Andrea Allievi
Independent Security Researcher

10/02/2014

AMD64 Memory Segmentation - Is the game over?

In these days that I was currently quite free, I have took the occasion to deepen a feature of all X64 systems... Indeed last month, when I was analysing a sample of Expiro File infector, I encountered an instruction like this:

```
mov     r11, gs:10h
```

Of course, according to the code context, and to my previous x86 experience, the previous opcode will move the content of current Teb (thread environment block) Stack limit field, in r11 register.
But how this is implemented in a X64 CPU?

According to Intel manuals (System Programming Guide, Chapter 3.2.4):
"In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as an additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures."

It seems that 64 bit segments and their descriptor tables are now useless. FS and GS segments are exceptions. I have dumped a GDT from a live 64 bit system, then developed a specific driver to perform some analysis. Here are the results:

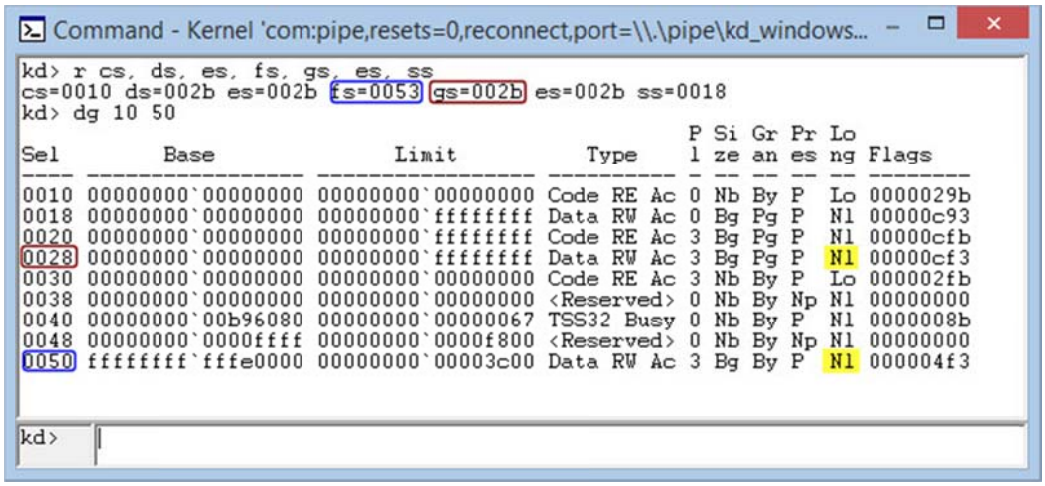

Figure 1. GDT Dump from a 64 bit Windows 7 system. FS and GS segment descriptors seems meaningless

According to the previous picture, it seems that all x64 segments are used only for memory protection (Ring 0-3 protections). But, one important question arise: how is possible that GS segment base address is 0? It is indeed very improbable that TEB could be located at address 0x00000000.
The following snapshot demonstrate that all segments are exploited in X64 architecture only for memory protection, meanwhile standard x86 segments are used for full segmentation as in the previous x86 architecture:

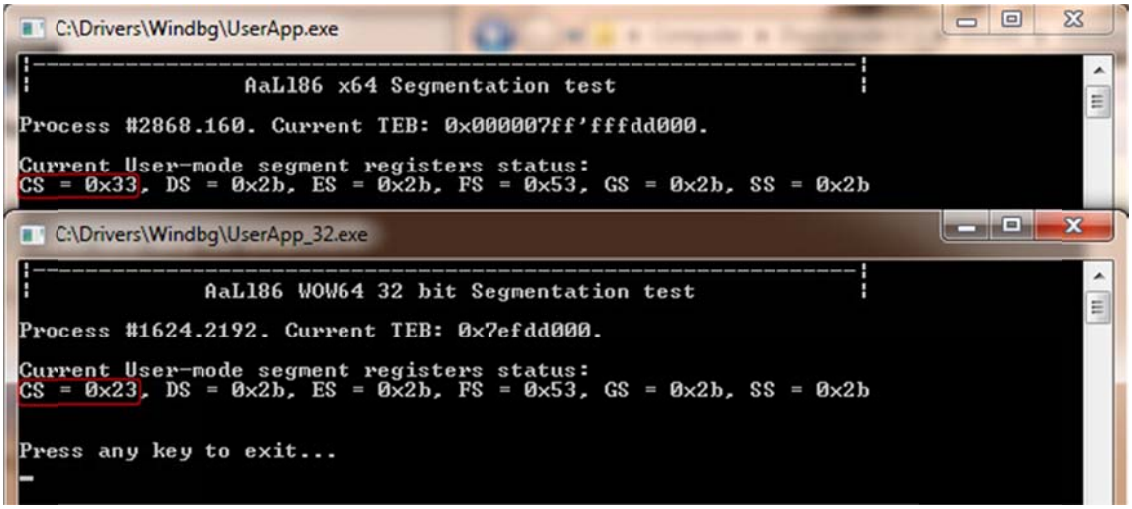

Figure 2. My GDT test application showing different segment selectors between 32 bit and 64 bit mode

