Digging Into The Core of Boot

Yuriy Bulygin  @c7zero
Oleksandr Bazhaniuk  @ABazhaniuk
Agenda

- Intro
- Recap of MMIO BAR Issues in Coreboot & UEFI
- Coreboot ACPI GNVS Pointer Issue
- SMI Handler Issues in Coreboot
- Write Protections
- Conclusions
Intro to Coreboot
Coreboot

• Coreboot is GPLv2 firmware implementation
• Started as LinuxBIOS in 1999 and renamed to Coreboot at 2008
• Supports x86, ARM, MIPS, POWER8, RISC-V
• Mostly in C, with some ASM. ASL for ACPI tables
• Support multiple payloads (“bootloaders”) to boot Chrome OS, Linux…
  • Depthcharge, SeaBIOS, TianoCore, FILO
• Modular arch to support many CPUs, SoCs, chipsets, devices
• Supports verified boot rooted in hardware write protected firmware
Coreboot Stages

CPU Reset

- Boot Block
  - Cache-as-RAM (NEM)
  - TPM, SPI Init Vboot

Read-Only

- Verstage
  - MemInit, SPD, Ucode update

Read-Write

- ROMStage
  - PCIe enum, SMM, Option ROMs, ACPI

- RAMStage
  - MemInit, SPD, Ucode update

- Payload
  - Vboot kernel, EC FW

Read-Only

- Payload
Chrome OS Boot

Read-Write

Applications
Chrome OS
Kernel -A
Kernel -B
VB_Kernel-A
VB_Kernel -B
Payload
Coreboot RAM stage
Recover System from Verified USB
Recovery Mode

Read-Only

Coreboot Boot Block and Verstage (ROM stage before SKL)
SoC / CPU
Verified Boot

• Verified Boot established signature validation mechanism for Chrome OS
• Root of trust is in read-only initial part of Coreboot firmware protected by /WP pin on SPI devices
• Verstage starting Skylake to reduce amount of read-only ROM stage firmware (as vulnerabilities in RO firmware cannot be patched w/o voiding warranty)
• Read-only firmware verifies RW firmware (new ROM stage & RAM stage)
• Read-write firmware verifies Chrome OS kernel
• Root public key in read-only flash verifies signature of RW firmware keyblock
• Can be disabled in developer mode (requires physically present user)
Recovery and Developer Modes

Recovery Mode

- RO firmware boots signed image on a USB
- Security or hardware failures trigger entering into recovery mode

Developer Mode

- Prior to entering Dev mode, the system erases local state in TPM and on a hard drive
- Root shell is available in Dev mode
- `crosstool dev_boot_usb=1` (boot from USB device)
- `crosstool dev_boot_signed_only=0` (load unsigned binaries)
- `crosstool dev_boot_legacy=1` (allow boot any payloads including MBR systems)
Read-Only Firmware

Chromebook firmware uses Write Protect pin (/WP) on SPI device to protect RO FW

8.4 Write Protect (/WP)

The Write Protect (/WP) pin can be used to prevent the Status Register from being written. Used in conjunction with the Status Register’s Block Protect (SEC, TB, BP2, BP1 and BP0) bits and Status Register Protect (SRP) bits, a portion or the entire memory array can be hardware protected. The /WP pin is active low. When the QE bit of Status Register-2 is set for Quad I/O, the /WP pin (Hardware Write Protect) function is not available since this pin is used for I02.

Winbond W25Q64BV spec

# flashrom -wp-status
WP: status: 0x0094
WP: status.srp0: 1
WP: status.srp1: 0
WP: write protect is enabled.
WP: write protect range: start=0x00600000, len=0x00200000
### SPI Chip Layout in Acer C720 Chromebook

```bash
# futility dump fmap -h /tmp/c720_spi_dump.bin
```

