SAND2019-2965C

Blackhat 2019 CFP Submission

Russell Graves, Sandia Labs

This document contains text segments that will be submitted into the Blackhat 2019 CFP form online.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Speaker Bio

Russell Graves (Sandia National Labs) has been playing in the weeds of x86 for nearly a decade. He does research and development on hypervisors, plays in system management mode, uses hardware features beyond their intended capability, and enjoys writing (if not debugging) ring 0 code. He previously worked for Google, both as a Site Reliability Engineer and as a member of the Cloud Security team, doing (among other things) large scale Rowhammer vulnerability testing.

Title

Deep OS Introspection from Hypervisors and the STM in the Post-Meltdown World

Abstract (300-word limit)

Most people know about hypervisors, and many people know about System Management Mode – but did you know about the hypervisor in System Management Mode? The SMM Transfer Monitor (STM) is just one of the interesting places we can use for introspection into running operating systems and hypervisors.

This talk discusses "deep introspection" – pulling out extensive amounts of data from an operating system or hypervisor without any cooperation from the guest operating system. Meltdown patches have made this more difficult, but there are techniques that can be used to mitigate their performance impact on introspection. I demonstrate techniques to capture detailed syscall information from a guest operating system, to capture data from IRETs (as well as SYSRETs) on the return from the kernel, and to extract detailed control flow information from an application running in the guest operating system – all without any guest OS cooperation or knowledge.

The STM (the SMM hypervisor) is the most privileged x86 operating mode on the platform – so, of course, it's another great spot for introspection code. I share results from experimenting with introspection from the STM, what it's good for, and why you might not want to use it. Plus, I share the novel technique of "STM Aided Detox" – replacing the STM-aware boot hypervisor with another hypervisor, cleanly and transparently, by bouncing through the STM when VMXON is called.

Presentation Outline

As this is a complex set of techniques to understand, I assume that the audience has a user/sysadmin level understanding of computers and may have done some coding. However, I do not assume they are familiar with the odd corners of x86, or the details of how syscalls work when talking to the kernel on a modern 64-bit operating system. The talk focuses exclusively on 64-bit x86, though much of it is applicable to 32-bit operating systems with a few tweaks (different instructions, different calling conventions).

The talk begins with an overview of the various modes of operation I'll be talking about – the kernel, applications, and a hypervisor. I explain the origins and purpose of System Management Mode, and then explain how the STM (SMM hypervisor) fits in, and how it gains execution from various events (taking VM exits from both SMM and the ring 0 "executive hypervisor").

Next, I provide a high level overview of page tables – how they're used for memory isolation, and, importantly for some later slides, how the Meltdown patches have changed things. This includes discussing the CR3 register, and how PCID (Process Context Identifiers) are used in the Meltdown patches to reduce the performance impact by avoiding TLB flushes.

The next segment starts going into detail on syscalls – and how we can extract their details from a hypervisor. I cover the syscall/sysret instructions and show how they're used to transition to the kernel with the standard calling convention, then discuss reading process memory to pull data from the pointer parameters. Capturing syscall/sysret instructions from a hypervisor is straightforward – clear EFER.SCE and lie to the guest if asked for the EFER MSR value.

Reality being yet more complex, with multiple processes, and multiple threads within a process, I talk about what this looks like, with syscalls being used for context switching by the kernel, and how (on Linux) different threads share a CR3 – but, importantly, they do not share stacks. This allows a CR3/RSP combination to uniquely identify a syscall site. This is useful for caching the syscall ID to take actions and extract data on the sysret (as the syscall ID in rax has been overwritten by the return value).

For the 10-15% of syscalls that return with an IRET for various reasons (IRET can set the entire register state, sysret cannot), these can be captured as well – set the TF (trap) bit in the FLAGS register going in, and you'll get execution on the instruction after returning to user code.

Raw syscall performance impacts are substantial (replacing a fast instruction with a pair of VM world switches isn't fast), but the actual application performance impact is a good bit less, as most applications do more than just making syscalls. We can mitigate the performance impact for "uninteresting" applications by simply not trapping syscalls for processes we don't care about. In the post-Meltdown world, this requires using CR3 Target Value support in the VMCS to avoid vmexits for every single syscall, while still allowing us to receive a vmexit on process switch (to enable or disable syscall trapping for the new process).

