About unchecked management

Bruno Pujos

July 16, 2016

Whoami

- Bruno Pujos
- RE, vulnerability research
- LSE 2015
- Sogeti since

About unchecked management

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities
2. Vulnerability
   - Reverse
   - Exploitation
3. Patch
4. Conclusion

Agenda

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities

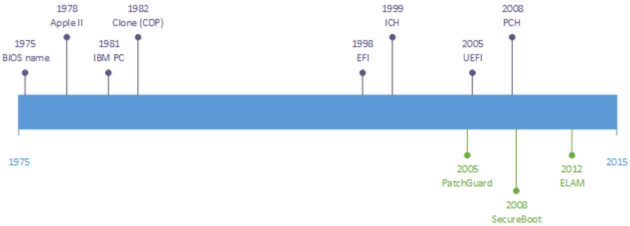
Agenda

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities

UEFI

- **U**nified **E**xtended **FI**rmware
- UEFI is based on EFI
- Specification for firmware development
- Replacing the **B**asic **I**nput/**O**utput **S**ystem (BIOS)
- Community effort organized through a forum

Time line



CDP : Columbia Data Product; PCH: Platform Controller Hub; ICH: I/O Controller Hub

UEFI specification stages

- Security (SEC) Phase
- Pre-EFI Initialization (PEI) Phase
- Driver Execution Environment (DXE) Phase
- Boot Device Selection (BDS) Phase
- Runtime (RT) Phase
- Afterlife (AL) Phase

Protocols

- Drivers communicate using protocols
- Drivers can declare and requests protocols
- Protocols are defined by GUID
- They exposed tables containing function pointers, variables, . . .

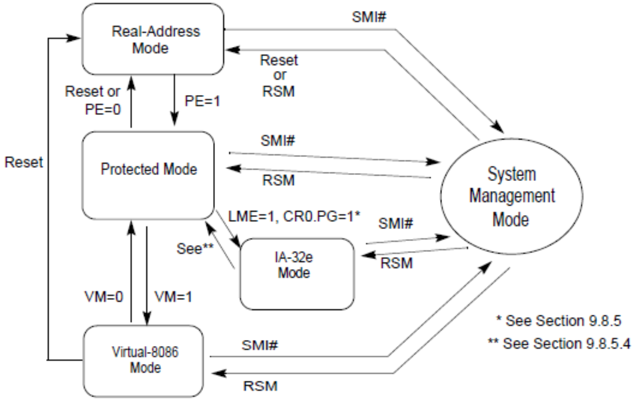
Agenda

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities

SMM

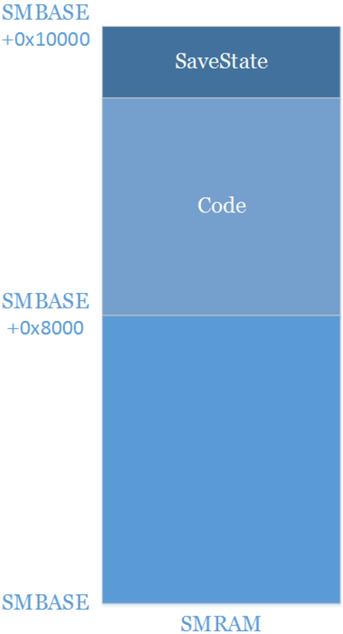
- Not a ring -2 but an Intel mode
- Switch occurred when System Management Interrupt (SMI)
- Different address space (SMRAM) but located in physical memory
- Initialized by the firmware (UEFI)
- In charge to protect and modify the firmware
- Should be protected

System Management Mode



Intel Modes Of Operation (Intel V.3 C.2 P.2)

SMRAM



SMM 101

Initialization

- Can be before DXE
- Change SMBASE
- Add basic handler

SMI handler

- SMI handlers are set mainly during the DXE phase
- SMI are often (only) triggered by the hardware
- SMI handlers are in long mode

SMM 101

SWSMI

- SWSMI are SMI using the IOPort 0xb2 (Advanced Power Management Control)
- Standard way to communicate with the UEFI
- Arguments are passed through the registers

```
mov dx, 0xB2
mov ax, SMINumber
out dx, ax
```