<table>
<thead>
<tr>
<th>SI_ALL</th>
<th>00000000</th>
<th>00200000</th>
<th>00200000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI_ME</td>
<td>00001000</td>
<td>00200000</td>
<td>001ff000</td>
</tr>
<tr>
<td>SI_DESC</td>
<td>00000000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># name</th>
<th>start</th>
<th>end</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI_BIOS</td>
<td>00200000</td>
<td>00800000</td>
<td>00600000</td>
</tr>
<tr>
<td>WP_RO</td>
<td>00600000</td>
<td>00800000</td>
<td>00200000</td>
</tr>
<tr>
<td>RO_SECTION</td>
<td>00610000</td>
<td>00800000</td>
<td>001f0000</td>
</tr>
<tr>
<td>BOOT_STUB</td>
<td>00700000</td>
<td>00800000</td>
<td>00100000</td>
</tr>
<tr>
<td>GBB</td>
<td>00611000</td>
<td>00700000</td>
<td>000ef000</td>
</tr>
<tr>
<td>RO_FRID_PAD</td>
<td>00610840</td>
<td>00611000</td>
<td>0000007c0</td>
</tr>
<tr>
<td>RO_FRID</td>
<td>00610800</td>
<td>00610840</td>
<td>00000040</td>
</tr>
<tr>
<td>FMAP</td>
<td>00610000</td>
<td>00610840</td>
<td>000000800</td>
</tr>
<tr>
<td>RO_UNUSED</td>
<td>00604000</td>
<td>00610000</td>
<td>00000c000</td>
</tr>
<tr>
<td>RO VPD</td>
<td>00600000</td>
<td>00604000</td>
<td>000004000</td>
</tr>
</tbody>
</table>

Read-Only

---
# SPI Chip Layout in Acer C720 Chromebook

| RW_LEGACY    | 00400000 | 00600000 | 00200000 |
| RW_UNUSED    | 003fa000 | 00400000 | 00006000 |
| RW_VPD       | 003f8000 | 003fa000 | 00002000 |
| RW_SHARED    | 003f4000 | 003f8000 | 00004000 |
| VBLOCK_DEV   | 003f6000 | 003f8000 | 00002000 |
| SHARED_DATA  | 003f4000 | 003f6000 | 00002000 |
| RW_ELOG      | 003f0000 | 003f4000 | 00004000 |
| RW_MRC_CACHE | 003e0000 | 003f0000 | 00010000 |
| RW_SECTION_B | 002f0000 | 003e0000 | 000f0000 |
| FWID_B       | 003dfc00 | 003e0000 | 00000400 |
| EC_MAIN_B    | 003c0000 | 003dfc00 | 0001f0c0 |
| FW_MAIN_B    | 00300000 | 003c0000 | 000c0000 |
| VBLOCK_B     | 002f0000 | 00300000 | 00010000 |
| RW_SECTION_A | 00200000 | 002f0000 | 000f0000 |
| FWID_A       | 002effc0 | 002f0000 | 00000400 |
| EC_MAIN_A    | 002d0000 | 002effc0 | 0001f0c0 |
| FW_MAIN_A    | 00210000 | 002d0000 | 000c0000 |
| VBLOCK_A     | 00200000 | 00210000 | 00010000 |
Recap of MMIO BAR Issues in Coreboot & UEFI
Firmware configures chipset and devices through MMIO

SMI handlers communicate with devices via MMIO registers
Recap: “MMIO BAR” Issues

Exploit with PCI access can modify BAR register and relocate MMIO range

On SMI interrupt, SMI handler firmware attempts to communicate with device(s)

It may read or write “registers” within relocated MMIO
Recap: “MMIO BAR” Issues in Coreboot

```
static void mainboard_smi_brightness_down(void)
{
    u8 *bar;
    if ((bar = (u8 *)pci_read_config32(PCI_DEV(1, 0, 0), 0x18))) {
        printk(BIOS_DEBUG, "bar: %08X, level %02X\n", (unsigned int)bar,
               *(*(bar+LVTMA_BL_MOD_LEVEL) & 0xf0);
        if (*(*(bar+LVTMA_BL_MOD_LEVEL) > 0x10))
            *(*(bar+LVTMA_BL_MOD_LEVEL) & 0xf0) = 0x10;
    }
}

static void mainboard_smi_brightness_up(void)
{
    ...
    if (*(*(bar+LVTMA_BL_MOD_LEVEL) < 0xf0))
        *(*(bar+LVTMA_BL_MOD_LEVEL) & 0xf0) = 0x10;
    }
}

int mainboard_io_trap_handler(int smif)
{
    ...
    switch (smif) {
    ...
    case SMI_BRIGHTNESS_UP:
        mainboard_smi_brightness_up();
        break;
    ...
    case SMI_BRIGHTNESS_DOWN:
        mainboard_smi_brightness_down();
        break;
    }
```

