6 The Hypervisor Exploit I Sat on for Five Years

by DJ Capelis and Daniel Bittman

Among its many failings, peer review is especially deficient when it comes to computer security. The idea that a handful of busy researchers will properly review a security system described solely in a paper in the time they're reading through a large stack of papers is one of the extreme blind spots of our field's academic process.

It is not surprising systems with holes appear in published literature. Unfortunately, there's not even a good process to correct these situations when holes *are* found. The authors of papers are not required to provide code, so even if one suspects a hole exists, writing a proof of concept requires reconstructing the system described in the paper sufficiently well enough to have something to exploit. And then, of course, there's no point in doing any of this work, since "I found a bug in a published system" is not usually publishable, unlike *every single other* branch of science where disproving a published result is notable. In computer science, it's never notable when our papers are broken.

So neighbors, this was the situation I found myself in for the past five years or so, as I sat on a hypervisor bug in a research system no one really used. The authors, meanwhile, ignored e-mails, filed a patent on the technology described in their paper, and went on to continue a successful career in research.

Luckily, in the intervening years, a few things happened:

1. PoC||GTFO started publishing, which means anything our Pastor likes can be published here. And, especially when the Pastor has been drinking, obscurity is no bar to entry.

2. I ran into Daniel, who was building an operating system *anyway* and figured making a PoC for this bug was something he might as well do. (I was too fed-up by this point to spend the time on it.)

So without further ado, let me describe the system we pwn'd and how we pwn'd it.

The paper we're breaking in this article is *Secure In-VM Monitoring Using Hardware Virtualization*, published in 2009 at the ACM Conference on Computer and Communications Security. As these things go, in academia this is considered a "top tier" conference. Back in the dark ages, when dragons roamed the earth, and we didn't have support of Extended Page Tables (EPT) in our Intel chips, rapid page table switches were expensive. The goal of this paper was to allow quick switching between security contexts without requiring an expensive VMEXIT/VMENTER. The researchers cleverly leveraged `CR3` Target Values, which allow a limited (4, usually) set of addresses that non-root VMX code can set as the page tables base in the `CR3` register. This effectively allows an untrusted operating system to switch page tables into the code used to do introspection without causing a VMEXIT.

This neat hack caused the average overhead of their syscall introspection code to go from 46% to 4%. Which basically means that their system moved from an unreasonable performance penalty down to a level where someone could take it seriously. Which is nice, if they could keep the same security guarantees.

The security constraints were implemented in the page tables, as shown in Figure 9.

In theory, this page table setup means that the system under monitoring can never set a `CR3` value without causing a fault, except by going through the entry and exit gates. Attempts to jump directly to the introspection code fail since those pages aren't mapped into the monitored code's view of memory. Attempts to change the `CR3` value to the introspection code's page tables outside the entry gates fail because the next instruction executes in the context of the introspection code, where all those pages aren't mapped as executable. The only way to jump into the introspection code, according to the paper, is through the entry/exit gates code present in the shared gate pages and mapped as executable in both.

What we really want is a way to cause the processor to jump and move page tables at the same time. In some other architectures (SPARC, for instance) there's the concept of a delay slot, where some instructions take another instruction to fill otherwise empty pipeline bubbles. In an architecture like this, jumping out of the security boundary is trivial... but this is x86; x86 doesn't have delay slots, right?

Turns out, that is not exactly true. Quoth the Intel Architecture Manual Volume 2B on the `STI` instruction:

