rampayloads

Ron Minnich
Google

Outline

- What
- Why
- How

- Is a rampayload
- Why would you do such a thing
- Is it done

What’s a ram payload?

- Payload loaded from the **romstage**
- Hence, it replaces the **ramstage**
- But what is a ramstage?

What is a ramstage?

- Ramstage: discover, allocate, configure, enable resources, before booting
- Implement runtime firmware (SMM, S3, etc.)
- Small on some systems, large on others
- Wait, doesn’t Linux do discover/allocate/enumerate/enable step?
- Yes … kind of
- We never planned on a ramstage; once ram was up we wanted to run Linux
- The original **romstage** for LinuxBIOS, in 1999 … Linux was the rampayload

```
memcpy(0x100000, 0xfff80000, bzimage_size);
((void *)(void) 0x100000)();
```

Why didn’t we do Linux-as-ramstage in the first place?

- We did in 1999 with Linux 2.2
- It did not work
- [digression: avg FSP size in github: ~450KiB. Bigger than Linux 2.2]

The problem

- Early LinuxBIOS boot working ca. Jan 2000
- It all went very well until we saw:
- “...device disabled (BIOS)”
    - Linux, booted from ROM, thought BIOS had disabled something
- **What**? **We *were* the BIOS**!
- Did we do something behind our own back?
- How could BIOS that never ran “disable” a device?
- It didn’t
- Linux 2.2 interpreted “not enabled” as “disabled”

The problem: linux/drivers/block/ide-pci.c (in 2.2.0)

```
/*
 * Setup base registers for IDE command/control spaces for each interface:
 */
for (reg = 0; reg < 4; reg++)
        if (!dev->base_address[reg]) {
                fail...
        }
```

- If anything was going to set the base_address values, it would be Linux
- But it did not, then complained that no one had

Linux 2.2 in 2000 ...

- Was incapable of bringing up, e.g., unconfigured IDE PCI device
- Two options:
  a. Fix Linux
     - Which in 1999 meant dealing with a fairly closed group
     - I’d already tried to get 9p in in 1998 and failed (took another 8 years until 2006!)
     - Linux community not as open as they are today to strange ideas
  b. Fix LinuxBIOS
- We went with Plan b
- So we added LinuxBIOS PCI enumeration based on some Linux 2.2 code
- The result grew into the ramstage
- What we now know as the coreboot device model came in for V2

Perspective from the Third Millennium

- Ramstage has grown over time from a simple PCI configurator
- But Linux has grown in capability
- It can do more of what the ramstage does
- But how much is “more”?
- The “intersection point” of Linux and the ramstage has changed
- Do we always need the ramstage?

