SYNACKTIV
DIGITAL SECURITY

Bypassing SMM-EP

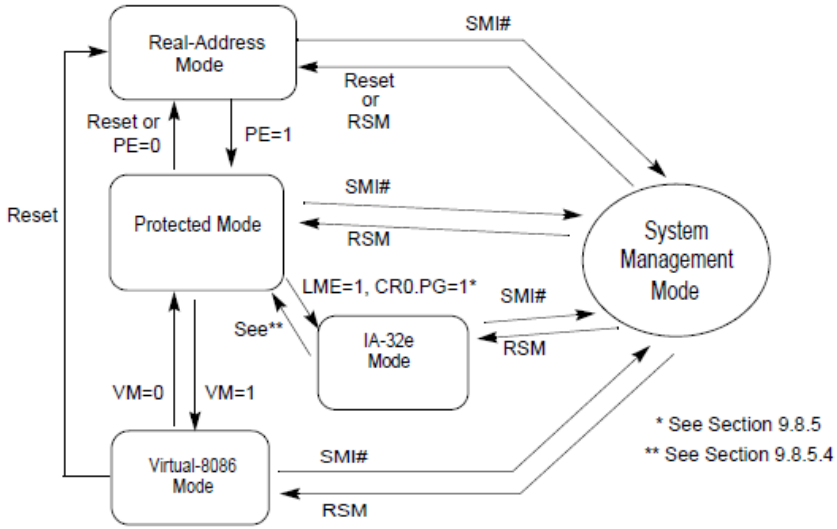
Présenté 12/02/2019

Pour LSE

Par Bruno PUJOS

SMM

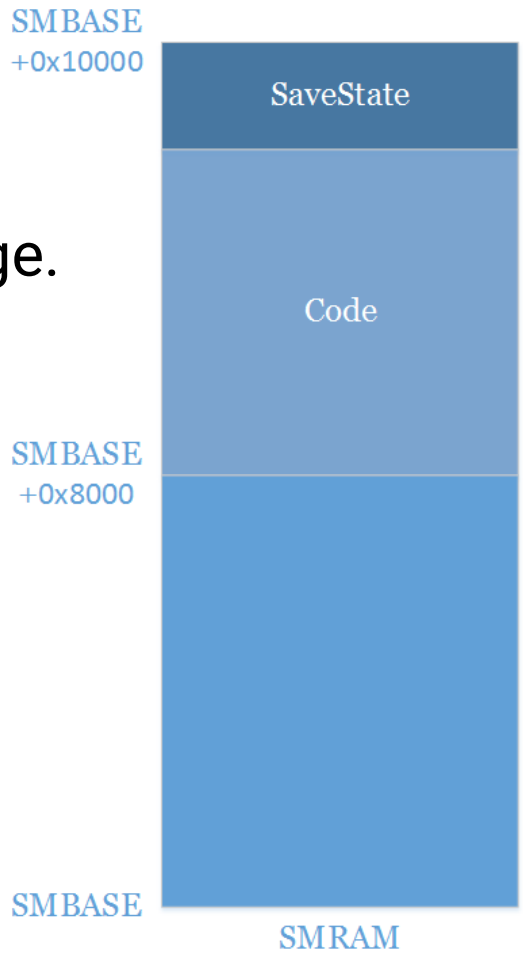
- **S**ystem **M**anagement **M**ode
- A Intel mode
- « ring -2 »



- **S**ystem **M**anagement **I**nterrupt (SMI#)
- Protected address space (SMRAM)
- For management & protection of the firmware

SMRAM

- Initialized during PEI & DXE phases.
  - Protected from « normal » access.
  - SMRR : MSRs which define the protected range.
- SMBASE + 0x8000 : SMM entry point.
  - Have to setup SMI handlers.
- SMBASE + 0xFC00 : Save the state.
  - Contain the SMBASE.