So far so good... Now it's time to understand why FS and GS segments work in the way as they do for 64 bit long mode. The answer to the question resides in Windows X64 Syscall handler. Intel manual states that SYSCALL new instruction transfers execution control to the address found in IA32_LSTAR model specific register (and changes CPU current privilege level). IA32_LSTAR register points to "*KiSystemCall64*" Nt kernel routine. Whenever a native API is called from user mode, Ntdll code exploits SYSCALL instruction to perform Kernel transition. The transition is managed by "*KiSystemCall64*" procedure.

```
KiSystemCall64  proc near
        swapgs                  ; Swap IA32_GS_BASE and IA32_KERNEL_GS_BASE
        mov     gs:10h, rsp     ; curKpcr->UserRsp = RSP
        mov     rsp, gs:1A8h    ; RSP = KPCR->Prcb.RspBase
        push    2Bh
        push    qword ptr gs:10h ; PUSH UserRsp
        push    r11             ; R11 = User mode RFLAGS register
        push    33h
        push    rcx             ; RCX = Address of next user mode instruction
        mov     rcx, r10
        sub     rsp, 8
        push    rbp
        sub     rsp, 158h       ; Allocate stack space
        lea     rbp, [rsp+80h]
        mov     [rbp+0C0h], rbx
        mov     [rbp+0C8h], rdi
        mov     [rbp+0D0h], rsi
        mov     byte ptr [rbp-55h], 2
        mov     rbx, gs:188h    ; RBX = Pcr->Prcb.CurrentThread
        prefetchw byte ptr [rbx+90h]
        stmxcsr dword ptr [rbp-54h] ; Store contents of MXCSR register
                                ; The MXCSR register is a 32-bit register containing flags
                                ; for control and status information regarding SSE instructions
        ldmxcsr dword ptr gs:180h ; Loads the mxcsr register from Pcr->Prcb.MxCsr
        cmp     byte ptr [rbx+3], 0 ; if (pCurThr->Header.DebugActive)
        mov     word ptr [rbp+80h], 0
        jz      NoDebugThr
```

Figure 3. Windows 8.1 "KiSystemCall64" code snippets

This routine first invokes "swapgs" instruction. According to Intel manuals: "SWAPGS exchanges the current **GS base register value** with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software". Based on our test, the latter information is not 100% accurate. The value of GS base register actually equals to the value contained in IA32_GS_BASE model specific register. In x64 Windows systems, these MSR contains:

- **IA32_KERNEL_GS_BASE** - Pointer to current processor control region (PCR)
- **IA32_GS_BASE** - Pointer to current execution thread TEB

- **IA32_FS_BASE** - Currently unused in Windows x64. Its value equals to the base address of 32 bit FS segment descriptor (located in GDT). In 64 bit executables an instruction like "`MOV RAX, FS:[10h]`" causes an access violation

The test driver I have developed confirms all the previous conclusions. As a matter of fact, Windows operating system, when working in 64 long mode, has GS segment that always **points to current thread TEB** (in user mode), whereas in kernel mode points to **current processor PCR**.

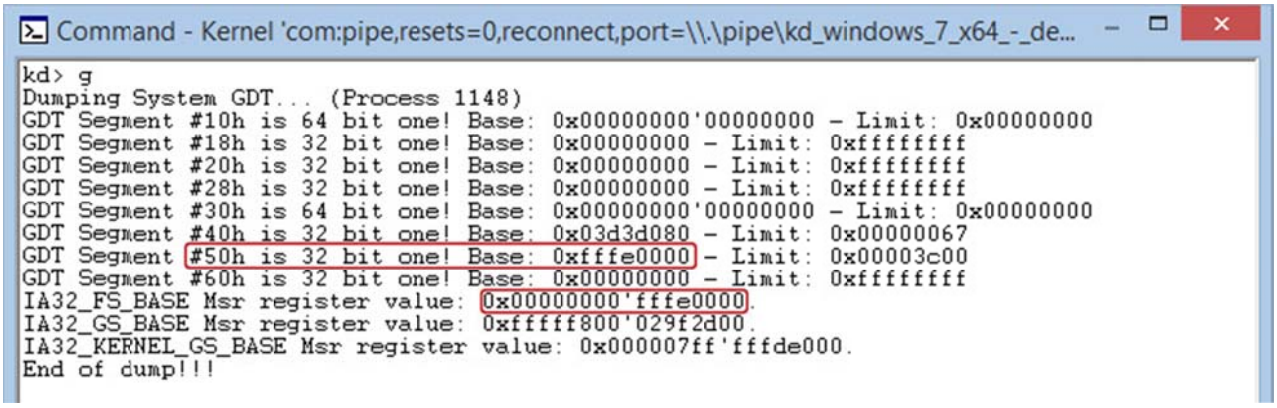

Figure 4. My test driver showing the 3 Model specific registers that deal with segmentation

Adding a segment descriptor, and doing some other tests, confirms again that X64 GDT is used in long mode **only** for memory protections. In 32 bit compatibility mode, all segments are used as normal for memory segmentation.

Returning to Windows System call handler, its job is quite easy: as the reader can see, the user-mode stack pointer is saved in PCR data structure, Kernel stack pointer is then retrieved, all GP registers (and MMX flag register) are pushed on the stack and user-mode debug environment is saved (if needed). Execution control is transferred to "*KiSystemServiceStart*". The latter routine is the key of System service dispatch feature: first, it calculates the right system table pointer **(**if the target native API number is above 0x1000, then the required function is a Win32 Gdi graphics one, otherwise a standard Nt kernel API). It retrieves the right native API pointer from table, copies all remaining stack parameters (*KiSystemServiceCopyStart*) and finally calls kernel API. Noteworthy is that Windows 8 & 8.1 Syscall dispatch is totally changed from older Microsoft operating systems. Deep describe these new features is behind the scope of this brief paper…

Last revision: 12/02/2014