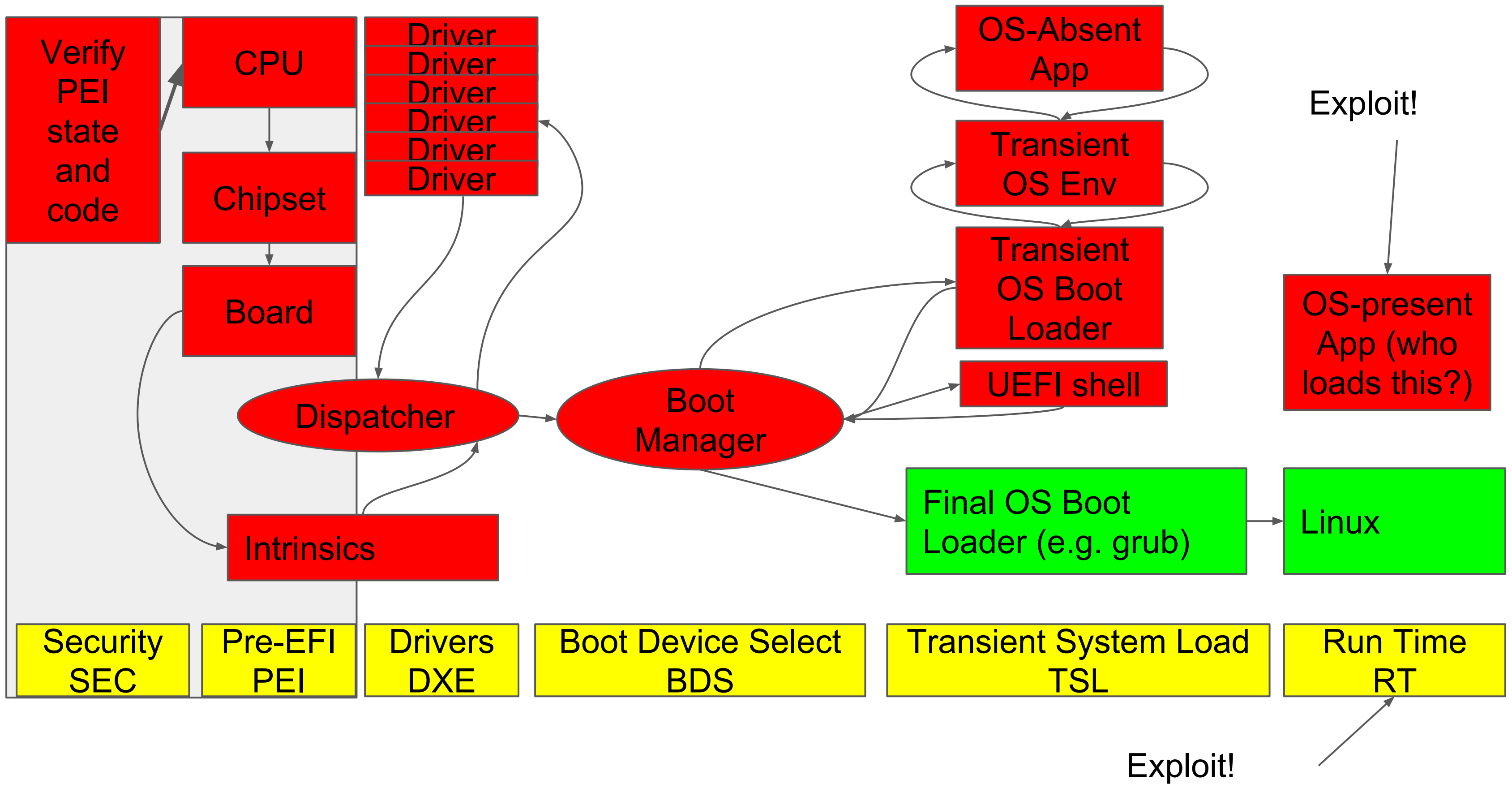
Do we always need the ramstage?

- On many RISC-V boards, the ramstage is or could be almost empty
  - To my regret, people seem to be taking the opportunity to grow it “just because”
- For many cases, we don’t want the services the ramstage sets up
  - SMM
  - S3 resume
  - Runtime services
- On many systems, we don’t need ramstage to enumerate resources
  - They’re simple and hardcoded
  - Linux PCI code is far more capable than it was
  - SMP setup was already done in earlier stages nowadays, so we don’t want it done again
    - Early LinuxBIOS did SMP startup in Linux … until the K7
  - We have other ways to provision things like ACPI
- A lot of ramstage code exists to support other ramstage functions