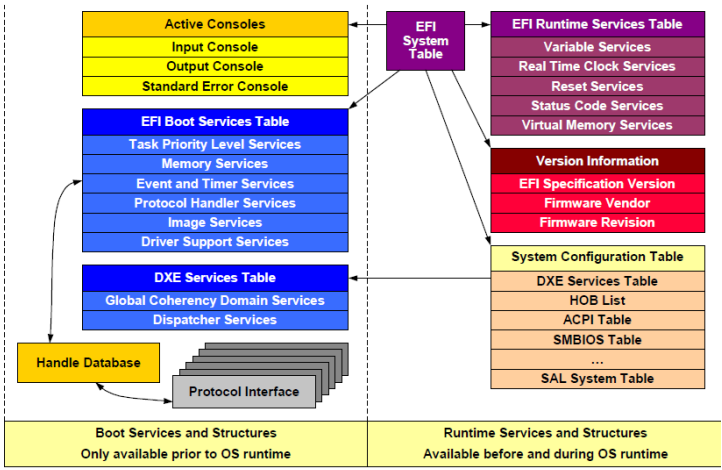


SMI

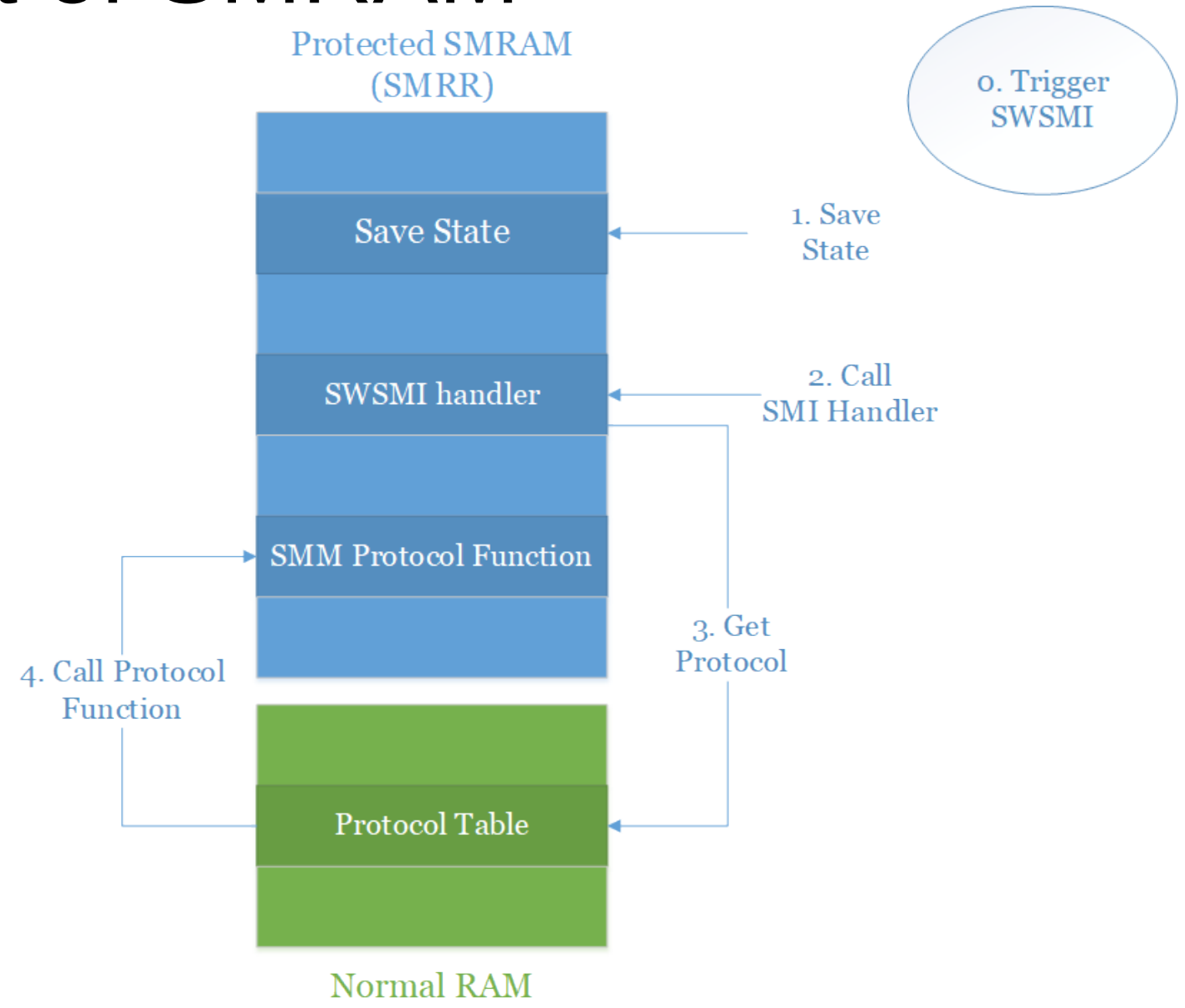
- Different kind of SMI : Timer, USB, ...
- **S**oft**W**are **SMI** (SWSMI) :
  - SMI trigger by IOPort *0xb2* (Advanced Power Management Control).
  - Standard way of communication for OS/SMM.
  - Passage of data is dependent of the code : registers, memory, …
- Usually 64bits in physical.