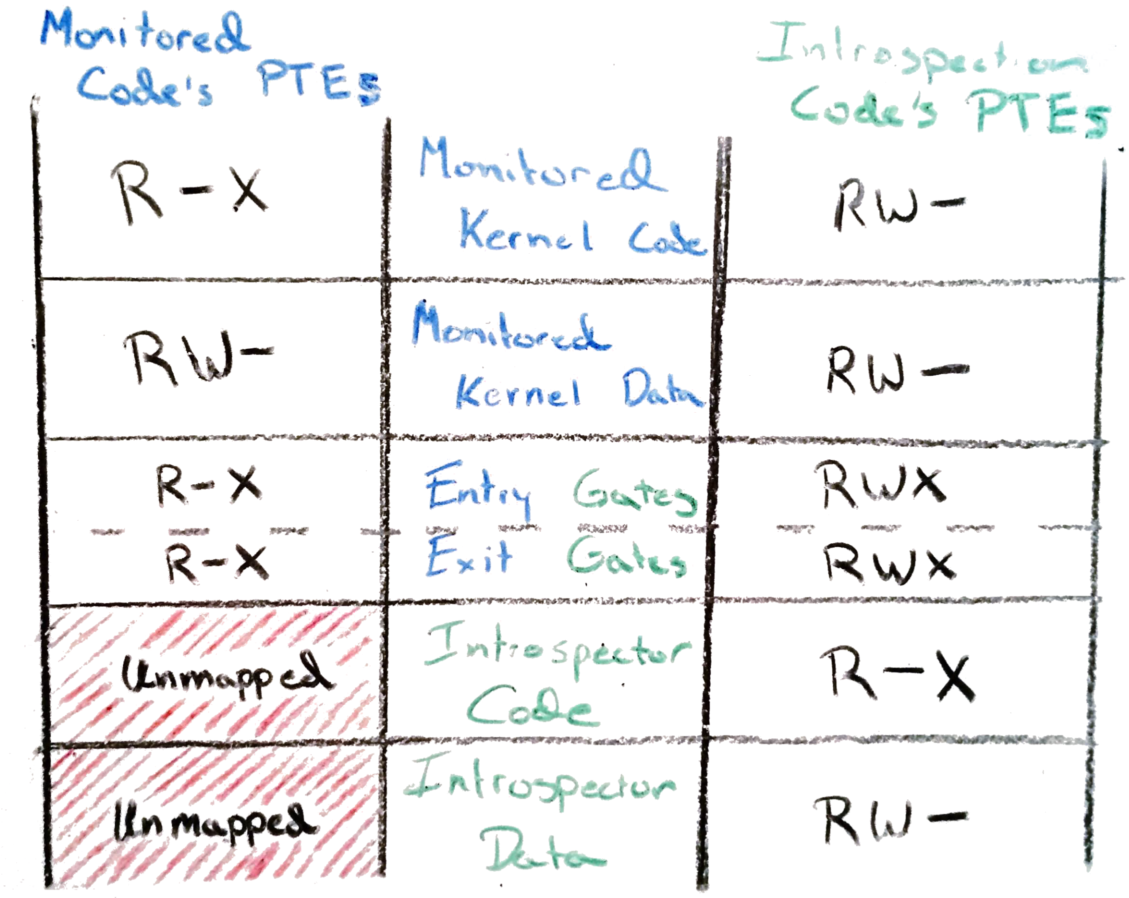
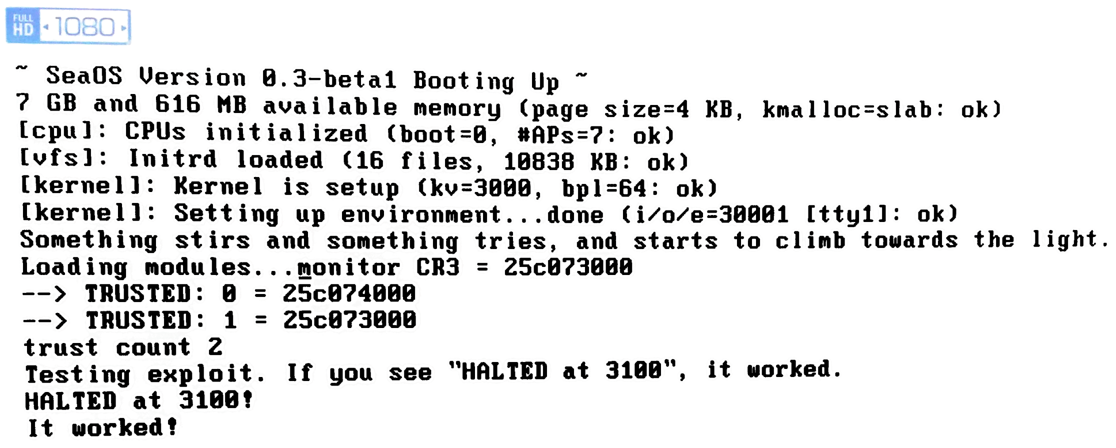


