



# SMM

System Management Mode  
The forgotten In Circuit Emulator

[ruxmon.com/sydney](http://ruxmon.com/sydney) 2016-03-18

# ToC

- What
- Where
- How
- Oops
- Panic?

% finger benjamin.d.low@gmail.com

In real life: Ben Low

On since 2009 on pts/7 from syd.google.com

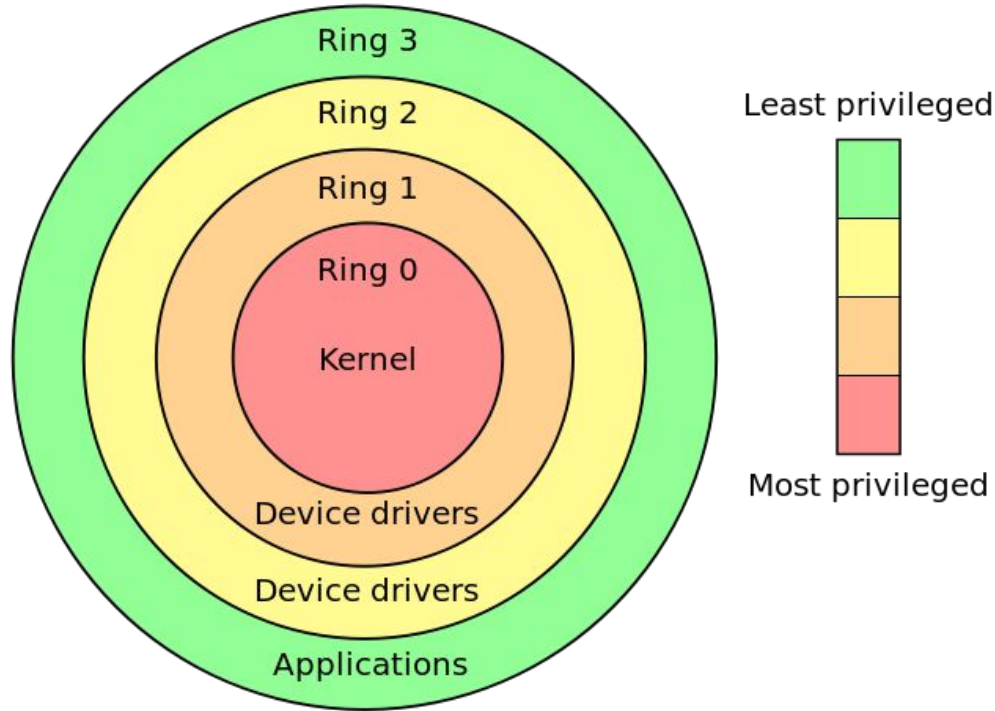
28 seconds Idle Time

Unread mail since Mon Feb 12 00:22:52 2001

Plan: improve firmware integrity

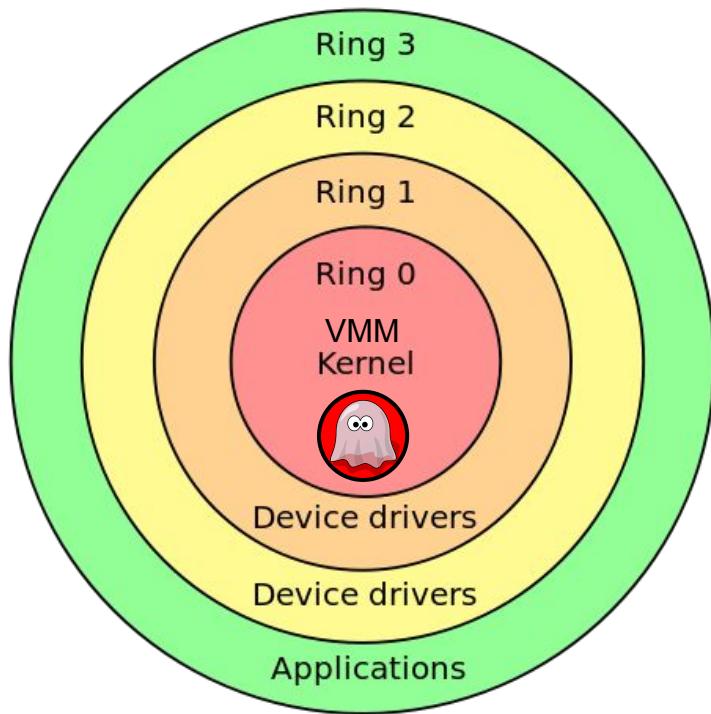


# Ring-a-ring o' roses

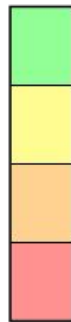


*credit: Wikipedia, natch*

# Ring-a-ring o' roses



Least privileged

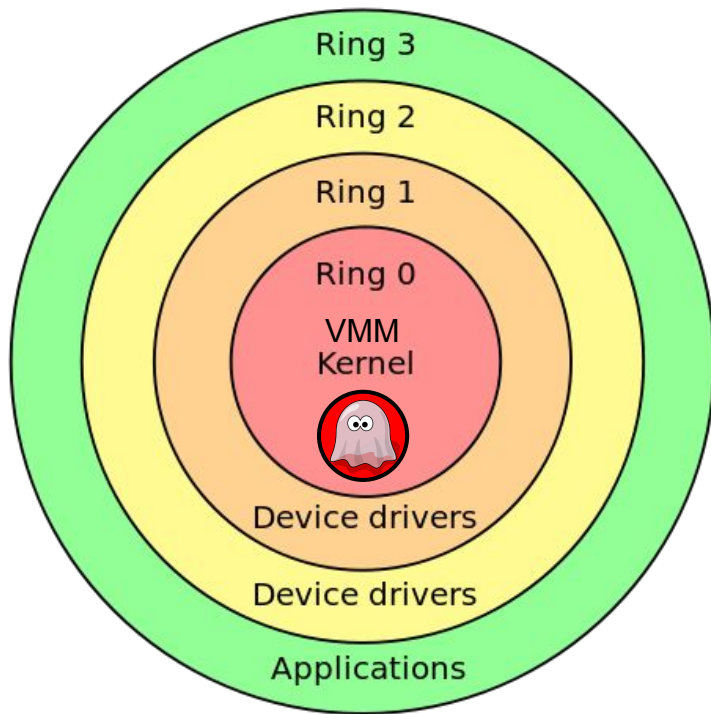


~~Most privileged~~

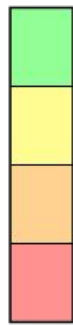
