x86 Stack Switching Issues and Improvements

Andrew Cooper <andrew.cooper3@citrix.com>, Citrix Hypervisor and upstream Xen developer.

Review, feedback, corrections and suggestions courtesy of Andy Lutomirski (Linux) and others.

Revisions

- Draft 3 -
    - Minor corrections
- Draft 2 - 15th August 2020
- Draft 1 - 12th August 2020

Foreword

The author acknowledges that some of the pertinent details here predate him, and are therefore not obtained from a first-hand source. Feedback and corrections gladly welcomed.

The x86 architecture has always contained quirks and sharp corners when it comes to switching stacks and retaining state safely. Some of these have been discovered inadvertently as industry-wide privilege escalation vulnerabilities. Others are complicated and exceedingly rare circumstances, which at cloud scale is otherwise known as “practically guaranteed to happen somewhere”.

Working around these quirks and corners ranges from “complicated and error prone” to “seemingly impossible”. This has a habit of resulting in poorly tested (or non-existing) error handling for exceptional corner cases, which has a material impact on customer uptime and engineering effort required to distinguish genuinely freak crashes from other unexplained ones which may have a systematic but rare cause. Furthermore, recent additions (CET-SS, SEV-ES/SNP) and proposed additions (BLD-#DB) take an already complicated situation and make it even harder for a kernel to maintain safety and state.

This paper is intended to be a solution-neutral description of the problems from a kernel’s point of view, and a request for certain behaviour changes which would eliminate entire classes of problems. A possible set of ISA changes is presented as a concrete talking point, but is not intended to be an expectation of how the requested changes are achieved.

Architecture

32-bit

The 32-bit x86 architecture introduced with the 386 processor had fairly simple stack semantics. Each of the 4 privilege levels had separate stacks. A change in privilege level loads the appropriate SS/ESP from the TSS (e.g interrupt/exception), or restores a previously stack context which was saved on the current stack (e.g. FAR/IRET). Far transfers within the same privilege level don't switch stacks.

Occasionally it is wanted to switch stack, even on a far transfer within the same privilege level. Most obviously, the #DF handler wants to switch to a known-good stack, as the source of the double fault might be a corrupt current stack. In the 32-bit architecture, a Task Gate is the only mechanism to force a stack switch within the same privilege. Reentrancy of tasks is prevented via the Busy bit in the TSS's GDT descriptor.

NMI Handling

Non Maskable Interrupts cannot be blocked by the Interrupt Flag, but reentrancy protection is still important for software correctness. To combat the lack of masking, hardware maintains an internal NMI block/mask which behaves as an "NMI in service" bit. The NMI block is cleared on an IRET instruction, which signals the end of the NMI handler. This mechanism causes at most one NMI to be queued until the current NMI handler completes.

Fast System Calls

The overhead using Interrupt Gates for system calls lead to the introduction of the Fast System Call instructions (SYSENTER/SYSEXIT - Pentium II, SYSCALL/SYSRET - K6). The general behaviour of these instructions were based on the fact that flat segmentation (on top of paged memory management) was the predominant usage.

Both pairs of instructions suffer from semantic problems - SYSENTER totally discards the calling state: CS, SS, EIP and ESP are all lost while EFLAGS isn't suitably sanitised or preserved, while SYSCALL doesn't switch the stack at the same time as switching privilege.

MCE Handling

Like NMIs, MCEs can occur on any arbitrary instruction boundary, and cannot be blocked. An OS can opt not to receive MCEs by not enabling the mechanism in CR4, at which point any MCE is fatal to the system. For OSes which opt to handle MCEs, reentrancy is controlled by the

MSR_MCG_STATUS.MCIP, which is set when an MCE is delivered, and cleared by software in the handler. A second MCE to appear when MCIP is set results in a shutdown.

64-bit

The 64-bit architecture made large changes, some also based on flat segmentation being the overwhelmingly common model.