Callout of SMRAM

- A SMI handler call a function out-side of SMRAM :
  - Code outside of SMRAM is not protected.
  - Attacker maps its code at the place called.
  - SMM code exec !
- Most common type of vulnerability :
  - Services & protocols are registered.
  - Stored/registered in normal world.
  - Called by SMI handler.
  - Vuln !



Callout of SMRAM



SMM_CODE_CHK_EN

- SMM_CODE_CHK_EN is a bit in the MSR_SMM_FEATURE_CONTROL :

When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.

- Basically an equivalent of SMEP for SMM.
- MSR_SMM_FEATURE_CONTROL :
  - Should be accessible only from SMM
  - In practice you can read it from the normal world.

CodeChk & Callout

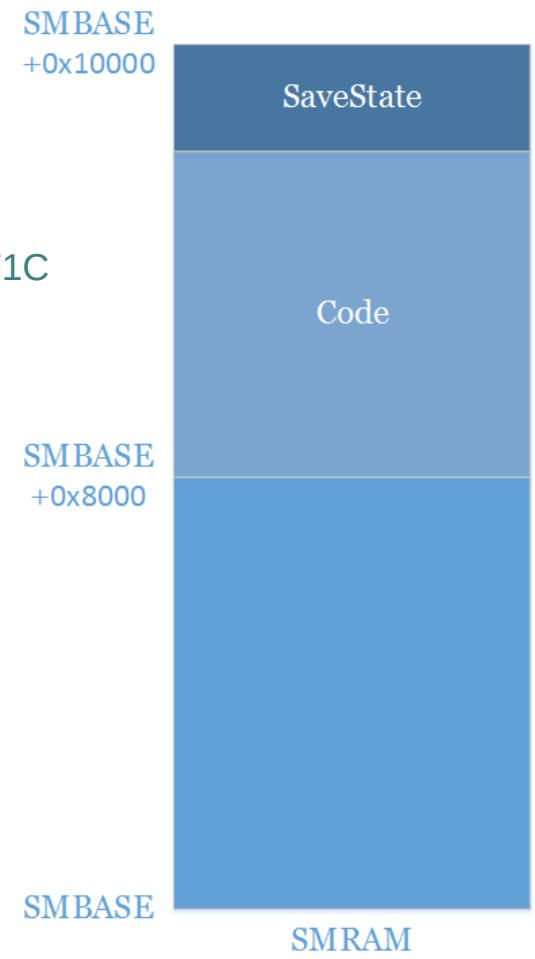
- SMM_CODE_CHK_EN begins to be enable in firmware !
- SMM_CODE_CHK_EN should destroy the callout of SMRAM vulnerabilities, but :
  - Callouts are often trigered by call to protocols.
  - Protocols are tables of pointers.
  - You can still change the pointers in a table if it is accessible from outside of SMM.
  - But you must call code **in** SMRAM.