Figure 9: Page Table Security Constraints



```
~ SeaOS Version 0.3-beta1 Booting Up ~
7 GB and 616 MB available memory (page size=4 KB, kmalloc=slab: ok)
[cpu]: CPUs initialized (boot=0, #APs=7: ok)
[vfs]: Initrd loaded (16 files, 10838 KB: ok)
[kernel]: Kernel is setup (kv=3000, bpl=64: ok)
[kernel]: Setting up environment...done (i/o/e=30001 [tty1]: ok)
Something stirs and something tries, and starts to climb towards the light.
Loading modules...monitor CR3 = 25c073000
--> TRUSTED: 0 = 25c074000
--> TRUSTED: 1 = 25c073000
trust count 2
Testing exploit. If you see "HALTED at 3100", it worked.
HALTED at 3100!
It worked!
```

Figure 10: SeaOS Exploit Running on Real Hardware

After the `IF` flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an `STI` instruction is followed by a `RET` instruction, the `RET` instruction is allowed to execute *before* external interrupts are recognized.

All we need to do is turn off interrupts, queue one, route the interrupt handler into the introspection code's address space, then `MOV` the introspection code's page table base into `CR3` right after we re-enable interrupts with the `STI` instruction. Then we can just ROP our way through the monitor code and do as we please.

And that's where I stopped at three o'clock in the morning five years ago. I had the concept, but it took us another five years to getting around to proving it works on real hardware. As you can see in Figure 10, it totally does.

The final exploit turned out a little different. The most straightforward way to implement this in practice is to utilize the trap flag (`TF`). When you enable this, `POPF` has the same one-instruction delayed behavior that we see in `STI`, and so you merely just set `TF` with `POPF` and move a new value into `CR3` as the next instruction. Thus, the resulting code looks like this:

```
cli
mov rsp, 0x2500 ; we'll need a stack for the interrupt handler
mov rax, qword [0x1000] ; read the monitor's CR3 from somewhere in the trap code
lidt [idtr] ; load the interrupt table
pushfq ; get the flags
or qword [rsp], 100000000b ; set TF
popf ; set the flags
mov cr3, rax ; change address spaces
; <—— TF triggers interrupt here
loop:
jmp loop
```

6.1 Reproducibility

Everything you see here can be reproduced by running the code in the `vm-exploit` branch of the SeaOS kernel tree.[21] The code for the proof of concept itself is also in that repository.[22]

6.2 Concluding Rant

The scientific community has a *structural* problem. In computer science, we do not require researchers to build real systems that can be scrutinized. We do not have a mechanism for thorough review, so we generally do not bother publishing work that breaks another paper. Our field just doesn't consider a broken paper to be particularly notable.

Academics in computer science are too often doomed to talk nonsense unless we fix these issues. Further, researchers in our field are continuing to verge towards irrelevance if they simply follow the system of incentives that makes it a better career move to drop a paper and file a patent than do the work of building real systems and determining real truths about our machines.

To the authors of this paper in particular?
Enjoy your useless fucking patent.
Love,
~djc

[21] `https://github.com/dbittman/seakernel/`
`unzip pocorgtfo08.pdf seakernel-exploit.zip`
[22] `https://github.com/dbittman/seakernel/blob/vm-exploit/drivers/shiv/ex.s`