Put SMM handlers in Linux, not coreboot

Ron Minnich, Google
European coreboot Conference

Outline

- What is SMM?
- SMM in LinuxBIOS
- SMM in coreboot
- Why it will never die
- Why persistent firmware code can be dangerous
- The RISC-V inspiration
- An alternative: Linux can handle the SMIs
- Proof of concept and questions

System management mode

- Introduced with 486 to support power management
  - Close lid, what happens? DOS is not going to handle sleep!
  - Backward compatibility has to work
- Hence, need:
  - Higher priv level than Ring 0 (i.e. beyond DOS)
  - Operations, e.g. sleep, that run without Ring 0 knowing it
  - Additional interrupts vectored to these operations
  - Operations must be deeply hidden so Ring 0 does not break them
    - I.e. in a memory space Ring 0 can never see
    - Protected by one-way-locking registers
  - No state leakage across the boundary, esp. to Ring 0
- Design based on these requirements ends up at SMM
- But note: in the beginning, it’s all about DOS

Evil-ution

- If there is a place to put secret code that can never be seen and is highest privilege, will vendors use it?
- Well, duh ….
- If secret code is written to lowest-common-denominator standards, and handed to random vendors who put in random features designed for customer lock-in, will it be be full of 0-days and nasty bugs?
- That question answers itself
- For security, SMM is a serious problem
- So we want it either gone or under our control

Eliminating SMM

- No feature ever leaves the x86
  - See: DAA, the unused opcode that eats 1/256 of the space
- In some ways, it’s easy: don’t enable it, lock the memory, lock the register that disables it
- In other ways, it’s hard: there might be something about your hardware that would benefit from having it
- SMM model infects other architectures, such as RISC-V
- So this talk is about owning it, not killing it
  - Maybe RISC-V community will listen?
- First, a quick overview of SMM, then a description of our prototype, then some questions

Digression: you don't always need SMM

- Intel rep, 2004, to us at Los Alamos: “You can’t build a working server without an SMM handler”
- Us: “Linux NetworX’s 100K+ systems don’t agree”
- All SMM ever did in my old world (HPC) was cause trouble
  - Performance and security issues
- It was not even part of LinuxBIOS until 2006, 7 years after the project began!
- But a i945-based laptop needed it, so …
  - It’s all Stefan’s fault

SMM basics(discussion for i945/Q35, 32-bit)

- At PO/R hardware sets SMBASE to 0x30000 on all cores
- On SMI, state is saved at SMBASE + 0xfx00 (0xfc00 64-bit)
    - The actual offset used to be somewhat magic but is now standard
    - Sensible to assume worst case, i.e. 0x400
- Code (“stub”) starts at SMBASE + 0x8000
- Idea seems to be that stub would do per-core setup and call handler at SMBASE (i.e. 0x30000)
- How do you differentiate cores for stub and save state?
- By manipulating SMBASE
- Different SMBASE -> different state and stub pointers
- Segment base with assumed 64K (or other) limit