Bypassing SMM_CODE_CHK_EN

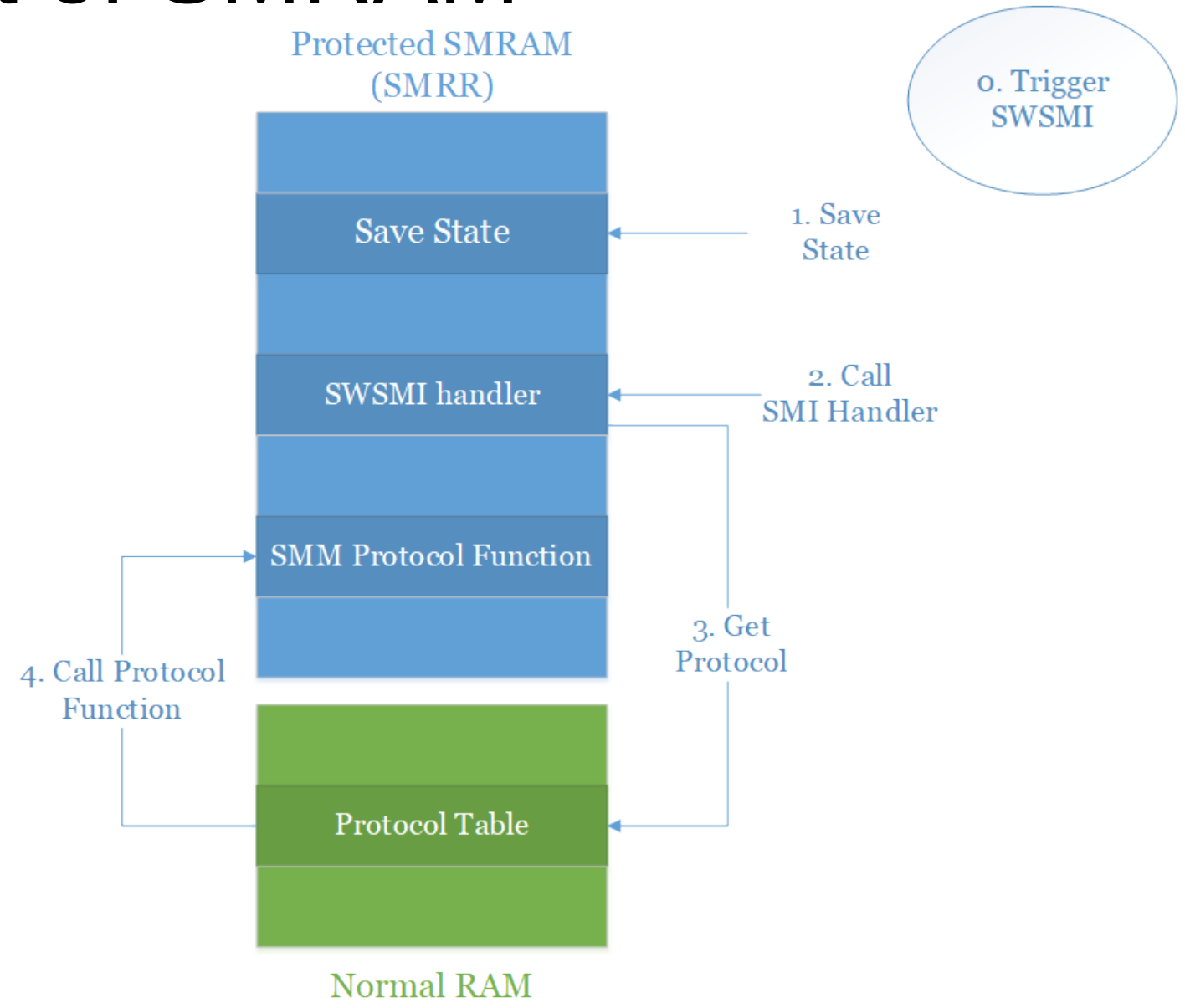
- The SMRAM Save State contains interesting value, in particular the registers :