*Really most privileged...*

SMM, aka "Ring -2"

# Ring-a-ring o' roses

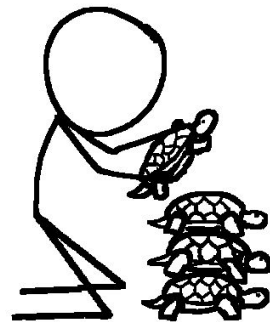


Least privileged



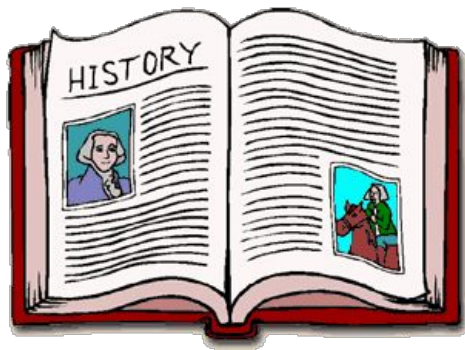
~~Most privileged~~

~~Really most privileged..~~



Unless you count the Management Engine.

# Once upon a time...



- Late last millennium (ca. 386SL), Intel introduced a feature intended to support "low level system management" operations
  - power: CPU clocks, thermal management
  - security: protect BIOS flash chip operations, ...
  - leveraged existing In Circuit Emulation features

- Totally up to the BIOS vendor to decide what goes in the SMM
  - a modern system could probably use ACPI alone