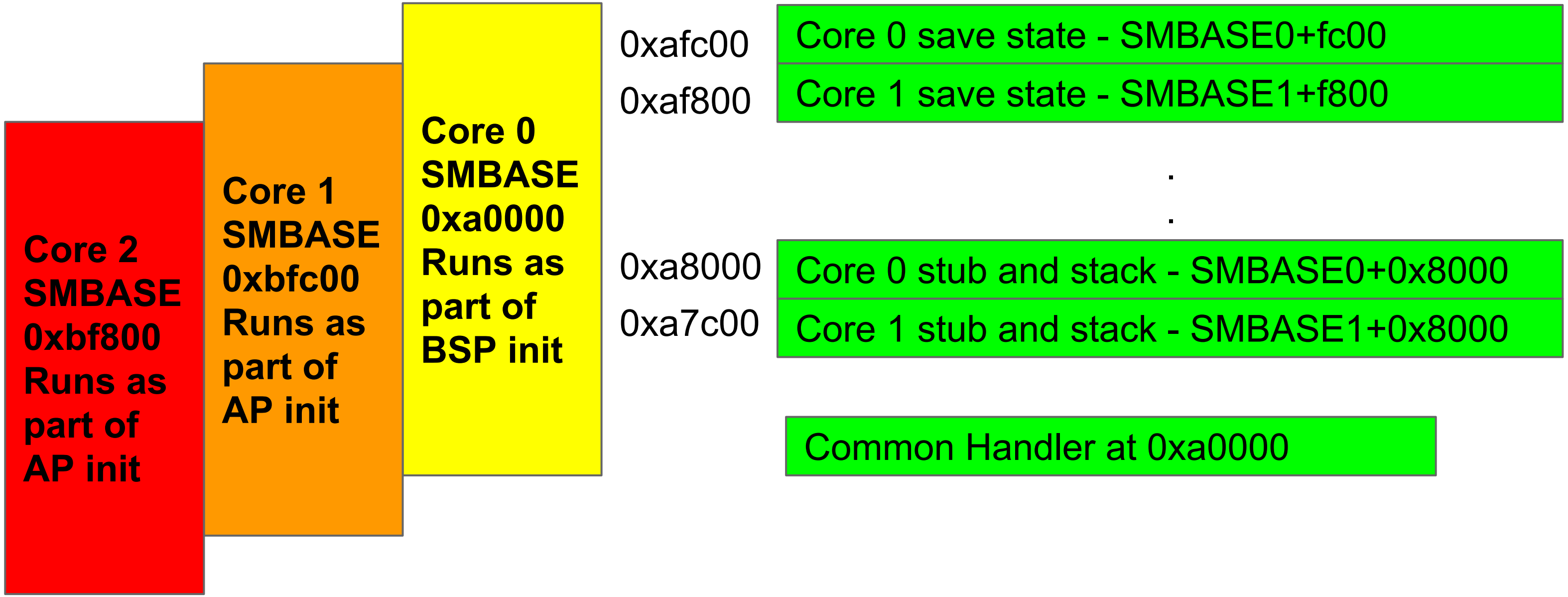
Digression: manipulating SMBASE

- Per-core SMBASE is a kind-of MSR invented before MSRs
- Defines entry point and save state area
- Access to SMBASE register is via SMBASE + 0xfefc (Intel)
- Change SMBASE, address of save state/entry point changes
- So the *next* SMI goes elsewhere
- See coreboot s*/c*/x*/s*/smmrelocate.S
- Note this particular version must be serialized
  - Save state area of SMI is same if SMBASE is 0x30000
- Again: SMBASE is MMIO-accessed, per-core “MSR”
- Strongly enforces hidden nature of SMM

First rule of SMM club is never talk about SMM club

- You can only manipulate SMBASE per-core register in SMM
- So to move SMRAM area, you also have to change SMBASE
- To change SMBASE, you have to go into SMM
- To go into SMM, you need an SMI or write to 0xb2
- Then you can change the hidden SMBASE
- So next time you go to new SMBASE
- But when you change it, uses *previous* SMBASE for RSM
- Am I the only one who finds this all a bit weird?

Smbase in coreboot as described in s*/c*/x*/s*/smmrelocate.S



Per-core smmrelocate.S actions (for i945, etc)

- Determine SMBASE MMIO location
- Compute per-core SMBASE value
- Save it in MMIO location
- Clear SMSTS, PM1STS, EOS
- RSM
- The code has already been set up at 0xa0000 by coreboot
- Back in ramstage, Ring 0 code locks down SMRAM and some other control bits in chipset registers

SMM handler at 0xa0000

- Actual implementation not completely consistent with comments in smmrelocate.S
- You should read both; handler is really well designed
- SMI saves data at SMBASE + 0xFC00 (0xAFC00 on core 0)
- Vectors to SMBASE + 0x8000 (0xa8000 on core 0)
- The actual stub in smmhandler.S is quite nice!
  - The individual code is just a far jmp and done
  - Stack starts at SMBASE + 0x8010
- Common code does everything else based on lapicid
- Has mitigation for LAPIC overlap reported in 2015!
- Shifts to protected mode and jumps to 32-bit handler

Some questions

- Where to run SMM code
  - Is SMI# higher priority than any Ring 0 interrupt including NMI#?
  - Will SMI# interrupt ALL Ring 0 activities? (i.e. unblockable)
  - Can we just run all the SMM code on BSP?
  - If yes, why not just run all SMM on the BSP?
- What is the origin of the “all cores halt” for SMM?
  - Is it because vendor SMM code is not SMP-safe?
  - Linux is SMP-safe
  - If SMI# goes to SMP-safe code, why have all cores spin in SMM?
- Big question: what blocks us from treating SMI# as a super high priority interrupt for the BSP?