- 0x80 bytes total

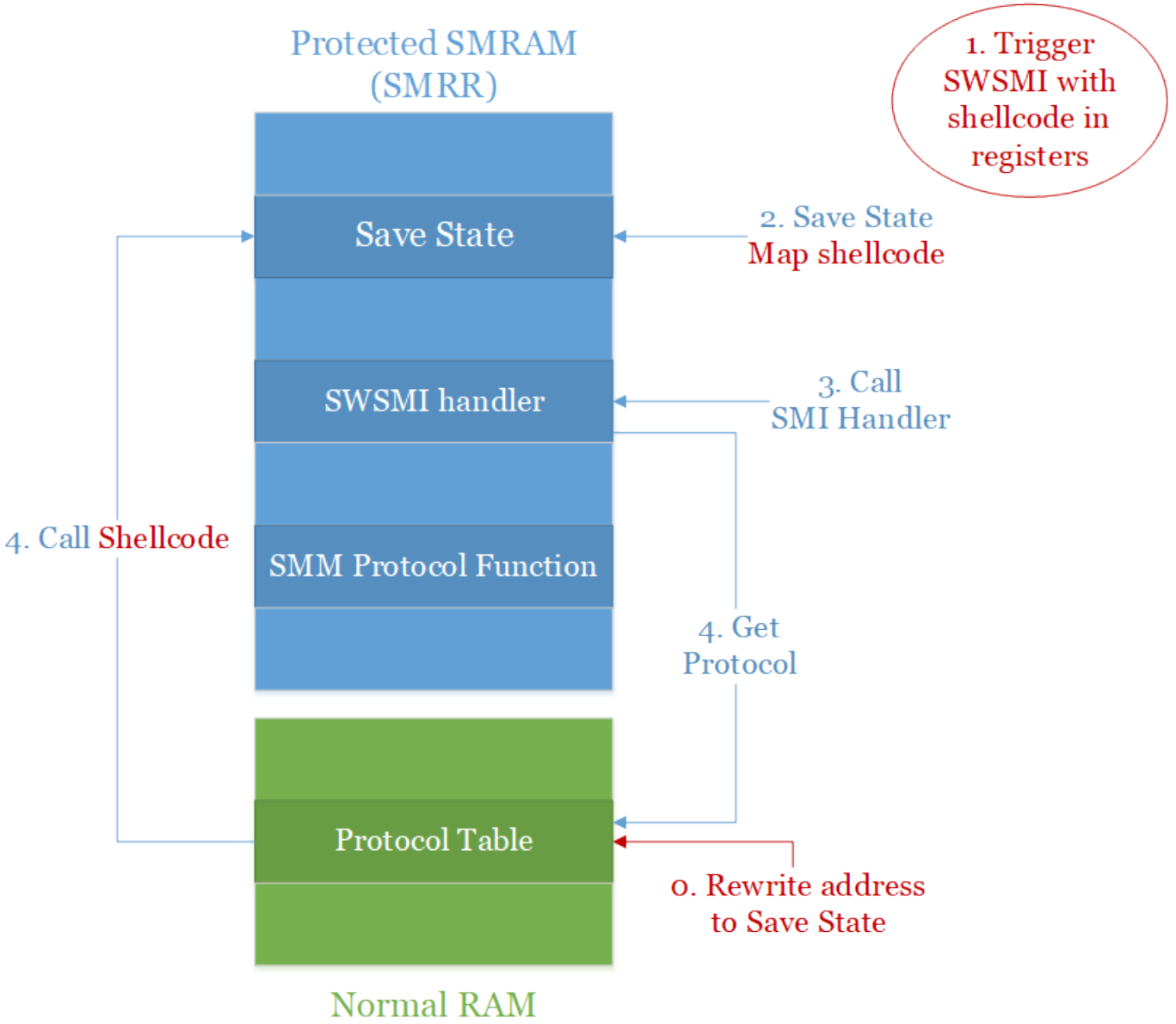
```
typedef struct _ssa_normal_reg {
    UINT64 r15; // start at SMBASE + 0xFF1C
    UINT64 r14; // 0xFF24
    UINT64 r13; // 0xFF2C
    UINT64 r12; // 0xFF34
    UINT64 r11; // 0xFF3C
    UINT64 r10; // 0xFF44
    UINT64 r9;  // 0xFF4C
    UINT64 r8;  // 0xFF54
    UINT64 rax; // 0xFF5C
    UINT64 rcx; // 0xFF64
    UINT64 rdx; // 0xFF6C
    UINT64 rbx; // 0xFF74
    UINT64 rsp; // 0xFF7C
    UINT64 rbp; // 0xFF84
    UINT64 rsi; // 0xFF8C
    UINT64 rdi; // 0xFF94
} ssa_normal_reg_t;
```



