pMmIdtEntry
```

The most interesting line here is the *R12* initialization. Whether we check this offset we will see:

```
KiInterruptInitTable dq 0
                     dq offset KiDivideErrorFault       ;  DIVIDE_ERROR
                     dq 1
                     dq offset KiDebugTrapOrFault       ;  SINGLE_STEP
                     dq 30002h
                     dq offset KiNmiInterrupt           ;  NMI_INTERRUPT
                     dq 303h
                     dq offset KiBreakpointTrap         ;  BREAKPOINT
                     dq 304h
                     dq offset KiOverflowTrap           ;  OVERFLOW
                     dq 5
                     dq offset KiBoundFault             ;  BOUND
                     dq 6
                     dq offset KiInvalidOpcodeFault     ;  INVALID_OPCODE
                     dq 7
                     dq offset KiNpxNotAvailableFault   ;  NPX_NOT_AVAILABLE
                     dq 10008h
                     dq offset KiDoubleFaultAbort       ;  DOUBLE_FAULT
                     dq 9
                     dq offset KiNpxSegmentOverrunAbort ;  NPX_SEGMENT_OVERRUN
                     dq 0Ah
                     dq offset KiInvalidTssFault        ;  INVALID_TSS
                     dq 0Bh
                     dq offset KiSegmentNotPresentFault ;  SEGMENT_NOT_PRESENT
                     dq 0Ch
                     dq offset KiStackFault             ;  STACK
                     dq 0Dh
                     dq offset KiGeneralProtectionFault ;  GENERAL_PROTECTION
                     dq 0Eh
                     dq offset KiPageFault              ;  PAGE
                     dq 10h
                     dq offset KiFloatingErrorFault     ;  FLOATING_ERROR
                     dq 11h
                     dq offset KiAlignmentFault         ;  ALIGNMENT
                     dq 20012h
                     dq offset KiMcheckAbort            ;  MACHINE_CHECK
                     dq 13h
                     dq offset KiXmmException           ;  XMM_EXCEPTION
                     dq 1Fh
```

```
dq offset KiApcInterrupt           ;  APC
dq 32Ch
dq offset KiRaiseAssertion         ;  RAISE_ASSERTION
dq 32Dh
dq offset KiDebugServiceTrap      ; DEBUG_SERVICE
dq 2Fh
dq offset KiDpcInterrupt          ; DPC
dq 0E1h
dq offset KiIpiInterrupt          ; IPI
dq 2 dup(0)
```

Doesn't it seem so interesting? After a short looking on the copy routine we can rebuild a theoretical structure for these raw interruptions entries.

```
typedef struct _KIDT_RAW_SOFTWARE_INTERRUPT_ENTRY64  {
    UCHAR       InterruptId;             // +0x00
    UCHAR       Unknow01;                // +0x01
    UCHAR       Unknow02;                // +0x02
    UCHAR       Reserved03;              // +0x03
    ULONG       Reserved04;              // +0x04
    PULONG64    InterruptionOffset;      // +0x05
} KIDT_RAW_SOFTWARE_INTERRUPT_ENTRY64, *PKIDT_RAW_SOFTWARE_INTERRUPT_ENTRY64;
```

As you see the pointer to `PKIDT_RAW_SOFTWARE_INTERRUPT_ENTRY64` allows us to get all protected-mode exceptions and interrupts detailed in the Intel Manual Volume 3[4].

For remaining the way to access to this "in-raw" structure is this one:
The way to access to the `KiServiceTable` is the following:

```
KiSystemStartup()
    => call KiInitializeBootStructures ()
        -> lea     r12, (KiInterruptInitTable+8)
```

Comparing memory interrupt address with their adjusted address is more effective than a basic checking between kernel address base and kernel base limit.
Imagine if an attacker wanted to interchange an *IDT* entry? It could affect the correct system operation.

For 32bits architecture a proof of concept is available without documentation using PhysicalMemory trick that I've written one year ago.
This tool I called "IDTGuard"[5] has been released on 10 December 2006. A paper about 32bits Windows System Protection should be published soon.

II. Syscall / Sysret

To call a native function Windows uses ntdll.dll to switch from CPL3 to CPL0. This switch is done by the *SYSCALL* opcode. Metasploit published a full listing for system call table index, available here [6].

After referring into the Intel instructions handbook [7], we note these following notes:

```
SYSCALL – Fast System Call
SYSRET – Return From Fast System Call

SYSCALL saves the RIP of the instruction following SYSCALL to RCX and
loads a new RIP from the IA32_LSTAR (64bit mode). Upon return, SYSRET
copies the value saved in RCX to the RIP.

The CS of the SYSCALL target has a privilege level of 0.
The CS of the SYSRET target has a privilege level of 3.
```

For remaining a ntdll’s function switcher looks like:

```
Ntxxxxxxxxxxxxx proc near
                                    mov     r10, rcx ; Ntxxxxxxxxxxxxx
                                    mov     eax, FunctionIndex
                                    syscall
                                    retn
Ntxxxxxxxxxxxxx endp
```

First, we notice the kernel function identifier is stored into the 32bits register: eax. Secondly, the ntdll’s function executes the *SYSCALL* opcode to switch into CPL0.

Some rootkits would rather hook the *SYSCALL* opcode than patching the *System Service Descriptor Table.*

On a 64bits system there are two important *MSR*s (=Model Specific Registers) which are initialized, `0xC0000082` and `0xC0000083`.

Let’s take a look at the structures and constants declaration.