- **bar** pointer points to MMIO range of device B1:D0.F0 which can be modified by an attacker.
- SMI handler then uses **bar** pointer to write to LVTMA_BL_MOD_LEVEL offset (when adjusting brightness level).
- SMI handler can be invoked by properly configuring I/O Trap hardware with BRIGHTNESS_UP/DOWN function.
Coreboot ACPI GNVS Pointer Issue
Coreboot x86 SMI Handlers

- smm_handler_start
  - smi_obtain_lock
    - cpu_smi_handler
    - northbridge_smi_handler
    - southbridge_smi_handler
      - smi_release_lock
      - smi_set_EOS

SMIs:
- SLEEP SMI
- APMC SMI
- PM1 SMI
- GPE0 SMI
- GPI SMI
- MC SMI
- TCO SMI
- PERIODIC SMI
- MONITOR SMI
ACPI Global NVS (GNVS) Area

Stores data used to communicate with ACPI and SMM including across S3 sleep:

- SMM interface buffer
- EC Lock function
- Thermal thresholds
- Fan speed
- USB power controls
- ChromeOS Vboot data
- ...
ACPI DSDT

• GNVs area is also defined in DSDT ACPI table
• GNVs layout is platform (SoC/southbridge) specific
• BDW/SKL: CBMEM TOC address at offset $0x18$

```c
/* Miscellaneous */
Offset (0x00),
OSYS, 16, // 0x00 - Operating System
SMIF, 8, // 0x02 - SMI function
PRMO, 8, // 0x03 - SMI function parameter
PRM1, 8, // 0x04 - SMI function parameter
SCIF, 8, // 0x05 - SCI function
PRM2, 8, // 0x06 - SCI function parameter
PRM3, 8, // 0x07 - SCI function parameter
LCKF, 8, // 0x08 - Global Lock function for EC
PRM4, 8, // 0x09 - Lock function parameter
PRM5, 8, // 0x0a - Lock function parameter
...
CMEM, 32, // 0x18 - 0x1b - CBMEM TOC
CBMC, 32, // 0x1c - 0x1f - Coreboot Memory Console
PM1L, 64, // 0x20 - 0x27 - PM1 wake status bit
GPEL, 64, // 0x28 - 0x2f - GPE wake status bit

/* ChromeOS specific */
Offset (0x100),
#include <vendorcode/google/chromeos/acpi/gnvs.asl>

/* Device specific */
Offset (0x1000),
#include "device_nvsaasl"
```
How do we find CBMEM and ACPI tables?
void southcluster_inject_dsdt(device_t device)
{
    global_nvs_t *gnvs;

    gnvs = cbmem_find(CBMEM_ID_ACPI_GNVS);
    if (!gnvs) {
        gnvs = cbmem_add(CBMEM_ID_ACPI_GNVS, sizeof(*gnvs));
        if (gnvs)
            memset(gnvs, 0, sizeof(*gnvs));
    }

    if (gnvs) {
        ACPI_CREATE_GNVS(gnvs);
        ACPI_MAINBOARD_GNVS(gnvs);
        ACPI_SAVE_GNVS((unsigned long)gnvs);
        /* And tell SMI about it */
        smm_setup_structures(gnvs, NULL, NULL);

        /* Add it to DSDT. */
        acpigen_write_scope("\"");
        acpigen_write_name_dword("NVSA", (u32) gnvs);
        acpigen_pop_len();
    }
}
Searching for GNVS in ACPI tables…

- GNVS area is allocated in CBMEM backed by in-memory database (IMD) with CBMEM_ID_ACPI_GNVS

- Pointer to GNVS area is stored in DSDT ACPI table in NVSA field
  - The table can be found with `chipsec_util acpi list` or manually in “Tables” area of Coreboot memory map

- After decompiling DSDT, NVSA field contains GNVS address (0xBFF2D000)

```c
Scope ()
{
    Name (NVSA, 0xBFF2D000)
}
```
Searching for GNVS in CBMEM...