Callout of SMRAM



Callout with CodeChk Bypass



SMM Shellcodes

Few practical shellcodes in SMM :

- Disabling the SMRR :
  - Disable SMRAM protections.
  - Can read/write in SMRAM after the shellcode.
  - Does not work with SMM_CODE_CHK_EN. :(
- Modifying SMBASE :
  - Write in the save state a new SMBASE.
  - Trigger a new SMI.
  - Will jump out of SMRAM so will not work. :(
- Could try both at the same time or... simply a good old memcpy.

```
mov ecx, 0x1F3
xor edx, edx
xor eax, eax
wrmsr
```

Wait didn’t you forget something ?

- This works and means we are not dependant of the firmware (Save State is Intel)
- … but how did we get SMBASE ?
- Techniques exist :
  - Guess.
  - Bruteforce from an arbitrary read or write : will make crash.
  - Read MSR IA32_SMBASE (from SMM only).
- What if we don’t want to make the computer crash ?

Getting SMBASE

- First thing tried : find an easy leak.
  - Dump all memory from outside of SMM.
  - Look for pointer(s) on the SMBASE.
  - Nothing :(
- Look where it is initialized :
  - SMM Driver : PiSmmCpuDxeSMM.efi
  - Open-Source in EDK2 but seems modify a little.

