POP SS/MOV SS Vulnerability

1 Summary

When the instruction, POP SS or MOV SS, is executed with debug registers set for break on access to a relevant memory location and the following instruction is an INT N or SYSCALL, a pending #DB will be fired after entering the interrupt gate or system call transition, as it would on most successful branch instructions. Other than a non-maskable interrupt or perhaps a machine check exception, operating system developers are assuming an uninterruptible state granted from interrupt gate semantics. This can cause OS supervisor software built with these implications in mind to erroneously use state information chosen by unprivileged software.

Since the SYSCALL control flow transfer is affected by this behavior, as well as INT N, the operating system’s INT 01 handler can be entered on a user stack pointer.

1.1 Impact

This is a serious security vulnerability and oversight made by operating system vendors due to unclear and perhaps even incomplete documentation on the caveats of the POP SS and MOV SS instructions and their interactions with interrupt gate semantics. The following depends on OSV implementation, but most, if not all, implement SWAPGS the same way.

For operating systems running on Intel and AMD hardware, an attacker is able to run the INT 01 handler on a user stack pointer. Furthermore, the attacker could also run the INT 01 handler with state information chosen by a lesser privileged execution mode. For instance, a user crafted GSBASE can be supplied since most operating systems determine the need to SWAPGS based off of the previous mode of execution.

POP SS and MOV SS are exploitable on any operating system where the INT 01 handler is not guarded with an IST stack (or a TSS based task switch in legacy mode), and where the handler makes assumptions about the possible previous system state such as if the handler was written without NMI semantics.

1.2 Background

The POP SS and MOV SS instructions, much like their relatives (POP/MOV sreg), are used to load a segment selector into SS, and fill the SS attributes accordingly from the corresponding GDT or LDT entry. In real-mode, this behavior is similar except that the SS value corresponds to the segment base\(^1\) and the remaining attributes are either the CPU reset values, or they were set before a transition back to real mode.

Somewhere around the release of the 8086, Intel decided to add a special caveat to instructions loading the SS register: MOV SS and POP SS. Even though system software developers could add interrupt guards\(^2\) to code loading SS, Intel added functionality where loading SS with either of the two previously mentioned instructions would force the processor to disable external interrupts, NMIs, and pending debug

\(^1\) SS <= 4
\(^2\) CLI, clear interrupts, and STI, re-enable interrupts
exceptions until the boundary of the instruction following the SS load was reached. For example:

```
xor eax, eax ; Recognize pending interrupts
inc rdi ; Recognize pending interrupts
mov bx, 50h ; Recognize pending interrupts
mov ss, bx ; INTR/NMI and certain #DB held
mov esp, eax ; Recognize pending interrupts in architectural order after instruction executes
```

The entire purpose of this functionality was to prevent an interrupt from being recognized and taken immediately after loading SS, but before loading a stack pointer. Hence, based on the design of an OS at the time, if an interrupt occurred, the interrupt would probably be taken on a bad stack linear address.

This functionality still remains in today’s architecture, even though segmentation is in little use today.

Coinciding with the Intel documentation, it should be noted that certain Intel CPUs we have tested will take an execution-only #DB exception on an instruction immediately following an SS load. However, other #DB exceptions, such as single steps and hardware breakpoints on read or write matches, will be held. While some CPUs exhibit this behavior, others do not, and this seems to be inline with the SDM documentation on this behavior.

Single steps and hardware breakpoints on read or write matches were most likely left pending because in a typical debugging scenario, they are less predictable.

## 2 The vulnerability

IDT gate descriptors come in 3 main flavors - trap gates, interrupt gates, and task gates. Task gates are out of scope of this document. The other 2 only have one difference between them.

Interrupt gates mask interrupts after the branch occurs. This means that the IF bit in EFLAGS is set to 0.

Trap gates leave EFLAGS.IF as it was on entry.

Interrupt gates are useful in this way, as it allows system software designers to bring the CPU into a serialized state before handing control off to the relevant interrupt handlers. For example, the system designer may want to clean the debug registers or switch out some selector values. This is important, because in the case of an inter-privilege interrupt, only SS and CS will be changed.