# Obligatory State Transition Diagram

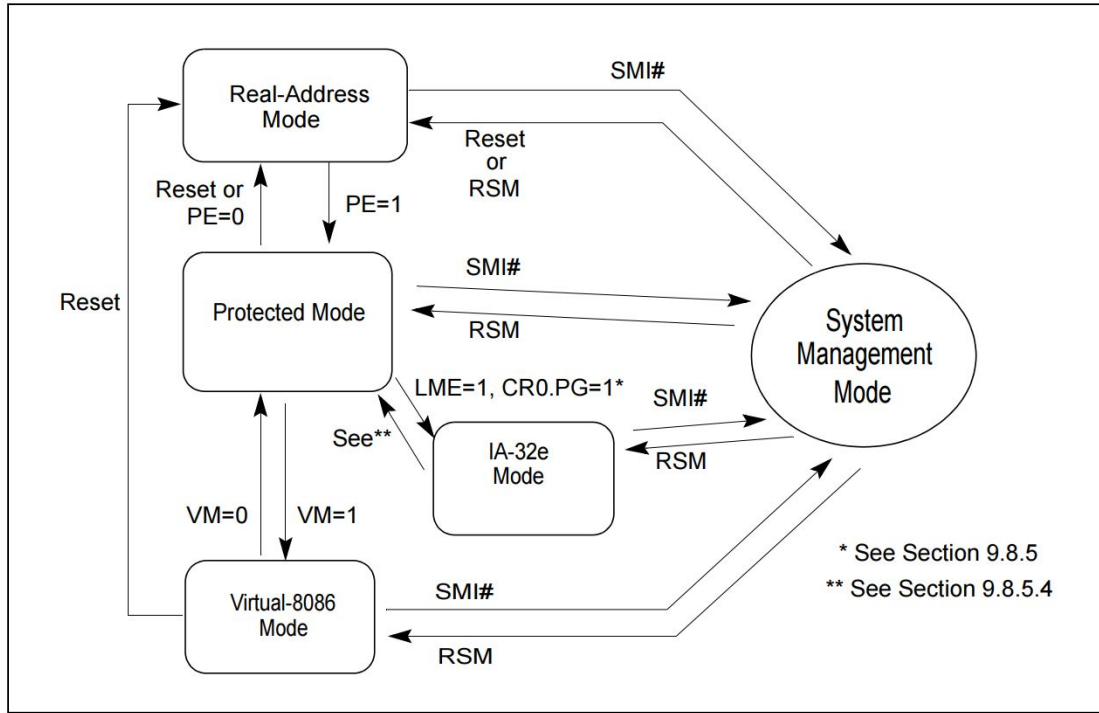


Figure 2-3. Transitions Among the Processor's Operating Modes

In SMM:

- CPU state saved & restored
- paging is disabled
- 16-bit mode of operation, though all physical memory can be addressed Real Address style (20-bit)
  - segment limits are extended to 4 Gb
  - 32-bit operand-size override prefixes may be used
- no restrictions on I/O ports or memory

# Casting Spells

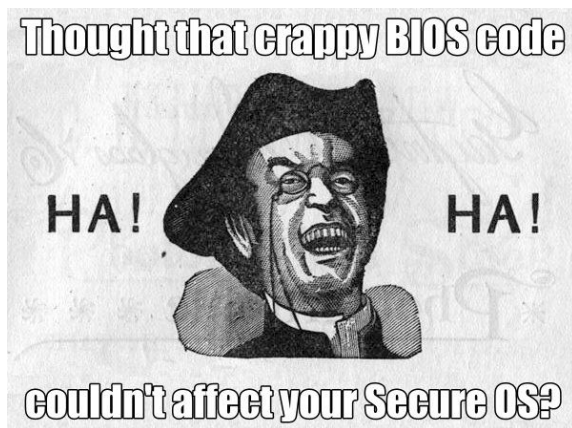


- Hardware triggered (SMI pin)
  - can also trigger from software with help from the chipset (port I/O)
- Entered from any mode at any time, unmaskable
- Saves the entire CPU state on entry, restores on resume
  - to protected area of RAM: SMRAM
- Hard to detect from kernel or userspace
  - interrupt counters, timing analysis
  - hardware on the SMI pin (got a soldering iron? start with Arduino lesson 1)




# BIOS, the Cause of and Solution to...

- The BIOS (usually based on UEFI nowadays)
  - configures SMIs for desired events
  - installs/configures SMI handlers (usually shared UEFI DXE drivers)
  - designates the region of RAM that should be used as System Management RAM (SMRAM)
  - sets chipset registers to protect access to SMRAM
  - sets chipset registers to protect access to firmware (BIOS) flash