Another neat trick we can do from the hypervisor is using the hardware Branch Trace Store debug capability to pull control flow data out of applications in the guest – again, without the guest operating system's knowledge. The performance impact is quite substantial, but the technique works. Using Branch Trace Store requires a set of virtual addresses in the given process address space. To do this without interfering with the guest, we allocate a chunk of memory on the host, and use this memory to

store EPT tables (for mapping a region of memory into guest physical space), page tables (for mapping it into process virtual), and then put the DS Header and BTS region into that memory. By setting the various debug controls properly, we can then have the guest application's branches stream out into our freshly set up memory region, giving us detailed control flow data for the application. We've developed some techniques to make this work properly even with the split page tables post-Meltdown as well.

Finally, I talk about the STM and our techniques for introspecting on hypervisors and operating systems from the STM. This starts with a description of what the STM is and how we boot it (we're using the open source Intel STM, which comes with a small hypervisor that handles initialization). One significant advantage of introspection from the STM over the legacy SMM handlers is that the STM has access to the VMX instructions – we can use the native VMREAD, VMPTRLD, and other related instructions to read data from VMCSes, instead of having to use unsupported direct memory reads to the VMCS (which may read stale data, or may not work properly on new processors).

I discuss the control interfaces we use to work with the STM on a Minnowboard Max – we take over the pulse width modulated output and both high speed serial interfaces to talk in and out from the STM. Some GPIO pins are configured to fire off a stream of system management interrupts when linked to the PWM output, giving us regular execution. The high speed UARTs are used for command & control, as well as high speed data logging from the STM.

After explaining how the STM loads and runs, I dive into explaining a novel technique we devised: STM Aided Detox. When another hypervisor of interest runs the VMXON instruction, we trap that in the boot hypervisor. The boot hypervisor then makes a call to the STM with the entire guest state (that of the operating system that just called VMXON). The STM copies the entire state of the guest operating system over into the STM's VMCS, which previously handled the transfer to the firmware hypervisor. The STM then launches the guest with this updated state (resuming to the new hypervisor) and simulates the proper response to a VMXON instruction. The guest is then operating in VMX Root Mode, can launch guests, and is entirely unaware that an STM exists underneath.

From this point, we can use the STM to inspect the running hypervisor and guests, using native VMX instructions, and we can (if helpful) modify the hypervisor to trap into the STM at useful points. A VMCALL instruction serves as a side-effect-free instruction (sadly, 3 bytes long) to trap down to the STM at specific points of execution, in the event we want to better understand what's happening. To fire off an SMI# traditionally requires either a hardware event or making a call to ACPI IO port – which requires setting several registers. The VMCALL is far cleaner and easier to work with.

What new research, concept, technique, or approach is included in your submission that the audience has never seen before?

To the best of my knowledge, the use of the STM for introspection has not been previously talked about at a conference or in any papers. SMM introspection is an established technique but using the STM allows us to use virtualization instructions, as well as allowing VMCALL breakpoints. I don't believe the use of BTS from a hypervisor, without OS cooperation, has been discussed before. While the use of Target CR3 fields in post-Meltdown introspection is reasonably obvious, I've not seen it mentioned elsewhere. And our STM Aided Detox is, to the best of our knowledge, a novel technique that has not been used anywhere else (STM based research being rare to begin with).

Takeaways (What are three actionable takeaways audience members will gain by attending the presentation?)

- A better understanding of both what a hypervisor can do to extract information from an operating system and applications without cooperation, and how to implement these capabilities in a hypervisor.
- An understanding of what the STM is, where it fits in the x86 architecture, and the advantages and disadvantages of this execution environment for introspection or security analysis.
- If not familiar with the details already, an overview of the syscall/sysret process at the architecture level, an understanding of how the Meltdown patches impact performance, and an understanding of how various hardware features can mitigate this performance impact to some extent.

If applicable, what problem does your research solve?

This talk demonstrates multiple practical techniques to observe (or modify/filter) the behavior of operating systems and hypervisors without their knowledge. These techniques can be used to implement malware-resistant syscall filtering, to observe detailed timing behavior of interactions with a system, or for simply better understanding of what's happening in hypervisors.

We push the limit of places to introspect from with our STM work, and demonstrate that, while somewhat limited, it offers a compelling vantage point for certain types of system observation.