Furthermore, the designer will likely want some quick access to global structures, and will base the need to switch off of the previously executing CPL. In a modern OS, this would probably be checked by the RPL on the stack. Ideally this would be performed in a non-interruptible scenario, so that software running a nested interrupt would not become confused and think that since the prior CPL was 0, that it can just blindly use certain attributes of the previous state. For instance, an example of a standard interrupt gate prologue in Windows³ can be seen below:

```
KiBreakpointTrap proc
sub rsp, 8
push rbp
sub rsp, 158h
lea rbp, [rsp+80h]
mov [rbp+TrapInfo.ExceptionActive], 1
mov [rbp+TrapInfo._Rax], rax
mov [rbp+TrapInfo._Rcx], rcx
mov [rbp+TrapInfo._Rdx], rdx
mov [rbp+TrapInfo._R8], r8
```

³ Disassembly taken from ntoskrnl.exe, Windows 10.0.15063.608
mov [rbp+TrapInfo._R9], r9
mov [rbp+TrapInfo._R10], r10
mov [rbp+TrapInfo._R11], r11
test byte ptr [rbp+TrapInfo.SegCs], 1
jz short ExecutingInKernelModeContext
swapgs
mov r10, gs:_KPCR.Prcb.CurrentThread
test [r10+_KTHREAD.Header.DebugActive], 80h
jz short DebugIsActive
mov ecx, 0C0000102h
rdmsr
...

If an interrupt could occur before the handler was able to set up a good state, this would spell disaster for the assumed state of GSBASE. For instance, if we could trigger an interrupt immediately after one transitions from CPL 3, but before the Swapgs instruction, we could trick system software into using a user GSBASE.

Much like the SYSRET vulnerability⁴, we would need an unexpected interrupt to occur during a software serialization point.

A common bad assumption is that when interrupts are disabled via EFLAGS.IF that somehow #DBs fall under this category. Hence when IF is 0, either by a CLI, or an interrupt gate: a pending #DB, NMI or machine check can still occur, and this is the main focus of our vulnerability.

Imagine the following instructions, where DR7 and DR8 are also set for access on the stack pointer at the exact linear address where the MOV SS will read it from the stack:

; DRx primed with RAX’s linear address
mov ss, [rax]
int 3

The #DB will not be immediately recognized after the MOV SS retires because of the functionality discussed earlier. It will be suppressed until after the INT 03 retires.

The oversight here is that provided INT 03’s DPL is accessible from the assumed CPL (here being 3) that INT 03 is a simple branch. In Windows, after the transition to the INT 03 handler in the kernel, EFLAGS.IF will be implicitly cleared since the handler was set up by the OS as an interrupt gate. A #DB will be recognized after the boundary into the INT 03 handler. This is similar to as if we had placed a simple JMP after the MOV SS above; the #DB would be dispatched after the branch retires.

This results in the first instruction of the INT 03 handler, a CPL 0 CS etc, being pushed onto the stack of the INT 01 handler. Since the INT 01 handler now thinks that our previous mode was CPL 0, we can now run the handler with a GSBASE of whatever we set from usermode, but with supervisor level access.

Aside from SYSCALL, this behavior seems only to occur if the DPL check passes with the interrupt instruction. So, for usermode code executing under Windows, this leaves us with INT 03, INT 04, and SYSCALL to activate the POP SS and MOV SS vulnerability. ICEBP also seems a likely candidate here but on all hardware tested, it just multiplexes the pending debug exception into DR6 and fires a single INT 01. Furthermore, this behavior also does not seem to occur with the INTO instruction. This leads us to believe that if there is any additional processing, the pending debug exception is simply discarded. Thus, we have concluded that any INT N instruction is vulnerable, provided the DPL check succeeds and doesn’t dispatch a #GP instead.

Since SYSCALL is also vulnerable, not only can we run the INT 01 handler with a GSBASE of choice, but we can run it with our desired stack of choice too. This GREATLY increases the threat landscape of this vulnerability. Further, if the operating system does not enable SMEP, execution of user code could easily be achieved. This is because SYSCALL does not switch stacks. It is left to the system software to perform the serializing state. This means that the INT 01 handler will be dispatched using a user stack pointer. In this case, the attack would be carried out as follows:

---

⁴ [https://nvd.nist.gov/vuln/detail/CVE-2012-0217](https://nvd.nist.gov/vuln/detail/CVE-2012-0217)
NOTE: If the instruction following a `ss` load has an execute `DRx` match armed in `DR7`, it can also trigger this behavior provided the read/write match is also armed for the stack access. For instance, the `INT 01` will be taken for the execute match, and another immediately after since the blocking behavior is released. Interestingly enough for this case, this only happens when breaking on certain instructions, even though these instructions are never executed.

4 Weaponizing

By discovering and leveraging additional pointer leaks within the Windows kernel, we were able to load and execute unsigned kernel code. It is important to note that the Meltdown vulnerability mitigation, known as kernel page table isolation, changes how the exploit must be carried out in order to achieve success. For instance, with KPTI on, we were able to load a user crafted CR3 value. To follow the steps we took to build these attacks, please check the slides from our upcoming BlackHat 2018 presentation.

5 Special thanks

Andy Lutomirski - Linux kernel
Andrew Cooper - Xen