Xen CET Supervisor Shadow Stacks

Release 4.15-unstable

Andrew Cooper, Citrix

Feb 11, 2021

Contents

Revision 4 - 5th June 2020

- Expand the NMI handling section.
- Expand the IST switching section.
- Minor other fixes from feedback.

Revision 3 - 14th May 2020

- Note about ISTs in SVM context.
- Rewrite INCSSP section.
- Add Livepatching and SYSENTER sections.

1 Overview

An account and observations from implementing Supervisor CET Shadow Stacks for the Xen hypervisor.

Xen (L0) was first modified to advertise the CET-SS CPUID bit to guests, the CR4 path updated to permit CR4.CET being set, and blanket guest access to the CET-SS MSRs permitted. This is sufficient to allow development of CET-SS in Xen (L1) as a virtual machine, with vCPU pinning to work around the lack of context switching for MSR_PL0_SSP.

Initial experimentation was first using XTF, a microkernel test framework and general VM playground, to gain familiarity with the instructions and their behaviour. Two critical advantages of developing in a VM first is that the edit/compile/go cycle is a matter of seconds, and it is possible to use the exception intercepts to identify the sequence of actions resulting in a triple fault.

One early accident was getting the hex opcode for SETSSBSY wrong, transposing `0x0f` for `0xf0`, turning it into `repz lock add %ebp,%eax`. This reliably triple faulting did however highlight a shortcoming in CET-SS, discussed later.

2 Relevant Xen details

Xen is a hypervisor supporting two types of virtualisation. First are known as Paravirtualised Guests (PV), which use Ring Deprivilegeing, and predate x86 hardware virtualisation support. Second is Hardware Virtual Machine (HVM) using AMD SVM or Intel VT-x extensions.

PV guest kernels execute in Ring 1 (32bit guests) or Ring 3 (64bit guests). CET-SS's asymmetric interaction with Ring 1 (vs Ring 3) makes it prohibitively difficult to use. As 32bit PV guests are only legacy workloads these days, and compatibility exists via the PV-Shim mechanism, this is unlikely to be a problem in practice.

CET support in guests is out of scope for this task, but considered nonetheless to inform the design. Support for HVM guests requires architectural additions to the existing emulation framework, but nothing more.

64bit PV guest kernels (Ring 3) have a bit of a harder time. However, there is a reasonable chance of being able to support them with a few paravirtual additions.

2.1 Stacks

For x86, Xen has per-pCPU stacks (per logical core/thread). For every CPU in the system, an 8-page (aligned) block of memory is allocated. From within this, 2 pages form the primary stack, followed by an unmapped guard page to detect stack overflows. There are also 4 IST stacks, for the #DF, NMI, #MC and #DB vectors. One page is spare.

In particular, there are not separate stacks for separate vCPUs (like there are separate stacks for each task under Linux).

TSS.RSP0 points at the primary stack, just under some metadata at the base of the stack, and TSS.IST[] point at the IST stacks. In line with other production kernels, we require IST for NMI/#MC/#DB to protect against privilege escalation vulnerabilities in the SYSCALL gap, where CPL0 code is executing with a user %rsp.

Other than SYSCALL, we would prefer to run without IST to avoid the reentrancy corner case which loses state, and ultimately renders the system broken. Like Linux in particular, when we interrupt userspace on an IST vector, we switch to the primary stack before servicing the exception. This provides a fairly good statistical defence against reentrant NMIs, which userspace can force to occur on Intel parts.

In the context of SVM vCPUs, the guests %tr is in context after VMExit, rendering IST unsafe. As a consequence, we disable all IST in SVM context (clearing the IST indices in the IDT entries), and a stack overflow causes a Triple Fault.

2.2 Context Switch

A vCPU context switch in Xen involves rewriting the base of stack metadata, and `longjmp()`-ing to assembly to enter guest context. This intentionally discards the schedulers stack frame.

3 Xen modifications

3.1 Stacks

An interesting property of the new shadow stack type in CET-SS is that shadow stacks and regular stacks act as each others guard pages as far as out-of-bound writes go. Shadow stacks also have less data on them than regular stacks, as only control flow information is present.