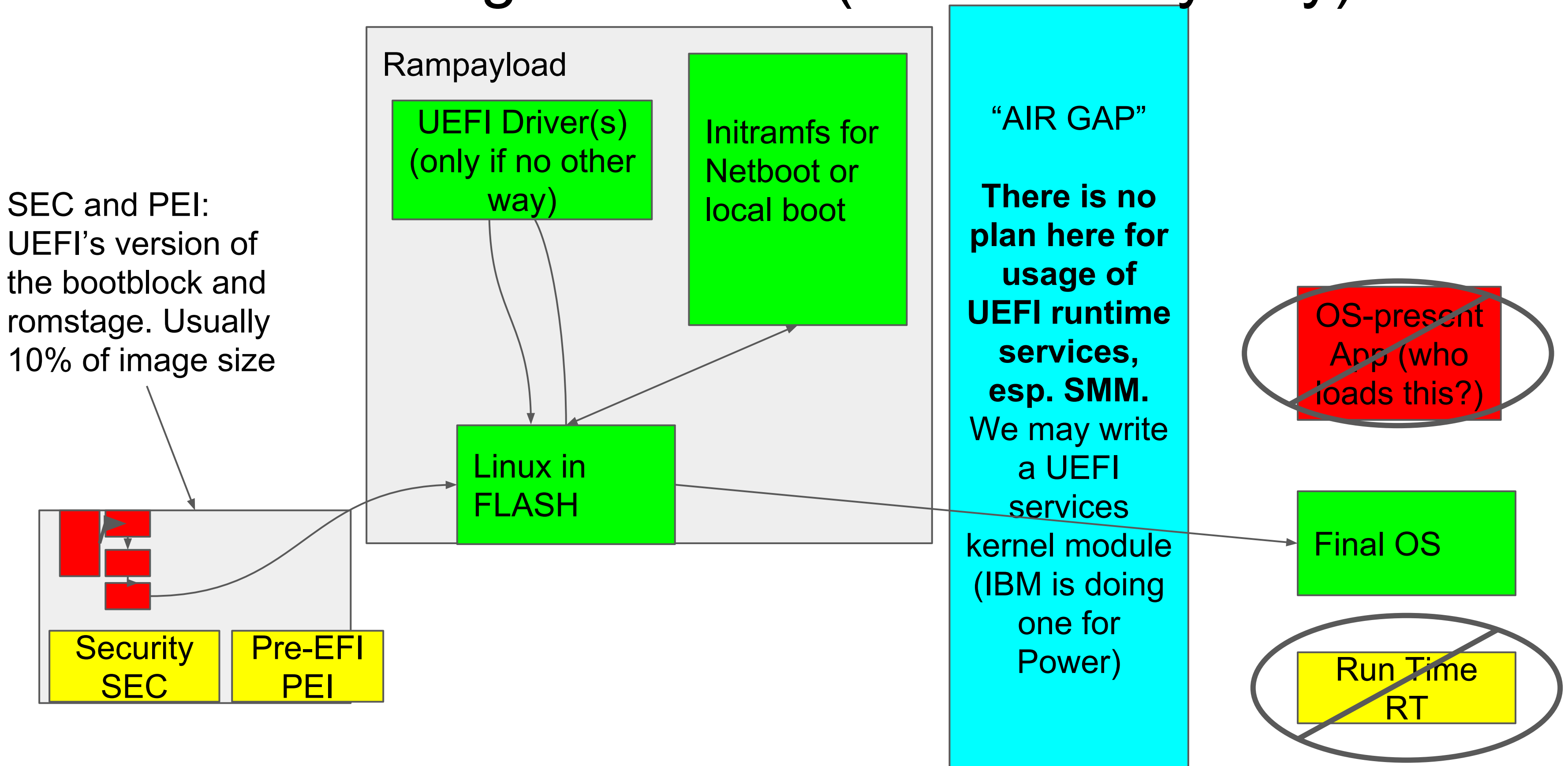
But can we go right to the payload? Maybe ...

- Early RISCV ports could
  - First few chips/FPGA bitstreams had empty ramstage
  - 2015 tests with Linux and Harvey showed we could boot without ramstage
- That’s what we’re hoping to do in UEFI ...

UEFI today with binary blobs in red



Linux-as-"ramstage" in UEFI (closer every day)


Rampayload
UEFI Driver(s) (only if no other way)
Initramfs for Netboot or local boot
Linux in FLASH
SEC and PEI: UEFI's version of the bootblock and romstage. Usually 10% of image size
Security SEC
Pre-EFI PEI
"AIR GAP"
OS-present App (who loads this?)
Final OS
Run Time RT

"AIR GAP"

There is no plan here for usage of UEFI runtime services, esp. SMM. We may write a UEFI services kernel module (IBM is doing one for Power)