```
#define LSTAR   0xC0000082
#define CSTAR   0xC0000083

//
// Syscall64
//
typedef struct _KLSTAR {
    ULONGLONG   TargetRIP4PM64Callers;
} KLSTAR;

//
// Syscall32
//
typedef struct _KCSTAR {
    ULONGLONG   TargetRIP4CMCallers;
} KLSTAR;
```

These two *MSR*s are configured by the *KiInitializeBootStructures()* function. If we look some lines after the *IDT* copy memory routine we can see the following part of code:

```
lea      rax, KiSystemCall32
mov      ecx, 0C0000083h
mov      rdx, rax         ; CSTAR
shr      rdx, 20h
wrmsr
```

```
lea      rax, KiSystemCall64
mov      ecx, 0C0000082h ; LSTAR
mov      rdx, rax
shr      rdx, 20h
wrmsr
```

As you can see function names are very explicit and are very easy to locate with a signature which looks like:

```
48 8D 05 XX XX XX XX  lea    rax, 0XXXXXXXXXXXXXXXXXh
B9 YY 00 00 C0        mov    ecx, 0C00000YYh
48 8B D0              mov    rdx, rax
48 C1 EA 20           shr    rdx, 20h
0F 30                 wrmsr
```

Only 5 bytes differ on 21bytes. But if we build a double signature there are 8 differing bytes on 42bytes.
Cause of `LSTAR` and `CSTAR` constant and *WRMSR* opcode, this part of code is very easy to be located.

III. System Service Descriptor Table

The `KeServiceDescriptorTable` pointer isn't exported on Windows Vista 64bits even if it's still to be on the 32bits version.

The similar points with previous version of Windows are that this pointer still being present in the `ALMOSTRO` section and `KiServiceTable` array still be in the `.text` section.

We have to look for these opcodes in the *KiInitSystem* function in the `INIT` section:

```
lea     rax, qword_1401C7120
mov     cs:qword_1401C7128, rax
mov     cs:qword_1401C7120, rax
lea     rax, KiServiceTable
mov     cs: KeServiceDescriptorTable, rax
mov     eax, dword ptr cs:KiServiceLimit
mov     cs:KiSwapEvent, 1
mov     cs:dword_1401F9990, eax
lea     rax, KiArgumentTable
lea     rax, KiServiceTable
mov     cs:KeServiceDescriptorTable, rax
```

There are several variables initialized into the `KiInitSystem` function, then find the pointer toward `KiServiceTable` could seem very delicate. Further, the `KiInitSystem` function isn't an exported function.

That's why using a 64bits *LDE* (=Length Disassembler Engine) or an open source disassembler [8] would be rather than a basic print code searching cause of these notes.
With counting instructions and opcode identification we could make a theoretical way to the *"lea rax, KiServiceTable"*.

The way to access to the `KiServiceTable` is the following:

```
KiSystemStartup()
    => call KiInitializeKernel()
        => call KiInitSystem()
            -> lea     rax, KiServiceTable
            -> mov     cs:KeServiceDescriptorTable, rax
```

Like for the IDT, get an access "in-raw" to the table is complex but not impossible. The main point of this access is the organization to use correctly a standalone disassembler to rebuild a virtual path to these variables.

For instance, you have to count the number of instructions "x" between the calling and the beginning of the function. Then, on another kernel binary file, you read "x" instructions and compare the current one with a *call*, if wrong compare the instruction at the position "x+n" and "x-n", for n a little number. Additionally, look for pushed arguments into registers and stack. Inside the function we can consider more information about instructions' scheme.

Here, we look for this instruction’s prototype “*lea reg64, [imm64]*” if we run a scan inside the function it will return numerous results. The ingenuity behind this idea is to use a basic isomorphs trick, comparing a personal signature with the compiled code.

Conclusion:

In this paper, we cover how to realize a kind of standalone “Patchguard” for 64bits architecture to check main targeted structures of rootkits.

The specificity of this paper is its 64bits oriented architecture and the improvement of authenticity trick compared to x86 existing tools like SVV (System Virginity Verifier) which are not allowed to restore interrupts or MSRs by their original values.

References

[1] Matthew Conover (2006), Windows Vista Kernel Mode Security
http://www.symantec.com/avcenter/reference/Windows_Vista_Kernel_Mode_Security.pdf

[2] Joanna Rutkowska (July/August 2006), Subverting Vista Kernel
http://invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt

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

[4] Intel, Protected-Mode Exceptions and Interrupts (5-3)
IA-32 Intel Architecture Software Developer's Manual. System Programming Guide

[5] Matthieu Suiche (December, 2006) IDTGuard v0.1 Public Build
http://www.msuiche.net/?p=9

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

[7] Intel, SYSCALL / SYSRET
IA-32 Intel Architecture Software Developer's Manual. Volume 2B

[8] Matthew Conover (2004), Open-source x64 Disassembler
http://www.cybertech.net/~sh0ksh0k/projects/x64dis/