Therefore, the guard page for the primary stack can become the shadow stack for the primary stack, and the free page can become the shadow stacks for the IST stacks.

The final layout (from numerically highest to lowest) is 2 pages for the primary stack, 1 shadow primary stack, 4 pages for the IST stacks and 1 page comprising of 4x 1024 byte shadow IST stacks.

Even with 1k shadow IST stacks, this equates to a call tree of depth 127 (accounting for the supervisor shadow stack token), which is more than ample for expected circumstances. However, in case of problems, the stacks were ordered with #DF numerically highest, so its supervisor token is not at risk of being clobbered from a shadow stack overflow.

3.2 IST Stack Switching

When taking an IST interrupt from userspace, Xen switches onto the primary stack. Switching the shadow stack is achieved by clearing the busy IST supervisor stack token, then switching to the primary stack. This is sub optimal, and is discussed later.

3.3 Context Switch

Context switching in Xen involves a `longjmp()`-like primitive, to unwind the stack to a calculated position. The same is true for the shadow stack.

For now, this is implemented with a single INCSSP after comparing the current SSP to the base. It is not expected to need to unwind more than 255 frames from the shadow stack, so no looping logic has been implemented.

3.4 Exception Handling

The #CP exception is new in CET, and the Shadow Stack bit is new in a #PF error code. Any exception involving these is considered fatal, as it signals that something went wrong with the shadow stack handling, which are exclusive to Xen at this point.

This choice will need adjusting when CET-SS support for 64bit PV guests is added.

For instructions which may suffer hard faults (e.g. accessing user memory, probing for an MSR, etc), an exception table entry is used, to allow the exception handler to locate the appropriate recovery code. This involves rewriting %rip in the IRET frame. With shadow stacks enabled, an equivalent adjustment is required in the shadow IRET frame.

One special case for handling faults on IRET instructions works by tweaking the #GP fault to superimpose it on the guest IRET frame. This discards one IRET frame from the main stack, so an equivalent adjustment is required in the shadow stack which can be achieved with INCSSP.

3.5 NMI Control

The function `enable_nmis()` deliberately executes an IRET-to-self to drop the hardware NMI shadow. This function was extended to write itself an equivalent shadow IRET frame.

There are no plain push operations for the shadow stack. Instead, 3 call instructions with non-zero displacement are used, and then `WRSS` is used to overwrite these 3 words with a far ret shadow stack frame.

While this does function correctly, it does interfere with RAS/RSB predictions in the remainder of the call tree.

3.6 Bootstrapping

As much logic as possible is written in C rather than assembly. While there is a fair quantity of boot assembly, and we could parse the command line, it is executing at a bootloader-chosen address, operating on 2M superpage mappings and there is no dynamic memory pool.

Instead, the BSP identifies the wish for CET-SS, adjusting its own stack mappings as well as stacks allocated for the APs. For convenience, the BSP also writes out the AP's Interrupt Shadow Stack Table, so the AP boot assembly can establish CET-SS before entering C.

The BSP turns CET on for itself when switching from its boot stack to its runtime stack (actually, different linear addresses for the same physical memory), which makes use of the `longjmp()` primitive discussed earlier. This ensures there is no regular stack frame needing unwinding on a pristine and empty shadow stack.

3.7 Alternatives patching

A second reason to defer BSP CET-SS enablement until the end is because of how alternatives are handled.

During boot, Xen patches its own code based on the availability of certain hardware features. For now, this is done by clearing CR0.WP and writing directly into the read-only mappings, which has a side effect of setting the dirty bit and turning the mapping into a shadow stack mapping.

Curiously, there seems to be no interaction between the instruction fetch pagewalk and shadow stack mappings, allowing them to be executed. (This is not unexpected given the typical split between instruction and data walks, but it is amusing nonetheless that shadow stacks are executable considering how much effort has gone into making regular stacks non-executable).

However, the dirty bits are cleared after patching to prevent the mappings (which are typically 2M superpages) from being eligible to be the target of shadow stack writes.