Own the firmware

or

Be owned by the firmware

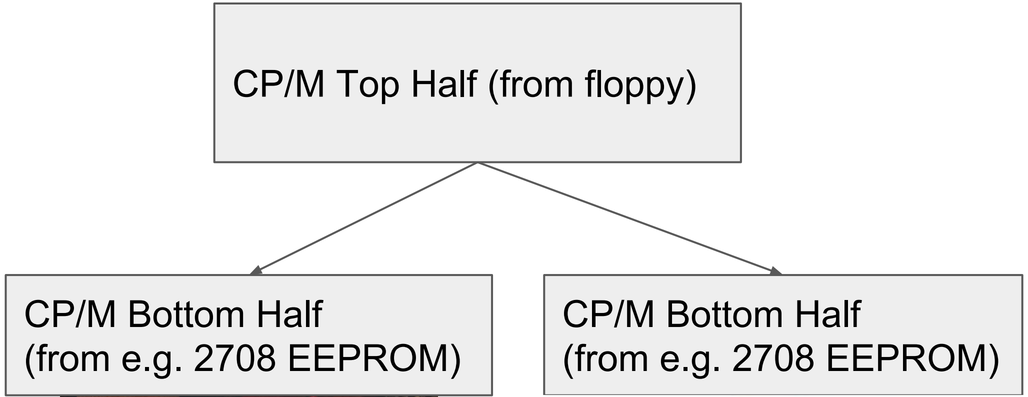
Digression

- Previous slide assumes we can make it work with BootGuard
- We need to fix signing
- Current model is *vendor* keys fuses *permanently*
    - We need owners to do this particularly for second-sale case (e.g. ITRenew)
- We had a discussion yesterday focused on this problem
- Sending *anything* to vendor for signing is unacceptable on any scale
- Need Chrome style model where owner can choose signing keys
- If anyone wants to have this discussion maybe we can talk Friday or Saturday
- We need to move from “docs by RE” to “release the docs!”

The Air Gap is Essential

- Breaks the connection between the OS you are running and firmware
  - I.e. there are no firmware services used at runtime, ever
- This was intentional from the start of LinuxBIOS
  - Locked out SMM in 2000 with no problem
  - In spite of vendor protestations that “that can’t work” … it did
  - Did not want to use ACPI as a former ACPI committee member told us it was exploit heaven
- Kept Air Gap for first 8 years until … laptops
- That’s a story we mostly know but … let’s step back
- Why does BIOS even exist?

1975: BIOS as a Hardware Adaptation Layer


CP/M Top Half (from floppy)
CP/M Bottom Half (from e.g. 2708 EEPROM)
CP/M Bottom Half (from e.g. 2708 EEPROM)

- Why do we have BIOS?
- That goes back over 40 years to the original BIOS and CP/M
- BIOS was one of the first Hardware Adaptation Layers
- Basically a library in EEPROM
- Vendor built hardware, wrote 1KiB of assembly, all done
- One CP/M to rule them all

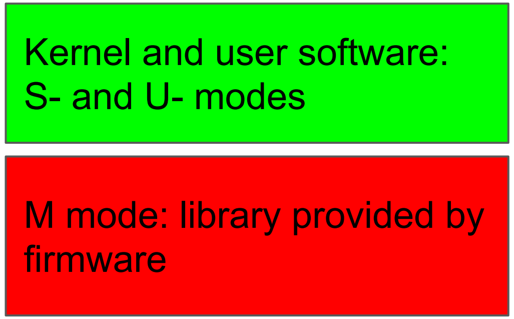
1980: BIOS as a lockin mechanism

- Initially, the OS vendor (Digital Research) made the rules
- *BIOS* had to conform to CP/M *kernel*
- But the IBM PC changed that
- Thus began the world we live in today: BIOS sets the rules
- The tables were turned: *kernels* had to conform to *BIOS*

Take your choice: today firmware is

- 8192 times larger
- Hardware Adaptation Layer
- Vendor lockin tool
- Complex operating system
- Large collection of built-in vulns/exploits
- Extremely inefficient way to access services
- Something you never want to use after you boot

BIOS mentality drives our thinking: e.g. RISC-V


Kernel and user software: S- and U- modes
M mode: library provided by firmware

- RISC-V envisions traps and calls from U- and S- mode to M-mode, which is a library
- What's this look like? Oh, right 1980
- Do we have to do this? No
- But we're used to it so we keep doing it
- There's no reason that the kernel can't supply M-mode code
- At least one RISC-V company I'm talking to will follow this model: kernel supplies M-mode after boot

What’s wrong with M-mode firmware library: security

- Why would you trust it?
  - It’s likely written by the same people who (mis)wrote your other firmware!
- How/when do you update?
  - Full flash updates come with huge risk at any scale
  - Always worth avoiding
- If you have an M mode security fix
- And kernel supplies M mode code
- You reboot to apply the fix
- Not reflash!