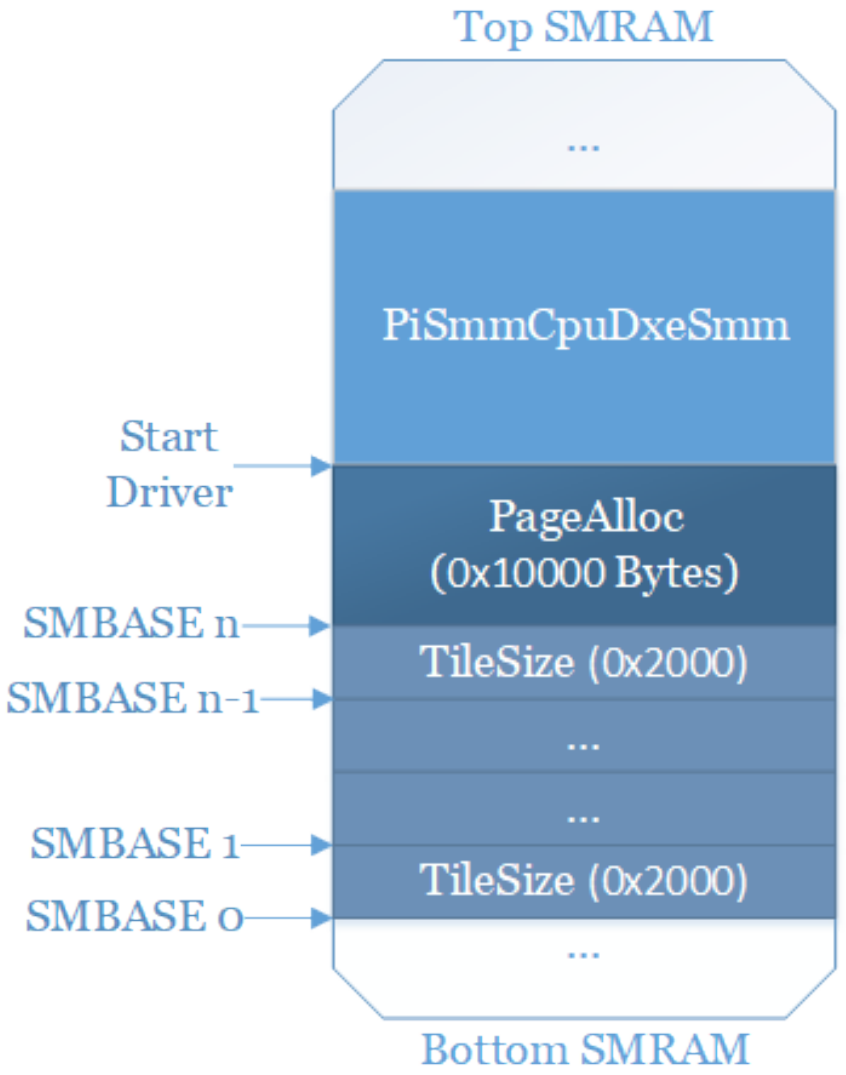
SMBASEs & TileSize

- The PiSmmCpuDxeSMM.efi driver initializes the SMBASE**s** :
  - Necessary to have one SMBASE per CPU.
  - If not 2 CPUs entering SMM will rewrite their save state.
- For RAM space optimization does not reserved 0x10000 for each CPU but just shift enough for not rewritting the Save State.
- Calculation of a *TileSize* in the driver which is always at 0x2000.
- If we got one SMBASE we got them all.

SMBASE Allocation

- The PiSmmCpuDxeSMM needs to reserved the memory (*0x10000 + TileSize * (NumCpu – 1)*).
- Uses the function *AllocateAlignedCodePages* which is a wrapper on *SmmAllocatePages*.
- Possible to ask *SmmAllocatePages* where to allocate memory but by default it uses *AllocateAnyPages* which is equivalent to *AllocateMaxAddress*.
- *SmmAllocatePages* will first look in a freelist and without result it will take the highest possible address.
- SMM drivers are also mapped in memory using this function ! And the last driver map is PiSmmCpuDxeSMM.

SMBASE Memory Map



PiSmmCpuDxeSMM Leak

- If we got an address in PiSmmCpuDxeSMM we know SMBASE.
- PiSmmCpuDxeSMM register a « normal » world protocol :

```
Status = SystemTable->BootServices->InstallMultipleProtocolInterfaces (
    &gSmmCpuPrivate->SmmCpuHandle,
    &gEfiSmmConfigurationProtocolGuid, &gSmmCpuPrivate->SmmConfiguration,
    NULL);
```

- *BootServices* is for the normal world.
- *gSmmCpuPrivate→SmmConfiguration* is located in PiSmmCpuDxeSMM.
- We can locate this pointer and get the address of PiSmmCpuDxeSMM !

Conclusion

- Using an Intel behavior it is possible to map a (small) shellcode in SMM.
- Using a leak in EDK2 (which is used by « everyone ») we can get the SMBASE.
- Combining both we can bypass SMM_CODE_CHK_EN.

- SMM_CODE_CHK_EN is still a good feature :
  - Exploitation is more complex.
  - « Pure » callout of SMRAM are not exploitable anymore.
  - OEM & IBV must fix the vulnerabilities if they want it enable.

AVEZ-VOUS DES QUESTIONS ?

Article : https://www.synacktiv.com/posts/exploit/code-checkmate-in-smm.html

Synacktiv recrute !

MERCI DE VOTRE ATTENTION,

SYNACKTIV
DIGITAL SECURITY