3.8 Livepatching

The runtime livepatching mechanism (typically used for small security fixes to avoid a system reboot) uses the same underlying primitives as the Alternatives patching. For now, this is achieved by temporarily disabling CR4.CET and CR0.WP, and cleaning the dirty bits after modification.

It would be better longterm to replace the patching mechanism with one which doesn't rely on clearing CR0.WP.

3.9 Runtime and shutdown

No modifications appears necessary when making EFI Runtime System calls. This is encouraging, if somewhat unexpected. For shutdown and kexec crash, CET needs disabling to avoid the subsequent environment crashing.

4 Observations of CET-SS spec

4.1 Mode-agnostic code

Various bits of functionality need to execute safely before CET is enabled. This means they cannot have unguarded CET-SS instructions. The behaviour of RDSSP, coming from the hint nop space, actually proved to be very helpful. Consider:

```
    mov     $1, %eax
    rdsspq  %rax
    cmp     $1, %eax
    je      no_shstk
            ...
no_shsk:
```

which relies on the fact that SSP always has minimum 4 byte alignment. In the case that CET is disabled, RDSSP is a nop and leaves the 1 intact, while in the case that CET is active, the alignment of SSP guarantees that the value won't be 1.

This is far preferable to reading CR4 and MSR_S_CET in low level exception handlers.

4.2 Establishing a supervisor shadow stack

Roughly:

```
wrmsr(MSR_PL0_SSP, ...);
wrmsr(MSR_IST_SSP, ...);
wrmsr(MSR_S_CET, CET_SHSTK_EN | CET_WRSS_EN);
write_cr4(read_cr4() | X86_CR4_CET);
                // <-- Critical region
setssbsy();
```

with very little flexibility in the in order of operations. SETSSBSY requires CR4.CET and MSR_S_CET.SHSTK_EN to avoid #UD, and requires MSR_PL0_SSP to be configured first pointing at a non-busy supervisor shadow stack token.

However, between the latter of CR4 and MSR_S_CET, and the SETSSBUSY instruction, SSP has the value 0. Any interrupt/exception at this point is fatal as the CPU tries to push a shadow IRET frame at fffffffffffffffc which is unmapped.

Therefore, MSR_IST_SSP needs establishing as well beforehand, and must include stacks for all asynchronous interrupts/exceptions which might occur. NMI and #MC are the most common two, but other configurations may need to protect #VE, #HV, #VC and/or #SX.

This forces a kernel to use IST (and gain the associated complexity) for vectors it may prefer to use without IST.

4.3 IST Stack Switching

Switching from an IST stack to the primary stack occurs when interrupting userspace. The ideal sequence to use would be SETSSBSY to activate the primary shadow stack, then CLRSSBSY to deactivate the IST stack.

However, this results in SSP being reset to 0, and is therefore not a viable outcome.

Instead, CLRSSBSY is used to deactivate the current IST, then SETSSBSY to activate the primary stack. This leaves an instruction boundary where there is no shadow stack in operation, and SSP is 0, which requires full protection from IST.

It is not possible to use RSTORSSP to switch stacks, because a restore token is incompatible with a supervisor shadow stack token in Long Mode, due to the differing use for the bottom two bits.

To safely use RSTORSSP, it would be necessary to use WRSS to activate the primary shadow stack, write a restore token under the supervisor token, execute RSTORSSP to move, INCSSP to discard the restore token, and WRSS again to deactivate the IST stack. (Note: this idea was not developed, and is therefore speculation about what would be necessary.)

None of the options here are terribly good, but it would help substantially if CLRSSBSY didn't have a side effect of zeroing SSP. This would allow the use of the first option, which is the safest alternative.

There is a further complication with using SETSSBSY for shadow stack switching. In HVM context (Intel VT-x, AMD SVM), MSR_PL0_SSP is not context switched on VMEntry/Exit like the rest of the supervisor CET state, meaning that the guest kernel's choice is in context.

For now, the fact we only switch off IST stacks when the primary stack is empty means that we will only switch off IST stacks when an IST vector interrupts root mode ring3 context.

However, the risk of using a guest controlled MSR_PL0_SSP is of concern, and may cause a real problem for other ongoing efforts to try and reduce the IST loss-of-state risk with nested vectors, by switching onto an already-in-use primary stack.