CBMEM_ID_ACPI_GNVS entry in in-memory database (IMD)
So why are we interested in GNVS area?

- GNVS area is allocated during “Write ACPI Tables” boot stage (bs_write_tables)
- A pointer to GNVS area (GNVP) is also stored in CBMEM_ID_ACPI_GNVS_PTR area allocated in CBMEM (on Broadwell based systems and above?)

```c
src/arch/x86/acpi.c
void ACPI_save_gnvs(u32 gnvs_address) {
    u32 *gnvs = cbmem_add(CBMEM_ID_ACPI_GNVS_PTR, sizeof(*gnvs));
    if (gnvs)
        *gnvs = gnvs_address;
}
```
When resuming from S3 Sleep...

```c
void acpi_resume(void *wake_vec)
{
#if CONFIG_HAVE_SMI_HANDLER
 u32 *gnvs_address = cbmem_find(CBMEM_ID_ACPI_GNVS_PTR);

 /* Restore GNVS pointer in SMM if found */
 if (gnvs_address && *gnvs_address) {
   printk(BIOS_DEBUG, "Restore GNVS pointer to 0x%08x\n"
         *gnvs_address);
   smm_setup_structures((void *)__gnvs_address, NULL, NULL);
 }
#endif

 /* Call mainboard resume handler first, if defined. */
 mainboard_suspend_resume();

 post_code(POST_OS_RESUME);
 acpi_jump_to_wakeup(wake_vec);
}
```

Find GNVS_PTR in CBMEM & read GNVP pointer

Update GNVP pointer restored from CBMEM in SMM (if SMI handler exists)

Jump to OS Waking Vector in FACS table
Updating SMM copy of GNVP pointer...

```c
void smm_setup_structures(void *gnvs, void *tcg,
{
/*
 * Issue SMI to set the gnvs pointer
 * tcg and smil are unused.
 *
 * EAX = APM_CNT_GNVS_UPDATE
 * EBX = gnvs pointer
 * EDX = APM_CNT
 */
asm volatile (  
"outb %al, %edx\n\t"
: /* ignore result */
: "a" (APM_CNT_GNVS_UPDATE),
  "b" ((u32)gnvs),
  "d" (APM_CNT)
);  

static void southbridge_smi_apmc(void)
{
  u8 reg8;
  em64t101_smm_state_save_area_t *state;
  
  /* Emulate B2 register as the FADT / Linux expects it */
  reg8 = inb(APM_CNT);
  switch (reg8) {
  
  case APM_CNT_GNVS_UPDATE:
    if (smm_initialized) {
      printk(BIOS_DEBUG,
            "SMI#: SMM structures already initialized!\n");
      return;
    }
    state = smi_apmc_find_state_save(reg8);
    if (state) {
      /* EBX in the state save contains the GNVS pointer */
      gnvs = (global_nvs_t *)((u32)state->rbx);
      smm_initialized = 1;
      printk(BIOS_DEBUG, "SMI#: Setting GNVS to \%p\n", gnvs);
    }
    /* EBX in the state save contains the GNVS pointer */
    gnvs = (global_nvs_t *)((u32)state->rbx);
    smm_initialized = 1;
    printk(BIOS_DEBUG, "SMI#: Setting GNVS to \%p\n", gnvs);
```
SMI handlers never check GNVS pointer

- GNVS pointer is stored in CBMEM area of DRAM which is preserved across S3
- During S3 resume, the pointer is restored from CBMEM in SMM
- SMI handlers use GNVS as a communication buffer with OS (read settings, write results)
- E.g. IOTRAP SMI handler writes byte 0x0 to gnvs → smif (if SMIF value is already 0x32)

→ Limited write primitive in SMM (with controlled address but not the value)
That leads to potential vulnerability on S3 resume

• Attacker could modify GNVS pointer in `ACPI_GNVS_PTR` area in memory (to e.g. overlap with SMRAM) & cause system to enter S3
• Firmware would restore modified GNVS pointer in SMM upon resuming from S3 state
• Attacker then could trigger SMI (e.g. IOTRAP) forcing SMI handler to write/modify memory at controlled GNVS address
• So far only 0x32 → 0 and 0x99 → 0 write primitives found
• Only some systems have this issue:
  • Not all systems with Coreboot store GNVS_PTR (some just store GNVS pointer in SMM once at normal boot like our IVB based Lenovo x230)
  • Not all systems support S3 state
Mitigation Options