SMBASE

- SMBASE chosen by UEFI
- Must be known for exploitation

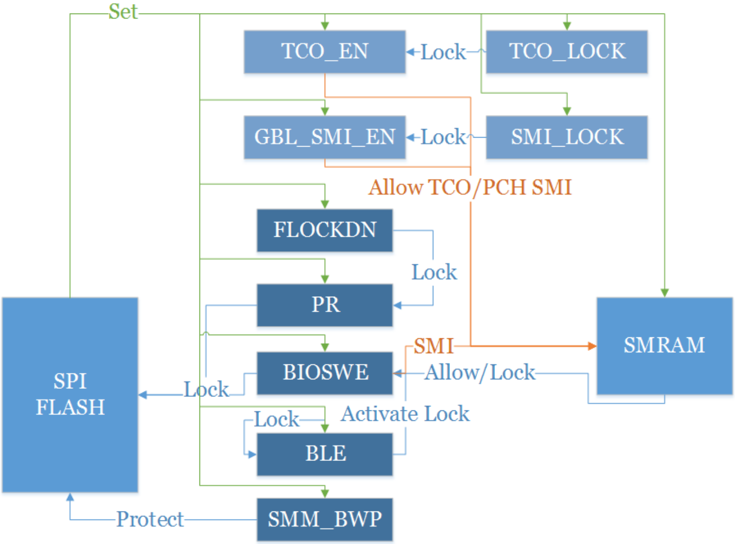
Agenda

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities

Locking mechanism

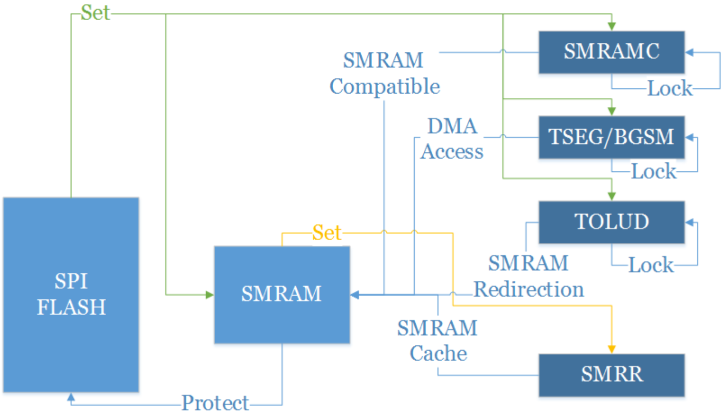
- Preventing corruption
- Root of trust: SPI Flash
- Specification say: if possible lock the flash
- Things to lock in reality:
  - SPI Flash
  - SMRAM

SPI Flash Protection



Configurations
Registers

SMRAM Protection



Agenda

1. SMM & UEFI
   - UEFI
   - System Management Mode
   - Protections
   - Vulnerabilities

Vulnerabilities

- UEFI is ”huge” (300 ”drivers”)
- One fail and it is over
- Main kind of vulnerabilities: memory corruption
- Almost no memory protection (ASLR, NX. . .)

Kinds of vulnerability

- ”Hardware”
- Configuration
- Software

Possible targets

- SMM
- UEFI

SMM attacks

- Only at runtime

Kernel type vulnerabilities

- TOCTOU
- dereference outside of SMM
- NULL dereference
- . . .

"Hardware" type vulnerabilities

- Cache poisoning
- DMA write
- . . .

Agenda

2 Vulnerability

- Reverse
- Exploitation

Agenda

2 Vulnerability
- Reverse
- Exploitation

Target: the firmware

- Dump the firmware from a ThinkCentre M92P (9SKT91A)
- Seems to use protocols from EDK (old Intel framework)
- Contain a lot of references to AMI
- Extracting the drivers (DXE & PEI)

Target: the driver

- Find a driver: `SMIFlash.efi`
- Looks interesting because Flash **and** SMM
- Lets Reverse it!
- *Disclaimer*: All functions and variables names are mine.

SMIFlash.efi

Step

- Initialization
- SWSMI handler

Initialization

- `smm_main` function
- Several variables and protocols recuperation
- Register SwSMI `0x20` to `0x25` with `SwSMIDispatchFunction`

SwSMIDispatchFunction

- Some initialization before a switch by SwSMI
- Recuperate `ECX` and `EBX` from current context
- Combine both for a pointer on a structure (`smiflash_arg`)
- Structure is pass to some functions in the switch
- We will interest ourself only with the SwSMI `0x21`

```
struct smiflash_arg {
    void    *addr_buf; // 0x0
    int32_t offset_bios; // 0x8
    int32_t size; // 0xC
    char    ret; // 0x10
};
```

SwSMI handler 0x21

- Simple SwSMI handler swsmi_handler21
- Read from the SPI Flash (ReadFlash)and write the content into the buffer
- addr_buf is the destination
- offset_bios the reading offset
- size the size to read
- ret a return value
- Basically a memcpy from SPI Flash to memory

```
struct smiflash_arg {
    void    *addr_buf; // 0x0
    int32_t offset_bios; // 0x8
    int32_t size; // 0xC
    char    ret; // 0x10
};
```