4.4 CLRSSBSY

The CLRSSBSY instruction takes an m64 operand. However, it only appears sensible to use when SSP is pointing at the supervisor shadow stack token wanting deactivating.

It seemas as if it would have been more useful to be operand-less, and not alter flags. Its necessary location in the SYSRET path took quite a lot of rearranging to avoid needing to spill a guest GPR to the stack.

4.5 INCSSP

INCSSP deliberately takes a limited input, and performs shadow stack accesses at the current and resulting SSPs, to catch a stack underflow.

The limit of 8 bits entries in practice makes a stride of 1020 or 2040 bytes. It would be more convenient for software to have permitted 9 bits of entries, but I expect this decision was taken to simplify the pipeline implementation.

INCSSP also takes its stack multiplier from the operand size, not the current operating mode. This creates a seemingly-unnecessary corner case where it is trivial to misalign the 64bit SSP by using the 32bit form of INCSSP with an odd number of entries.

4.6 IRET to CPL3

This is by far and away the most problematic aspect of the CET-SS spec as far as supervisor shadow stacks go.

An IRET to CPL3 explicitly does not fault if it fails to release the supervisor shadow stack token. Instead, userspace code executes until the next interrupt/exception/syscall, at which point the kernel suffers a double fault - either SYSCALL => SETSSBSY => #CP (already busy) => #PF (push frame at fffffffffffffffc) or interrupt/exception => #GP (already busy) => #GP (already busy).

Absolutely nothing good can come of having IRET failing to release the supervisor shadow stack token, and taking #CP on the problematic IRET is more helpful for debugging the problem than taking #DF at some arbitrary point in the future.

4.7 Interaction with CPL1/2

IRET to CPL1/2 reads two extra words off the shadow stack, and is asymmetric compared to CPL3. This makes it very complicated to context switch between code in Ring1/2 and Ring3, as it requires shuffling the entire shadow stack by two words and fixing up the shadow IRET frame.

In addition, SYSCALL/SYSENTER in CPL1/2 corrupts CPL3's shadow stack pointer. This is noted along with a comment saying "software needs to preserve PL3_SSP by other means", but the two ways of doing this are RDMSR and XSAVES, both CPL0 instructions.

With these constraints, the few uses of Ring 1/2 these days (Xen being one, for 32bit PV guests) are no longer feasible (at least not in the way they currently work), and the addition of MSR_PL1/2_SSP seems like a waste of MSRs.

4.8 SYSENTER

The specified behaviour of SYSENTER zeroing SSP like SYSCALL means that full IST protection is required when running userspace, despite the fact this was previously not necessary as SYSENTER does establish a Ring0 regular stack properly.

This forces OSes to use IST for NMI/etc (and gain the associated reentrancy problems) even if they'd previously chosen to avoid SYSCALL to avoid the need for NMI to be an IST vector.

5 Closing remarks

During this exercise, one area of Xen was found which was genuinely using return oriented programming. It has been updated to use more normal call mechanics.

The use WRSS has been necessary in several positions to keep existing mechanism functioning correctly. It was hoped to leave MSR_S_CET.WRSS_EN clear unilaterally, but this is not possible. For simplicity, the current implementation has WRSS_EN set unconditionally. However, all the uses are in slowpaths, so the overhead of modifying MSR_S_CET isn't necessarily a concern - the bigger concern is over the NMI safety of the sequences. More thought is required to see if the WRSS_EN regions can be reduces to an absolute minimum.

The behaviour of binutils isn't terribly helpful. Unlike most instructions, gas doesn't accept INCSSP, RDSSP, WRSS and WRUSS without a mandatory D or Q suffix. First of all, D is not correct in AT&T syntax (L is the appropriate suffix used consistently elsewhere), and none of the instructions are ambiguous, given a register operand. The lack of suffix-less instructions makes creating 32 and 64bit-compatible code harder.

Overall, it took a couple of days to get working. This is largely down to Xen's particularly simple stack layout compared to some kernels. Work is ongoing to clean the series up for submission upstream.