# Flash

- BIOS is usually stored on flash
    - connected via a Serial Peripheral Interface (SPI) bus
  - Kind of important to not let attackers modify it
- 
- SMM can be used to police writes to the flash (we'll talk again on this)
  - Not the only protection
    - protected regions, protected ranges, lockdowns, command sequencing, oh my
    - Intel noticed vendors struggle with securing flash, so they patented a solution
      - "Platform firmware armoring technology" aka BIOSGuard, patent WO2012039971A2
  - tl'dr = owning SMM doesn't necessarily mean you own flash

# SMM Attacks

Two broad SMM-related vulnerabilities:

- getting SMM code exec
- subverting SMM protections



# First, SMRAM

- System Management RAM (SMRAM) - dedicated by the BIOS to SMM
  - default 64/128kB @ 0x30000 on reset, can be up to 4GB located anywhere there's space
    - Intel provide some recommendations, e.g. 0xA0000, same place as legacy video buffer
- Where, exactly? SMBASE - a private (SMM only) CPU-internal register
  - there's a copy in SMRAM save-state area (populated on SMI)
  - only the CPU knows for certain where SMBASE is
- SMRAM is meant to be locked down 🔒 - you're not supposed to be able to fiddle with SMRAM from outside SMM

# SMRAM Convolutions and Protections

- Can't trust the MMU & IOMMU to protect SMRAM (controlled by the kernel)
- Solution: the Memory Controller Hub (MCH)
  - create a bunch of chipset registers to protect the SMRAM region
- Intel have added these protections over time and according to certain feature sets
  - details vary substantially depending on the exact chipset
- So, we have a rather large and convoluted set of registers, all of which need to be set correctly to be secure
  - what could possibly go wrong?



[Domas, 2015](#) Memory Sinkhole (APIC remap)  
paper for a good summary of the bits and pieces.

# SMM Code Injection

Choices, choices:

1. subvert SMRAM protections
  - write directly or indirectly to SMRAM
2. subvert non-SMRAM code called by SMM
  - hook the unprotected called code



Fig 1. Traditional Ming Dynasty era BIOS construction technique

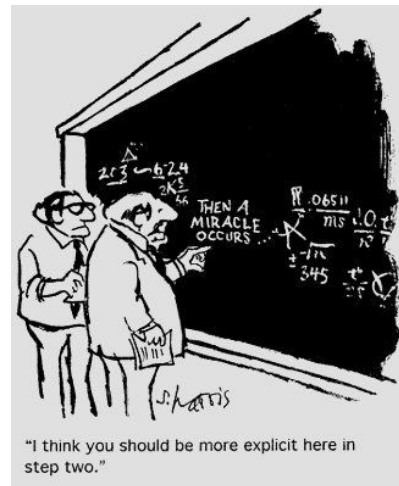
*oops, I forgot to lock my door*

*vs.*

*oops, maybe I shouldn't have contracted the local bikie gang to install my safe*

# (Un)Protected SMRAM

- Forget to lock SMRAM, win the "Inaugural SMM Vulnerability" award
- to lock SMRAM means primarily to set the SMRAM Control Register correctly (D\_LCK, D\_OPEN, see refs at end)
- Some vendors simply didn't set the D\_LCK bit
- Demo'd by Duflot in 2006 on OpenBSD
  - starting with root...
  - set D\_OPEN via PIO, write a custom SMI handler to SMRAM (which OpenBSD thinks is video RAM and now exposes via /dev/xf86)
  - trigger SMI (wave hands here)
  - profit