- One option is to always store GNVS pointer (GNVP) in SMRAM and not restore from CBMEM as SMRAM is also preserved in S3
- In general, SMI handlers have to check all pointers/addresses for overlap with SMRAM just like EDKII does
SMI Handler Issues in Coreboot
static void smi_interface_call(void)
{
    u8 *mmio = (u8 *)pci_read_config32(PPCI_DEV(0, 0x02, 0), 0x14);
    // mmio &= 0xffff80000;
    // printk(BIOS_DEBUG, "mmio=\%x\n", mmio);
    u16 swsmi = pci_read_config16(PPCI_DEV(0, 0x02, 0), 0xe0);

    if (!((swsmi & 1))
        return;

    swsmi &= ~(1 << 0); // clear SMI toggle

    switch (((swsmi>>1) & 0xf) { 
    case 0:
        printk(BIOS_DEBUG, "Interface Function Presence Test.\n");
        ...
        // write magic
        write32(mmio + 0x71428, 0x494e5443);
        return;
        ...
    case 5:
        printk(BIOS_DEBUG, "Call MBI Functions.\n");
        mbi_call(swsmi >> 8, (banner_id_t *)((read32(mmio + 0x71428) & 0x000ffffff) + OBJ_OFFSET) );
        ...
}
Read base address of GPU MMIO range

static void smi_interface_call(void)
{
    u8 *mmio = (u8 *)(pci_read_config32(PCI_DEV(0, 0x02, 0), 0x14));
    // mmio = 0xffff0000;
    // printk(BIOS_DEBUG, "mmio=%x\n", mmio);
    u16 swsmi = pci_read_config16(PCI_DEV(0, 0x02, 0), 0xe0);

    if (!(swsmi & 1))
        return;

    swsmi &= ~(1 << 0); // clear SMI toggle

    switch ((swsmi>>1) & 0xf) {
    case 0:
        printk(BIOS_DEBUG, "Interface Function Presence Test.\n");

        // write magic
        write32(mmio + 0x71428, 0x494e5443);
        return;

    case 5:
        printk(BIOS_DEBUG, "Call MBI Functions.\n");
        mbi_call(swsmi >> 8, (banner_id_t *)((read32(mmio + 0x71428) & 0xffff0000) + OBJ_OFFSET));

        break;
    
    ...
GPU-SMM MBI Interface in i82830 SMI Handler

```c
static void smi_interface_call(void) {
    u8 *mmio = (u8 *)pci_read_config32(PCI_DEV(0, 0x02, 0), 0x14);
    // mmio = 0xffff80000;
    // printk(BIOS_DEBUG, "mmio=%x\n", mmio);
    u16 swsmi = pci_read_config16(PCI_DEV(0, 0x02, 0), 0xe0);

    if (!(swsmi & 1))
        return;

    swsmi &= ~(1 << 0); // clear SMI toggle

    switch (((swsmi>>1) & 0xf) { // address controlled by exploit
        case 0:
            printk(BIOS_DEBUG, "Interface Function Presence Test.\n");
            break;
        ...
        // write magic
        write32(mmio + 0x71428, 0x494e5443);
        return;
        ...
        case 5:
            printk(BIOS_DEBUG, "Call MBI Functions.\n");
            mbi_call(swsmi >> 8, (banner_id_t *)((read32(mmio + 0x71428) & 0xffffffff) + OBJ_OFFSET));
            break;
        ...
    }
}
```