Relevant to this discussion is the removal of Task Switches, and introduction of Interrupt Stack Tables as a replacement for forcing a stack switch, as well the handling of GS bases.

The 7 architectural stack pointers live in the Task State Segment. Any IDT vector with a nonzero IST field loads the specified stack pointer from the TSS, which forces a stack switch even for an interrupt/exception taken in the same privilege.

Kernel thread local storage is encouraged to be in GS, given the introduction of both MSR_GS_BASE and MSR_GS_KERN, and the SWAPGS instruction which exchanges the two values, and therefore the absolute address for GS-relative memory operands.

Problematic existing behaviour

SYSCALL gap

The SYSCALL/SYSRET instructions do not switch the stack, and leave this as an exercise to software. On the SYSCALL side, it typically requires 3 instructions to establish a kernel stack (SWAPGS; MOV %rsp, %gs:user_rsp; MOV %gs:kern_rsp, %rsp) without losing user state.

An absolute minimum of one instruction (XCHG %rsp, ptr) can be achieved with some virtual address space tricks, at the cost of the fully locked memory access, but this only reduces the vulnerable window - it doesn't remove it.

During this period of time an NMI, #MC or #DB can occur. The former two because they are truly asynchronous events which can't be masked, and the latter because of a MovSS-delayed breakpoint across the SYSCALL instruction itself.

Taking any of these exceptions before establishing a kernel stack is a privilege escalation vulnerability, as the processor takes an exception on a stack which userspace can write to.

SWAPGS

All entry and exit paths must take care to swap the user and kernel GS bases appropriately. For the simple cases, SWAPGS needs executing unconditionally in the fast system call paths (as these can only switch between user and kernel), and needs executing conditionally in interrupt/exception handlers, based on the interrupted context.

The current user/kernel GS state, for most intents and purposes, is a hidden binary state; one too many, or one too few SWAPGS instructions will result in kernel code executing with a user thread local storage pointer. This is a fertile source of security vulnerabilities.

Worse however are the IST vectors, which are capable of interrupting the kernel before the entry/exit paths have switched to/from the user GS base. These paths need to determine whether a SWAPGS instruction is necessary even when interrupting kernel code.

IST

In the 32-bit world, the TSS GDT Busy bit provided reentrancy protection, by escalating any forced-stack-switching reentrancy to #DF. This offers an opportunity for a proper backtrace to be constructed and saved somewhere for a developer to investigate.

In the 64-bit world, there is no reentrancy protection with IST. Any vector which uses IST and re-enters itself will clobber the outer IRET frame with an inner IRET frame pointing at somewhere on the current stack. When this occurs in practice, the result tends to be an infinite IRET-to-self loop covering the tail of the interrupt/exception handler which suffered reentrancy to begin with.

With CET-SS enabled, any reentry of this form becomes #DF, as the shadow IST stack Busy bit is already set when hardware tries to push the shadow IRET frame. While this is arguably better than losing state and entering an infinite IRET-to-self loop, it is also a corner case which can be forced in practice by userspace profiling tools (see NMI Reentrancy below), and is therefore a user-triggerable security vulnerability.

MCE Reentrancy

A first MCE sets MCIP and invokes the #MC handler. At some point during the handler, MCIP is cleared by software, to signal that the Machine Check has been handled.

However, this is not atomic with the IRET to leave the handler. Therefore, there is a window in the tail of the #MC handler where a second #MC can arrive without shutting down the logical processor and losing state.

Given system behaviour for ECC failures, it is easy to construct the above scenario using ACPI EINJ error injection, and not impossible for malicious userspace or a VM to construct it using Rowhammer.

NMI Reentrancy

NMIs have better reentrancy handling than MCE, but the hidden NMI block still makes the assumption that the next sequentially encountered IRET instruction will logically belong to the NMI handler.

