











POP SS Vulnerability

 Nick Peterson	 Nemanja Mulasmajic
 Everdox Tech LLC	 triplefault.io
 everdox@gmail.com	 nm@triplefault.io
 nickeverdox	 0xNemi
 everdox	 0xNemi

1 Summary

When the instruction, `POP SS`, is executed with debug registers set for break on access to that stack location and the following instruction is an `INT N`, a pending `#DB` will be fired *after* entering the interrupt gate, 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.

On AMD hardware, not only is `INT N` affected, but so is `SYSCALL`. This means the `INT 01` handler can be entered on a user stack pointer, since OS software has little to no reason to setup an IST or task gate for the `INT 01` handler.

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` instruction and its interaction 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 hardware, an attacker is able to execute the `INT 01` handler with a user `GSBASE` pointer.

The implications are worse for AMD hardware. An attacker is able to run the `INT 01` handler with a user `GSBASE` pointer and a **user stack pointer**.

`POP SS` is 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` instruction, much like its relatives (`POP sreg`), is 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 pretty similar, except that the `SS` value corresponds to the segment base¹ 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 guards² 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 exceptions until the boundary of the instruction following the `SS` load was reached. For example:

¹ `SS <= 4`

² `CLI`, clear interrupts, and `STI`, re-enable interrupts

```

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 done 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
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
swaps

```

³ Disassembly taken from `ntoskrnl.exe`, Windows 10.0.15063.608

```

mov r10, gs:_KPCR.Prpcb.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 transition from CPL 3, but before the `SWAPGS` instruction, we could trick system software into using a user `GSBASE`.

Much like the `SYCRET` 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 `DR0` are also set for access on the stack pointer at the exact linear address where the `POP SS` will read it from the stack:

```

; GSBASE would ideally first be primed with
; WRGSBASE in a 64 bit code segment
; Hardware breakpoint (DR0) set to memory
; address where stack is, e.g. 0x401000.
call SetThreadContext

; Lets imagine that 0x401000 contains a valid SS
; selector.
mov esp, 401000h
pop ss
int 3

```

The `#DB` will not be immediately recognized after the `POP 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 `POP 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.

We have tested this behavior on Intel hardware with the `SYSCALL` instruction, and instructions that cause a fault. In these cases, the `#DB` seems to be discarded entirely. This behavior only occurs if the DPL check passes with the interrupt instruction. So, for usermode code executing under Windows, this leaves us with `INT 03` and `INT 04` to activate the `POP 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 `INT0` 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.

On AMD hardware, the `SYSCALL` instruction is also vulnerable. This means, 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 on AMD systems to include arbitrary

⁴ <https://nvd.nist.gov/vuln/detail/CVE-2012-0217>

code execution. 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:

```
call SetThreadContext
mov esp, 401000h
pop ss
syscall
```

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.

3 Mitigations

It seems, in a way, that this is just a giant oversight. `INT N` as an instruction boils down to a branch. System software developers are assuming exclusive execution after an interrupt gate, but are still vulnerable to the occurrence of a `#DB` in this rare circumstance. In all likelihood, we expect Intel and AMD to update their instruction specifications to make clear note of this edge case.

In the meantime, `#DB` handlers should be written much like a NMI handler, and should `SWAPGS` unconditionally to a known good state regardless of the previous mode. Next, the handler needs to know if the `INT 01` should be treated as a user or kernel mode exception. There are a number of ways this could be achieved. One idea, would be to set a “last known CPL” bit in a privileged space and then testing or toggling that bit outside of the boundary where a spurious `#DB` is possible, but before normal interrupts are enabled.

This is much like the *paranoid* entry type in Linux, and by default gives Linux kernels an edge to not falling prey to this vulnerability, provided that the `INT 01` handler was built as such. However, whether or not the exception should be treated as user or kernel generated would still need to be addressed.

Additionally, for operating systems compatible with AMD hardware, the `INT 01` handler needs not only to follow the previously prescribed mitigation, but also must use an IST entry for the `INT 01` handler on x86-64. On x86 legacy, a task gate should be used for a known good stack.

Similar to `SYSENTER` not clearing `EFLAGS.TF`, the `INT 01` handler for the AMD mitigation will also have to check the instruction pointer pushed onto the stack to determine if the exception should be handled as user or kernel generated.

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, hence 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 on Windows running on AMD hardware. However on Intel, since `SYSCALL` isn't vulnerable to a spurious `#DB`, it is difficult to achieve anything if the attack is running in a shadowed address space. To follow the steps we took to build these attacks, please check the slides from our BlackHat 2018 presentation.