What’s wrong with M-mode firmware library: performance

- What is the ABI for calling firmware-supplied M mode?
- Maximally conservative
- Has to support any possible caller, hence has to save maximal context
- Firmware-provided M mode must always assume worst case scenarios
    - Because what’s calling it? It has no idea but it’s likely buggy
- I’ve seen this model before, recently on Blue Gene, and it did not end well
    - One version had library in firmware to “preserve” core IP (came in as blob)
- Kernel called it
    - Firmware had to flush lots of machine state for each call, which made calls very inefficient
    - So engineers just rewrote the firmware for their kernels (or we did anyway …)
- If the kernel owns M mode, then it can optimize call paths

Kernels should supply firmware services they need

- For security, maintainability, ownership
- If there is a “firmware service mode” in the CPU
- Then kernels can supply the code for that service
- This is trivial to do on RISC-V
- It’s not much code and much of it is architecture code, not platform code
- This policy acts against the tendency to drop giant piles of code into M mode
    - “Oh we need this? We can just add another M-mode call”
- It would force a degree of restraint otherwise lacking
- It would also restrain vendor’s impulses to force lockin
- So, with that motivation in mind, back to the rampayload

Test setup

- Rampayload fork on coreboot.org
- romstage loads rampayload, if that fails, loads ramstage
  - Handy for when things go wrong
  - Aside: we need to clean up calls to run_ramstage
    - ramstage calls die() on failure
    - Most calls to it don’t seem to know, and call die() after it (never) returns
    - Maybe call it run_ramstage_or_die as a hint?
- Initial result: works, Linux boots, we get to a shell prompt from a Go shell
  - In QEMU :-)
- Rest of this talk is about making the rest of it work

Changes so far

- 13 files

Changes to payloads: add linuxcheck payload

payloads/Makefile.inc

payloads/linuxcheck/Makefile

payloads/linuxcheck/linuxcheck.c

payloads/linuxcheck/x86config

- Linuxcheck is a regression test to see if Linux can boot and checks:
- Console_out is non-NULL
- Lib_sysinfo.serial is non-NULL
- N_memranges is > 0
- Will grow over time as we find problems
- We did find a coreboot problem: romstage console still seems to be broken?

Kconfig, cbfs_and_run, and uart8250io

● src/Kconfig	● Add RAMPAYLOAD config variable
● src/arch/x86/cbfs_and_run.c	● Add call to run_ramprog()
● src/drivers/uart/uart8250io.c	● CONFIG_RAMPAYLOAD also enables uart_fill_lb()

Add run_ramprog to prog_loaders

- src/include/program_loading.h
- src/lib/prog_loaders.c

- Prototype for run_ramprog()
- Add run_ramprog()
- Add bootmem_write_memory_table()
  - Far simpler than the ramstage version

Tables, and selfboot for romstage

- src/lib/Makefile.inc
- src/lib/coreboot_table.c

- A few changes as to what is compiled
- Disable calls if IS_ENABLED(CONFIG_RAMPAYLOAD):
  - Lb_arch_add_records
  - Arch_write_tables

- src/lib/imd_cbmem.c
- src/lib/romselfboot.c

- Enable cbmem_list if IS_ENABLED(CONFIG_RAMPAYLOAD)
- SELF loader for romstage
  - Much simpler than the ramstage version
  - Most like the early LinuxBIOSv3 code
  - I understand the desire to have one implementation, but ...
  - I’m prepared to argue about that having two is OK
    - ELF defines a file format
    - Many systems parse it: grub, syslinux, linux kernel, kexec
    - With different parser implementations
    - Why worry if two SELF parsers exist?

That’s it so far

- What about ACPI?
- In LinuxBoot on UEFI, we are considering not trusting ACPI tables
  - Part of the “Air Gap”
- Known to have issues
- For coreboot, we might also find the “Air Gap” is desirable
- Still an open question

Summary

- Rampayloads were the original LinuxBIOS design
- Ramstage was created to cover Linux limitations
- Not needed in all cases
- With the desire to eliminate runtime firmware services, rampayloads make more sense than ever
- We are developing them for UEFI, coreboot, u-boot on x86, ARM, RISCV

Demo and Q&A