In practice, the NMI handler is heavily overloaded, commonly accommodating functionality including PCI SERR/IOCK, profiling or other uses of performance counters (watchdogs, DoS mitigations, etc), and even IPIs in some corner cases (e.g. trying to interrupt CPU0 to shut down during a crash, when it is not safe to enable interrupts, and not enabling interrupts risks deadlock conditions).

Alternative sources of problems can occur from dynamic code modification (e.g. runtime livepatches) which need to transiently introduce an INT3 instruction atomically to avoid other logical processors encountering a transiently-malformed instruction as changes are being made.

Any exception which occurs during the process of handling an NMI will cause a nested handler to run, the IRET of which be sequentially ahead of the NMI handler's IRET. This causes the hardware NMI block to be dropped before the NMI handler has logically ended. Any subsequent NMI which hits this window will reenter itself.

An IRET instruction can fault for many reasons, most of which are under kernel control, but the userspace CS/SS selectors can be forced to fault by malicious userspace (e.g. with the modify_ldt system call), yielding a #GP/#SS exception. Intel parts drop the NMI block at this point, so the whole exception handler runs with NMIs unblocked. AMD parts do not drop the NMI block if IRET faults, but the IRET from the exception handler still creates a one-instruction window where a pending NMI will reenter.

IST Nesting

The NMI and MCE handlers can interrupt each other. This means that the MCE handler's IRET can drop the NMI block at an arbitrary point in the NMI handler, possibly even on the first instruction.

Both NMI and MCE can also interrupt #DB at arbitrary points, including the first instruction. Therefore, any logic required to protect against reentrancy (e.g. clearing DR7 for the duration of the handler) needs to be repeated by every other recoverable IST vector.

Detecting nesting of IST vectors is very complicated. There are no obvious mechanisms, given the ability for some vectors to nest at the head of the handler. This makes it complicated to determine when an IST vector could avoid using the IRET instruction to avoid dropping the NMI block of an outer NMI which is still in progress.

A separate consequence of arbitrary nesting is that it is that a nesting (e.g. NMI; MCE; livepatching-#DB; NMI) which loses state will end up in an infinite IRET loop between the two stacks and tails of handlers, rather than an infinite IRET-to-self loop. Spotting this pattern and even just crashing cleanly is complicated.

#VE (Intel)

The #VE vector occurs within non-root mode for conditions which would otherwise cause an EPT_VIOLATION VMExit, and it can be delivered in the SYSCALL gap. Reentrancy protection is provided by a check of an unnamed field in the Virtualisation Exception Information Area, but like MCEs and the MCIP bit, clearing this unnamed field isn't atomic with the end of the handler, resulting in a window where a nested #VE can occur and lose state.

#SX (AMD)

The #SX vector is used for INIT redirection. This is commonly associated with Dynamic Root of Trust functionality, but some hypervisors use INITs as IPIs when they are attempting to avoid virtualising the majority of interrupt handling on the system.

An INIT can be delivered in the SYSCALL gap, so must be IST. However, two INITs can be delivered in very quick succession, and if #SX is IST, there is nothing software can do to avoid losing state.

In practice, this prohibits the use of INIT redirection with root mode userspace, unless you are willing to suffer the performance overhead of using pre-Fast System Call mechanisms in preference to SYSCALL.

#VC (AMD)

The #VC vector is used by encrypted virtual machines, and is raised by hardware for any Non-Automatic Exit VMExit. Various actions (benign or otherwise) from the hypervisor or guest userspace can cause #VC to be delivered in the SYSCALL gap, and therefore #VC must be IST.

Under Secure Nested Paging, arbitrary memory references might result in #VC, meaning that malicious or accidental behaviour can cause #VC to nest, lose state, and fail to recover.

SYSENTER and CET-SS (Intel)

SYSENTER switches the regular stack, but doesn't switch the shadow stack.