Read base address of GPU MMIO range

Write “CTNI” magic to offset 0x71428 in GFx MMIO (address controlled by exploit)
Calling MBI functions...

```c
static void smi_interface_call(void)
{
    u8 *mmio = (u8 *)pci_read_config32(PCI_DEV(0, 0x02, 0), 0x14);
    // mmio &= 0xffff8000;
    // printk(BIOS_DEBUG, "mmio=%x\n", mmio);
    u16 swsmi = pci_read_config16(PCI_DEV(0, 0x02, 0), 0xe0);

    if (!(swsmi & 1))
        return;
    swsmi &= ~(1 << 0); // clear SMI toggle

    switch ((swsmi>>1) & 0xf) {
    case 0:
        printk(BIOS_DEBUG, "Interface Function Presence Test.\n");
        ...
        // write magic
        write32(mmio + 0x71428, 0x494e5443);
        return;
    ...
    case 5:
        printk(BIOS_DEBUG, "Call MBI Functions.\n");
        mbi_call(swsmi >> 8, (banner_id_t *)((read32(mmio + 0x71428) & 0x0000ffff) + OBJ_OFFSET));
        ...
```
Calling MBI functions…

- SMI handler reads argument for MBI from SWF16 (SW Flags) register at offset 0x71428 in Graphics MMIO (VGA Display).
- That GPU register is not locked so attacker can control its contents.
- The value of the register is an address to `banner_id_t` structure.

```c
typedef struct {
    u32 mhid;
    u32 function;
    u32 retsts;
    u32 rfu;
} __attribute__((packed)) banner_id_t;
```
Unchecked banner_id pointer... 1/3

- Writes at the controlled address pointed to by `version`

```c
static void mbi_call(u8 subf, banner_id_t *banner_id)
...
    switch(banner_id->function) {
    case 0x0001: {
        version_t *version;
        printk(BIOS_DEBUG, "| - MBI_QueryInterface\n");
        version = (version_t *)banner_id;
        version->banner.retssts = MSH_OK;
        version->versionmajor = 1;
        version->versionminor = 3;
        version->smicombuffersize = 0x1000;
    ...
```
case 0x0201: {
    obj_header_t *obj_header = (obj_header_t *)banner_id;
    mbi_header_t *mbi_header = NULL;
    printk(BIOS_DEBUG, "|-- MBI_GetObjectHeader\n");
    printk(BIOS_DEBUG, "| |-- objnum = %d\n", obj_header->objnum);

    int i, count = 0;
    obj_header->banner.retdsts = MSH_IF_NOT_FOUND;

    for (i = 0; i < mbi_len; ) {
        mbi_header = (mbi_header_t *)&mbi[i];
        len = ALIGN((mbi_header->size * 16) + sizeof(mbi_header) + ALIGN(mbi_header->name_len, 16));
        if (obj_header->objnum == count) {

            int headerlen = ALIGN(sizeof(mbi_header) + ALIGN(mbi_header->name_len, 16), 16);

            memcpy(&obj_header->header, mbi_header, headerlen);
            obj_header->banner.retdsts = MSH_OK;

            if (obj_header->banner.retdsts == MSH_IF_NOT_FOUND)
                printk(BIOS_DEBUG, "||-- MBI object #%d not found.\n", obj_header->objnum);
            break;
        }
        i += len;
    }
}
case 0x0203: {
    get_object_t *getobj = (get_object_t *)banner_id;

    printk(BIOS_DEBUG, "| - objnum = %d\n", getobj->objnum);
    printk(BIOS_DEBUG, "| - start = %x\n", getobj->start);
    printk(BIOS_DEBUG, "| - numbytes = %x\n", getobj->numbytes);
    printk(BIOS_DEBUG, "| - buflen = %x\n", getobj-> buflen);
    printk(BIOS_DEBUG, "| - buffer = %x\n", getobj->buffer);

    int i, count = 0;
    getobj->banner.retrys = MSH_IF_NOT_FOUND;

    for (i = 0; i < mbi_len; ) {
        mbi_header = (mbi_header_t *)&mbi[i];
        headerlen = ALIGN(sizeof(mbi_header) + ALIGN(mbi_header->name_len, 16), 16);
        objectlen = ALIGN((mbi_header->size * 16), 16);

        if (getobj->objnum == count) {
            printk(BIOS_DEBUG, "| - len = %x\n", headerlen + objectlen);

            memcpy((void *) (getobj->buffer + OBJ_OFFSET),
                   ((char *) mbi_header) + headerlen, (objectlen > getobj->buflen) ? getobj->buflen : objectlen);

            getobj->banner.retrys = MSH_OK;
        } 

        i += (headerlen + objectlen);
        count++;
    } 

    if (getobj->banner.retrys == MSH_IF_NOT_FOUND)
        printk(BIOS_DEBUG, "MBI module %d not found.\n", getobj->objnum);

    default:
        printk(BIOS_DEBUG, "| - function %x\n", banner_id->function);
Write Protections
What about write protections?