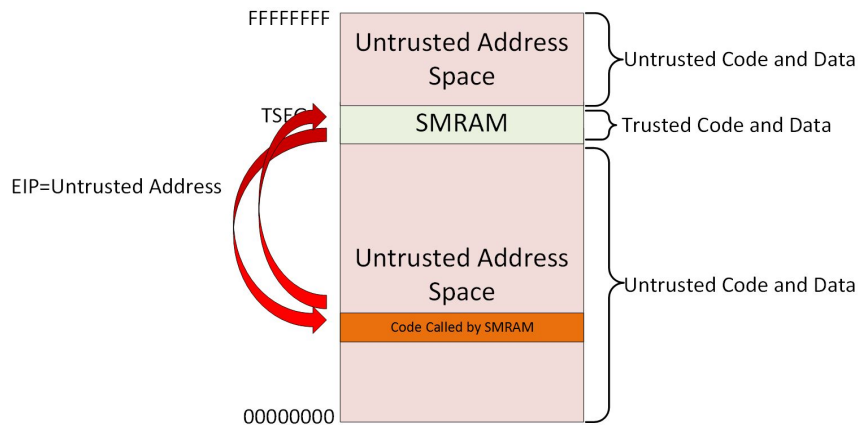
# SMM Execution from Not-SMRAM

- Even when SMRAM is properly protected, various (legacy) chipset features can be abused to write to SMRAM or otherwise cause SMM handlers to run code not actually in SMRAM
  - memory remapping for fun and profit
    - GART remapping and a confused USB controller ([Dufлот, Absil 2007+2010](#))
    - PCI / MMIO memory reclaim ([ITL, 2008, 2009](#))
    - APIC remap sinkhole ([Domas, 2015](#))
  - sleeping on the job - exploiting sleep/resume
    - DMA-after-snooze ([Oleksiuk, 2015](#))
    - ACPI sins of omission ([osxreverser, 2015](#))
  - ...



# SMM Execution from Not-SMRAM

- calling out to non-SMRAM
  - ACPI callouts [ITL 2009](#)
  - many others [Legbacore, 2015](#)
  - cache fun [Dufлот 2009](#), [ITL 2009](#)



*BIOS and SMM Internals, OpenSecurityTraining*

# When the SMM Enforcer trips

- SMM is omnipotent, so it's a great place to police access to SPI flash, right?
  - when BIOS\_CNTL[BLE] is enabled, an SMI is generated on a SPI flash write attempt (setting of BIOSWE), and the SMM handler is expected to "deny" the write by unsetting BIOSWE



[https://www.youtube.com/watch?v=2A7vAbq\\_uIY](https://www.youtube.com/watch?v=2A7vAbq_uIY)

# When the SMM Enforcer trips

- SMM is omnipotent, so it's a great place to police access to SPI flash, right?
  - when BIOS\_CNTL[BLE] is enabled, an SMI is generated on a SPI flash write attempt (setting of BIOSWE), and the SMM handler is expected to "deny" the write by unsetting BIOSWE
- which works wonderfully, right up until you realise you have multiple cores
  - and that your SMM code has to wait until all are in SMM (rendezvous)
  - Speed Racer: [Kallenberg, Wojtczuk, 2015](#)

# Detection

- The One True Way: physical inspection
  - hardware CPU debugging
  - dumping flash (and comparing to known good)
    - some diffs are ok, which ones?
- Otherwise: chipsec
  - and a good dash of hope that there's some indication left behind



# Mitigation

SMM vulnerabilities have been mitigated by some or all of:

- firmware vendors properly using chipset features to protect SMRAM and flash
- using chipsec to check for vulnerable configurations
- Intel's addition of CPU and chipset protections and support for BIOS vendors
  - Haswell / 4th Gen (ca. 2013)
    - SMM\_Code\_Chk\_En = machine check exception on code exec outside of SMRAM when in SMM
    - BootGuard: measured boot / verified boot, firmware is verified before running; occurs very early in Platform Initialisation (PI) code, UEFI ~SEC phase
    - BIOSGuard: "we'll look after the sharp knives", Intel to vendors
    - SMM-transfer monitor (STM): a VMM that operates inside SMM (ca. 2015)

# Mitigation

SMM vulnerabilities have been mitigated by some or all of:

- firmware vendors properly using chipset features to protect SMRAM and flash
- using chipsec to check for vulnerable configurations
- **Basically: pick a good BIOS, patch it, and don't use old hardware.**
  - Intel Core 2 Duo / Pentium 4 / Celeron 4 and older don't have SMM vendors
    - SMM\_Code\_Chk\_En = machine check exception on code exec outside of SMRAM when in SMM
    - BootGuard: measured boot / verified boot, firmware is verified before running; occurs very early in Platform Initialisation (PI) code, UEFI ~SEC phase
    - BIOSGuard: "we'll look after the sharp knives", Intel to vendors

# A Really Nice Example

[Exploiting SMM callout vulnerabilities in Lenovo firmware](#), Oleksiuk (Cr4sh), 2016