#DB is required to be IST to fix the privilege escalation vulnerability of a MovSS-delayed data breakpoint across the SYSCALL instruction. On a theoretical future part which doesn't suffer from this bug (e.g. by having SYSCALL switch stack properly) the OS will want to avoid having #DB configured to be IST.

However, on such a part with CET-SS enabled, a new DoS vulnerability would be exposed, from a MovSS-delayed data breakpoint across SYSENTER. This is because SYSENTER zeroes SSP, and the delivery of #DB would try to push a shadow IRET frame to linear addresses at the very top of the upper canonical half of memory.

To gain the benefit of allowing #DB to function without using IST, such a future part would need to fix SYSENTER's behaviour under CET to also switch shadow stack.

Existing workarounds

In an effort to statistically reduce the likelihood of problems from reentrant IST vectors, various kernels have schemes where IST entries from userspace switch from the IST stack to the main stack.

This is relatively easy on entry from userspace, where it is known that the main stack is empty. The equivalent shadow stack switch is also easy with CET-SS, as MSR_PL0_SSP is already pointing appropriately.

It is very difficult to safely switch onto the main stack from an IST entry from kernel space, due to the mutual IST nesting problem. Only one IRET frame will either point to userspace (signifying that the main stack is empty), or point onto the main stack (indicating where it is safe to copy on to), and this will be the outermost nested IST vector. When IST vectors have nested, this will not be the current IRET frame.

Performing this stack switch is possible, but still has open safety questions about how to copy onto the main stack when interrupting a different IST vector which is in the middle of doing just that. Furthermore, making any of this work with CET-SS is still an open question, due to the extremely limited set of primitives for manipulating SSP directly.

Problematic future behaviour

Bus Lock Debug Exception (Intel)

Newly specified in Intel's Instruction Set Extensions guide is the Bus Lock Debug Exception. The behaviour is specified to cause a #DB trap after any instruction which results in a bus lock. #DB is an IST vector, and a bus lock can in principle happen for any instruction with a memory operand, as well as (in principle) a segment load from the GDT/LDT setting an Accessed/Busy bit when the GDT/LDT base isn't aligned on a power of 8.

Any nesting of #DB is fatal, and while it might be reasonable to extend the #DB handler to clear MSR_DEBUGCTL for the duration of the handler, the same requirement extends to all other IST vectors which may interrupt the #DB handler.

A theoretical change to this behaviour, e.g. which cleared the MSR_DEBUGCTL enable bit automatically would solve the general risk of nesting, but still leave corner cases for an NMI/MCE hitting the tail of the #DB handler, as there is no atomic way to re-enable the feature at the end of the #DB handler.

For VMMs, #DB is currently intercepted to address an DoS vulnerability caused by infinite exception delivery. This is necessary, but not desired, as it separately triggers a VMExit state management bug which loses pending breakpoint information, resulting in non-architectural behaviour from the guests point of view.

#HV (AMD)

The #HV vector is used with the Restricted Injection feature of Secure Nested Paging. It is raised by the VMM to signal that an interrupt has become pending. The VMM cannot see the state of the VM, which means that delivery of #HV may be during the SYSCALL gap, or even on top of an STI/MovSS block. There is also no queueing or hardware-managed feedback system, so the VMM can't trivially know when it is safe to deliver another #HV, to avoid losing state inside the VM.

Number of IST vectors

There are 7 IST vectors available in the architecture, due to the IST field in an IDT entry being 3 bits wide, and 0 meaning "no IST". For the readers who haven't been counting so far through this document, Intel are up to 5 IST-necessary vectors, and AMD are up to 7.

Preferred software behaviour

The author notes that he is generalising, and doesn't speak for all kernel/hypervisor developers, but anticipates that many of them will have been bitten by these corner cases, and will hopefully agree with the recommendations presented here.

Modify SYSCALL to switch stack

The IST mechanism to force a stack switch creates some very ugly corner cases, some which are impossible to recover from, or even to present a coherent crash message.