Linux implementation questions

- Disable SMM setup in coreboot?
  - Leave it there for now. Just don't lock it down.
- SMP issue
  - That's for you to tell me
- Run special SMM handler in linux that is above, outside, beyond the kernel as in firmware?
  - No. use 64-bit trampoline to get back into the kernel proper
- How to structure the code
  - For now, pull chipset code into linux
  - This is OK IMHO because it seems the SMM chipset stuff is being made very generic

Files changed/added

arch/x86/{Kbuild,Kconfig}

arch/x86/include/asm/realmode.h

arch/x86/realmode/Makefile

arch/x86/realmode/init.c

arch/x86/realmode/rm/Makefile

arch/x86/realmode/rm/header.S

arch/x86/realmode/rm/trampoline_64.S

a*/x*/realmod/rm/trampoline_common.S

arch/x86/realmode/rm/chipset/i82801ix.h

arch/x86/realmode/rm/chipset/i82801ixnvs.h

arch/x86/realmode/rm/smmhandler.S

arch/x86/realmode/rm/smmrelocate.S

arch/x86/realmode/linuxbios.c

arch/x86/realmode/i82801ix.c

Quick aside on file structure

- Kconfig/Kbuild
  - New config variable: LINUXBIOS
- arch/x86arch/x86/include/asm
  - Added smm struct members to real_mode_header
- arch/x86/realmode
  - Linux support code for realmode, built as part of Linux
- arch/x86/realmode/rm
  - Standalone 16-bit stubs and trampolines, assembled into blobs in realmode.bin and then compiled into a struct
- arch/x86/realmode/rm/chipset
  - From coreboot, needed for a few of the chipset-specific bits

Connecting it all together

- Change realmode/rm/ to build 16-bit smm stub and handlers
- Set up Linux-based code in realmode/
- Add options to Kconfig
- Add more files to Kbuild

Changing rm/

Makefile:

```
+realmode-$(CONFIG_LINUXBIOS)   += smmrelocate.o
+realmode-$(CONFIG_LINUXBIOS)   += smmhandler.o
+targets        += $(realmode-y) $(smm-y)
+SMM_OBJS = $(addprefix $(obj)/,$(smm-y))
```

A note on linux realmode/rm blobs

- Write your .S file(s) with exported symbols named pa_xxx
  - E.g. pa_smm_start
- Add pa_ symbols to a*/x86/r*/rm/header.S
  - This defines initializers for a struct
- .S are assembled
- Nm | sed pipeline automagically makes pasyms.h
- That is included in realmode.lds.S
- A few more passes create realmode.elf
- Then realmode.relocs, realmode.bin
- Incorporated into kernel via a*/x*/r*/rmpiggy.S
- rm/ does not assume fixed addresses but smm is special

Using the blobs

- Setup: a*/x*/r*/linuxbios.c calls smm_init()
  - Yep, the coreboot smm_init() works fine in kernel
- Code is mostly the same, save
  - Have to map in 0xa0000
  - Printk looks different
  - More debugging prints :-)
- Not SMP-ready yet!
- Due to my lack of understanding only recently repaired
- One plan: let coreboot do most setup, but not lock memory
- Just change the handler at 0xa0000
  - Doesn’t help NERF (i.e. when Linux embedded in UEFI)
  - Can’t KASLR the SMBASE

Smmhandler is very different ...

- Mainly adapted from linux 64-bit trampoline
- With minor changes due to being in SMM
- One major issue is that we have to run with nonxe=off
- Bug in Linux trampoline

Actual SMM handler in kernel

```
void smm_test(void)
{
        printk("well here I am\n");
}
```

Exciting eh?

Demo time

QUESTIONS

Questions

- Why do this?
  - If we can’t kill SMM, we have to co opt it
  - SMM is appearing on other architectures :-(
- SMP?
  - Yeah
- Model?
  - Program as though it’s a nested NMI?
- What about what SMM does? Sleep?
  - Great question!
-

Where

- https://github.com/rminnich/coreboot/tree/LinuxSMM
- https://github.com/rminnich/linux/tree/smmfromlinux
- Must have at least qemu v2.10
- Linux config: config_smi_linuxbios
- Coreboot config: config-linuxbios
- To run in QEMU, use QRUN file in coreboot
- You need u-root if you want to use my initramfs, see u-root.tk and check with me on how to build (needs Go)
- If you don't use u-root, then just boot and do
  - Outb 0xb2 0
  - However you do IO