- gently fuzzed SMI handler using chipsec, machine hung soon enough
- dug into the UEFI drivers and located an SMM driver that was attempting to call other non-SMM UEFI code that had been unloaded by runtime (hence the crash), and could be replaced by attacker-provided code
- proceeded to implement cross-platform PoC using only chipsec, then productionised for Windows in a couple of variations
  - with help from a third party kernel driver from an endpoint security product: "I found them very useful for local privileges escalation and DSE bypass"
- Flash was safe, Lenovo might not have grokked SMI handlers but did get the SPI flash protections right

# Further Reading / Work

- [opensecuritytraining.info/IntroBIOS.html](https://opensecuritytraining.info/IntroBIOS.html)
  - 828 slides of of SMM and BIOS goodness
- [BIOS and Secure Boot Attacks Uncovered](#), Intel (also covers SMM), 2015
- [Building reliable SMM backdoor for UEFI based platforms](#), Cr4sh 2015
  - and the subsequent posts
  
- AMD?



**Bonus...**

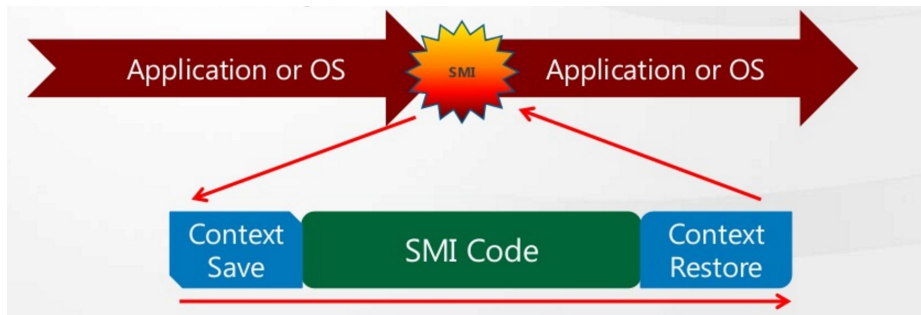
# Where does this fit?

Most low-level attacks take aim at UEFI/BIOS, SMM has been one way. Others:

- [De Mysteriis Dom Jobsivs](#) (Snare, 2011-2012)
  - EFI-based malware incl. leveraging PCIe Option ROMs
- [Practical Hardware Backdooring](#) (Brossard, 2012)
  - take Coreboot + SeaBIOS, make it hostile
- [Thunderstrike](#) (Trammel, 2014; refreshed in 2015)
  - Thunderbolt Option ROMs can hook firmware updates
- [Speed Racer, Darth Venemis](#) (Wojtczuk & Kallenberg, 2015)
  - SMM flash protection race, S3 boot script injection
- [Prince Harming](#) (Vilaça / osxreverser, 2015)
  - Apple forgot to lock the flash on resume from sleep

# Do One Thing and (Not) Do It Well

- SMM is not real time, exacerbated on a multi-core system
- latencies are in the "YMMV" service class
  - hwlat\_detector.ko might help to measure these
- hopefully your power supply won't go up in smoke by the time SMM responds
  - ok, you're probably fine on this one
- hopefully SMM interruptions won't affect your o/s and user space processes
  - building an RTOS? you'll need to talk to your BIOS vendor



*BIOS Customizations for Optimized RTOS Performance, Insyde, 2013*

# Advanced Configuration and Power Interface

- Why do we need SMM when ACPI is a thing?
- SMM predates ACPI (1990 vs. 1996)
- Up to the vendor to decide what they do in SMM vs. ACPI
  - e.g. Apple: seem to be ACPI all the way; some some Lenovo machines (X1) do one SMI on power events but the rest is ACPI



# ICE, ICE, Baby

- Q: how do you debug on an x86 CPU?
- A: In the oooooold days (late 80's), you'd pay a lot of money for In Circuit Emulator hardware
- literally in-circuit



# ICE > On Chip Debugging

- As CPUs got faster, ICE became impractical
  - "[we can't] design an ICE that runs at such a fast CPU speed" (33 MHz)
- so, build the debugging features *into* the CPU
  - eventually In-Target Probe eXtended Debug Port, still \$\$\$\$
- ICE Mode
  - once a breakpoint is triggered, the CPU enters an alternate operating state called "ICE mode"
  - during ICE mode, all memory cycles appear to stop, and the CPU appears to be dormant
  - the ICE-MODE pin acts as an A32 address line, a separate memory space
  - still has access to user memory (undocumented UMOV instruction)
- Sound familiar?