However, the real problem is that NMI, MCE and #DB (and #VE, #SX, #VC and #HV for more specialised use cases) are only IST in the first place to cover the privilege escalation hole created by SYSCALL not switching stack.

The #DF vector is the only one which realistically wants to switch stack, and only in the case of stack overflow in the first place. This tends not to be reentrant.

Fixing SYSCALL (and SYSENTER under CET) to switch stack (and Shadow Stacks) properly would remove all the "lose state due to reentrancy/weird nesting" cases, by not requiring these vectors to be IST in the first place. They would nest correctly like other interrupts/exceptions on the main stack, and not risk overwriting an outer IRET frame.

To be useful in a backwards compatible manner, such a change in behaviour shouldn't affect the userspace ABI for SYSCALL/SYSRET, although adjusting the kernel ABI is fine.

IRET without lifting the NMI Block

NMI handlers tend to be very overloaded, and are commonly hooked, meaning that near-arbitrary code runs in practice, possibly with arbitrary exception handling during that period.

Kernel code tends to have one path for returning to the kernel, and a separate path (or paths) for returning to userspace/guest. To cover arbitrary logic in the NMI handler, it would be ideal to have the return-to-kernel code path use a "never lift the NMI block" IRET-like mechanism, and for the NMI handler to use plain IRET (or alternatively change the behaviour of IRET and introduce a separate "lift the NMI block" which the NMI handler alone can use).

This avoids special casing all the current or future logic which might legitimately end up nesting within an NMI, from interfering with the NMI queueing mechanics.

Automatic SWAPGS on CPL change

There doesn't appear to have been any reason why SWAPGS couldn't have been performed automatically on CPL change, even in the original AMD64 spec. It was presumably not specified like that for simplicity reasons.

However, the fact that SWAPGS isn't automatic on CPL changes has caused a large number of security vulnerabilities, and requires some extremely complicated software logic to ensure the kernel is properly running with the correct GS base.

Furthermore, the conditional nature of the SWAPGS logic resulted in speculatively-vulnerable code patterns, requiring the insertion of one (AMD) or two (Intel) LFENCE instructions to mitigate.

Having the swap performed automatically on CPL change will be much faster than stalling speculation at the head of each interrupt/exception handler, and prevent the possibility of security issues caused by using the wrong GS base.

Atomic ends to other vectors

On a theoretical future part which did have SYSCALL stack handling fixed, and majority of the IST vectors had been switched to not being IST to resolve the state corruption corner cases, a new corner case is exposed.

#MC, #HV and #SX are all caused by asynchronous actions, and lack functionality similar to the NMI block. As a consequence, specific timing sequences of any of these interrupts/exceptions can end up nesting arbitrarily deep and overflowing the kernel stack.

When fixing the NMI block, adding blocks for the other asynchronous vectors would finally give an upper bound on the stack usage caused by arbitrary combinations of interrupts.

Sample extensions

The author is not a CPU architect, and accepts that these proposed extensions may have unforeseen problems. They are simply to start a conversation with a concrete suggestion, which to a naive opinion seems as if it would be fairly simple to implement.

- A new CPUID enumeration, and mode bit in EFER which compatible kernel software can opt into the changed ABI.
- CPL changes cause an automatic SWAPGS