• Read-Only part of Coreboot firmware in SPI flash devices is hardware write protected in Chromebooks. Yes, with a screw asserting /WP in SPI!
• What if you manually flash Coreboot on a random system?
After flashing Coreboot on Lenovo x230…

[x][ Module: BIOS Region Write Protection
[x][
[*] BC = 0x09 << BIOS Control (b:d.f 00:31.0 + 0xDC)
[ 00] BIOSWE = 1 << BIOS Write Enable
[ 01] BLE = 0 << BIOS Lock Enable
[ 02] SRC = 2 << SPI Read Configuration
[ 04] TSS = 0 << Top Swap Status
[ 05] SMM_BWP = 0 << SMM BIOS Write Protection
[-] BIOS region write protection is disabled!

[CHIPSEC] Modules failed
[-] FAILED: chipsec.modu
[-] FAILED: chipsec.modu
[-] FAILED: chipsec.modu
[-] FAILED: chipsec.modu
[-] FAILED: chipsec.modu
[*] BIOS Region: Base = 0x00500000, Limit = 0x00BFFFFF
[-] FAILED: chipsec.modu
[-] FAILED: chipsec.modu

<table>
<thead>
<tr>
<th>PRx (offset)</th>
<th>Value</th>
<th>Base</th>
<th>Limit</th>
<th>WP?</th>
<th>RP?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR0 (74)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR1 (78)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR2 (7C)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR3 (80)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR4 (84)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
To summarize

- SMM based firmware write protection is off
- SPI protected range registers are disabled
- TCO and Global SMI are not locked down
- SPI config is not locked
- SMRAM can be DMA’d into

- And the system doesn’t use /WP pin on SPI device like in Chromebooks

⇒ Super Crazy Developer Mode
That’s the protection...

```c
static void southbridge_smi_tco(void)
{
    u32 tco sts = clear_tco_status();

    /* Any TCO event? */
    if (!tco sts)
        return;

    // BIOSWR
    if (tco sts & (1 << 8)) {
        u8 bios cnt1 = pci_read_config16(PCH_DEV LPC, BIOS_CNT1);

        if (bios_cnt1 & 1) {
            /*
             * BWE is RW, so the SMI was caused by a
             * write to BWE, not by a write to the BIOS
             *
             * This is the place where we notice someone
             * is trying to tinker with the BIOS. We are
             * trying to be nice and just ignore it. A more
             * resolute answer would be to power down the
             * box.
             */
            printk(BIOS_DEBUG, "Switching back to RO\n");
            pci write_config32(PCH_DEV LPC, BIOS_CNT1,
                (bios_cnt1 & ~1));
        } /* No else for now? */
```
What about Libreboot?

From https://libreboot.org/faq.html#how-do-i-program-an-spi-flash-chip

How do I write-protect the flash chip?

By default, there is no write-protection on a libreboot system. This is for usability reasons, because most people do not have easy access to an external programmer for re-flashing their firmware, or they find it inconvenient to use an external programmer.

On some systems, it is possible to write-protect the firmware, such that it is rendered read-only at the OS level (external flashing is still possible, using dedicated hardware). For example, on current GM45 laptops (e.g. ThinkPad X200, T400), you can write-protect (see ICH9 gen utility).

It’s possible to write-protect on all libreboot systems, but the instructions need to be written. The documentation is in the main git repository, so you are welcome to submit patches adding these instructions.
Conclusion

- Coreboot contains significant amount of platform dependent code
- Platform dependent SMI handlers don’t check pointers
- ACPI NVS is an attack vector as it stored data across S3 sleep state
- Not a lot of public research into Coreboot vulnerabilities
- In Chromebooks, Coreboot uses SPI device’s /WP mechanism and Verified Boot. In other systems, Coreboot is not write protected
- If you want to build and flash Coreboot on x86 non-Chromebooks, enable write protection manually (set BC.SMM_BWP)
Thank You!