Agenda

2 Vulnerability

- Reverse
- Exploitation

Exploitation

Goal Code execution in SMM

Vulnerability

- `addr_buf`, `offset_bios` and `size` are user-control
- There is no check on their value
- `addr_buf` is a physical address
- We can write in SMM where we want and whatever we want as long as it is in the Flash
- Not a real constraint: every possible byte is in the flash

Possibility

- Write a shellcode
- **Relocate the SMRAM**

SMBASE

SMBASE location

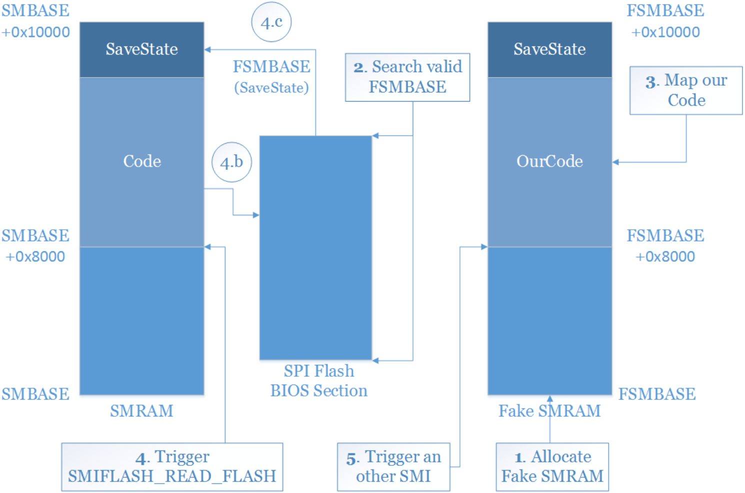
- The SMBASE is needed for relocated the SMRAM
- SMBASE does **not** need to be aligned
- Always the same across reboot (for now)
- Several possibility:
  - RE the SMRAM initialization
  - Guessing it
  - Fuzzing it

Fuzzing SMBASE

- Minimum: `SMRR_BASE - 0x8000`
- Maximum: `SMRR_TOP - 0x10000`
- Probably aligned
- Minimum step: 0x1000

Pretty efficient, but can crash a lot

Exploitation



Shellcode

```
mov ecx, 0x1F3
xor edx, edx
xor eax, eax
wrmsr
mov eax, $realsmbase
mov ebx, ($fakesmbase + 0xFEF8)
mov [ebx], eax
rsm
```

Agenda

3 Patch

Patch

- We reported the vulnerability
- Some time later firmware got an update: 9SKT92A
- Of course I was interested on how they did it
- Let's go reverse!
- Patch is in two part

Patch 9SKT92A: Initialization

- `smm_main` recuperate new informations
- Recuperate the *HOB* list from the `ConfigurationTable`
- Search in the *HOB* list for a structure and copy it
- This structure contain the `SMRAM_BASE` and the `SMRAM_SIZE`

HOB: Hand-Off Block

Patch 9SKT92A: Check

- Add a new `isPointerOutSMRAM` function
- Use `SMRAM_BASE` and `SMRAM_SIZE`
- It take a buffer (`buf`) and a `size` in parameter
- It is used for the first structure and in SwSMI handler `0x21`, `0x22`, `0x23`.
- Check that buf is bellow `SMRAM_BASE` or above `SMRAM_BASE + SMRAM_SIZE`
- Same check for `buf + size`

Patch 9SKT92A: fail

- In SwSMI `0x21` it is used on `addr_buf`
- We can put :
  - `addr_buf < SMRAM_BASE`
  - `addr_buf + size > SMRAM_BASE + SMRAM_SIZE`
- **Fail** we pass the check
- There is not even an overflow check. . .

Patch 9SKT92A: exploit ?

- Really harder to exploit
- Potentially impossible in some firmware, but:
  - `ReadFlash` will potentially stop in the middle in some cases
  - Rewrite the code: potential for a multi-cpu race
  - The overflow can help us
- It is necessary to have an exact layout of the SMRAM
- Exploit will probably depend on the firmware version :(
- But we report it. . .

Patch v2

- Got an update: 9SKT95A
- And an advisory: LEN-4710 !
- Modification in the `isPointerOutSMRAM` function:
  - Check for overflow
  - Check that `buf` and `buf + size` are on the same side of the SMRAM
  - There is even too much check. . .

Agenda

4 Conclusion

Can't we make things better

- Lot of way to fail with a design like that
- Not really anything standardized
- Just a buffer at a static physical address reserved by the BIOS will be a much better idea
- But retro-compatibility (especially in firmware)

Conclusion

- Lenovo are not the only one to be impacted
- Only one to have published an advisory
- 10 constructors at least are impacted
- Probably several thousands computers

The End

Thank you for your attention. Questions ?