- Three new per-logical-thread MSRs - MSR_SYSCALL_RSP, MSR_{SYSCALL,SYSENTER}_PL0_SSP.
    - Alternatively, MSR_SYSENTER_RSP and MSR_PL0_SSP could be reused. While the current architecture does allow for the SYSCALL/SYSENTER paths to use totally different stacks to the interrupt/exception path, this doesn't appear to be a flexibility used in practice.
- SYSCALL pushes an IRET-like frame onto the kernel stack (and shadow stack as applicable). Specifically, it pushes the values of CS, next-RIP, RFLAGS, SS, and RSP at the time of the SYSCALL instruction.
    - Pushing these values isn't technically needed for improved behaviour. It is for symmetry with the SYSRET changes which are needed, and the observation that this is how kernels arrange the stack in practice.
- SYSRET restores state from the IRET-like frame on the stack (and Shadow Stack), still permitting the "flat segmentation" optimisation which prevents it from needing to access the GDT/LDT.
    - Specifically, the CS and SS *selectors* from the IRET frame are loaded into the CS and SS registers, but neither the GDT nor the LDT are referenced. Instead, the remainder of the CS and SS registers are filled in with default values. For CS, this means that all CS fields except the selector itself are populated as they are on current CPUs. For SS, the segment should be reset to a flat segment. On AMD, the latter is an additional change in behavior -- see below.
- An extension to RFLAGS, to use bits [63:32] as a "vector in progress" bitmap (at least for NMI, #MC, #HV and #SX, but see other musings below). This accumulates as interrupts/exceptions are delivered. Nesting of these vectors will push IRET frames with multiple bits set in bitmap. A handler is expected to clear its own bit in the on-stack image, so that IRET can perform a "in_progress &= ~RFLAGS[63:32]" operation to atomically mark the vector as no longer in progress, at the same point as the handler is complete. For #MC, this should logically be the already-existing MCIP bit.

Appendix A - Minor requests

The following bugs/mis-features require working around in various expensive ways. They are all corner cases, whose behaviour could be changed in newer parts without adversely affecting existing software.

ESPFIX

When using IRET to return to a 16-bit segment, only the lower 16 bits of ESP are loaded from the IRET frame, and leak from kernel context into user context. This causes real malfunctions in legacy code, as ESP[31:16] are meaningful in some cases, particularly in mixed 16/32-bit code.

The workaround for this (called ESPFIX64 in Linux) takes a range of virtual address space per logical processor, and relies on a #DF occurring from trying to write to a read-only stack being precise. This property is not guaranteed by the architecture, and depends on CPUs continuing to behave as they currently do.

The high 32 bits of RSP are zeroed on IRET to 16/32-bitcode on Intel parts and are unchanged on IRET to 16/32-bit on AMD parts. The AMD behavior results in a difficult-to-avoid information leak.

A fix for this would be to restore all 64 bits of RSP even when returning to code using a 16/32-bit stack segment.

SYSRET (Intel)

Intel parts take #GP on a user stack when returning to a non-canonical address. Working around this privilege escalation hole involves a check (variable size when LA57 is in use) in all system call fast paths.

SYSRET (AMD)

AMD parts restore the SS selector on SYSRET, but don't reload the segment attributes as expected. This results in an unusable stack when returning to 32-bit userspace from an interrupt, using SYSRET directly.

Appendix B - Other musings

Some observations and suggestions have been given during review, which the author thinks are worthy of thought/discussion. They are not obviously an improvement as stated, but perhaps there are alternative ways to address the same underlying problems?

Reentrancy protection for regular exceptions?

The reentrancy of #VE and #VC vectors are more complicated to reason about. They are synchronous with CPU state, but not necessarily state directly under the guest kernel's control, and can potentially use an unbounded quantity of stack as a result. #VE can be handled quite simply if the guest kernel can't safely take a second #VE, by falling back to the VMExit, although #VC can't be handled in this manner.

A tangentially related problem is to do with the additional exception information for #PF and #DB being stored in CR2/DR6 respectively. Nesting of these exceptions too early in the handler can corrupt state in a manner which is not obvious to subsequent logic, and can cause some very

subtle bugs. One idea might be to extend the RFLAGS[63:32] “in progress” logic so a nested #PF or #DB escalates to #DF rather than silently corrupting state.

Taking this one step further, it can happen that #DF handlers overflow their stack and end in an infinite loop. It would be more helpful to escalate #DF with “#DF in progress” to a triple fault, because by this point, no useful progress is likely to be made by continuing execution.