SYNACKTIV
DIGITAL SECURITY

Through the SMM-Glass

And a vulnerability found there.

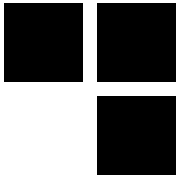
Bruno (@BrunoPujos)

November 18, 2019

Introduction

"It's a poor sort of memory that only works backwards"

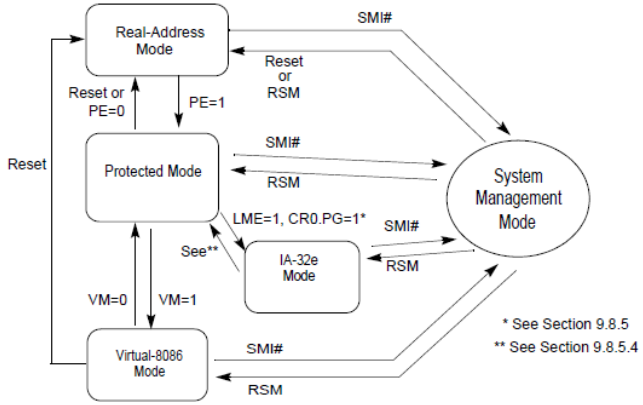
Introduction



- The UEFI/BIOS is stored on a (SPI) flash.
- It is executed before the OS and initializes SMM.
- Main interest for an attacker:
  - Persistence even after OS reinstall.
  - Invisibility to the OS.
- In theory protected once the BIOS give the execution to the bootloader.
- Wait…SMM ?

SMM

- **S**ystem **M**anagement **M**ode
- An Intel Mode..."Ring -2"
- A kind of weird duplicate of all the other modes...



- Initialized during the UEFI boot.
- Its code is part of the UEFI firmware.
- Used for management and protection of the firmware.
- In practice almost the same kind of vulnerabilities than in kernel, except no protections, everything in physical...

SMI#

System Management Interrupt (SMI)

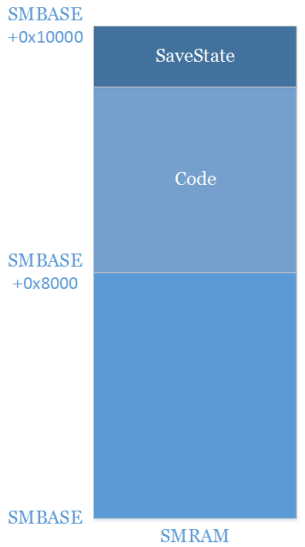
- Interrupt which make the CPU core switch to SMM.
- Different kinds of SMI : Timer, USB, …

SoftWare SMI (SWSMI)

- SMI triggered by IOPort `0xb2` (Advanced Power Management Control).
- Standard way of communication between the OS & SMM.
- Data transition is code dependent: registers, memory, …

SMRAM

- Initialized during the PEI & DXE phases.
- Protected from:
  - "normal" access, protected by the SMRRs (MSR)
  - DMA access
  - ...



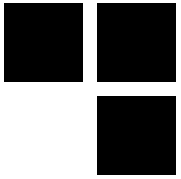
Map of SMRAM

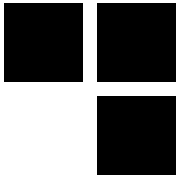
- One SMBASE by CPU Core
- `SMBASE + 0x8000`: SMM entrypoint, after a SMI is triggered, should dispatch to handlers.
- `SMBASE + 0xFC00`: *Saved State* of the previous mode, also contains the SMBASE.

UEFI Services & Protocols

- UEFI is composed of several phases, the main one is the **D**river e**X**ecution **E**nvironment (DXE).
- The DXE phase is composed of hundreds of drivers.
- All drivers are provided with *services*: a set of functions, configuration tables, …
- Drivers can register *protocols*, identified by a GUID for sharing functionality (functions, data, …).
- In practice it is just a pointer link to a GUID which can be retrieved by other drivers.
- Some are well known and documented, some depend on the constructors (OEM, IBV).

Summary

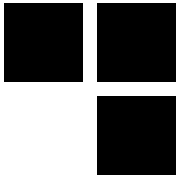




Callout of SMRAM

"You may call it nonsense if you like"

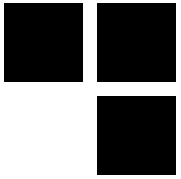
SmmOEMInt15



- Was reversing the firmware of my Lenovo P51s.
- Found a driver named *SmmOEMInt15*.
- Really small: 7 KB and 21 functions.
- The driver registers a SWSMI handler:

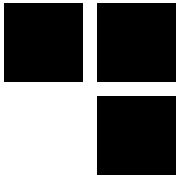
```
// [...]
res = gSmst->SmmLocateProtocol(&UnkProtocolGuid, 0i64, &unk_protocol);
// [...]
swsmi_number = 0xFFFFFFFF;
res = (*unk_protocol)(&swsmi_oemint15_guid, &swsmi_number);
// [...]
// swsmi_handler is the handler function
EfiSmmSwDispatch2->Register(SmmSwDispatch2, swsmi_handler,
    &swsmi_number, &handle_swsmi);
// [...]
```

SWSMI Number



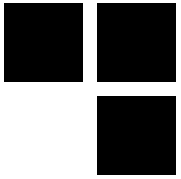
- `SmmSwDispatch2` is a known protocol for registering SWSMI handlers.
- Unknown protocol used for getting the SWSMI number, with GUID: *FF052503-1AF9-4AEB-83C4-C2D4CEB10CA3*.
- Registered by the driver *SystemSwSmiAllocatorSmm*.
- Allow to get a SWSMI number not registered and to associate it with a GUID.
- Install a configuration table in normal world with GUID: *7E791691-5752-4392-B888-EFF9C74F5D77*.
- This table contains a pointer to a list of the registered SWSMI, with the GUID and the number.
- Can be retrieved and enumerated easily from a UEFI shell.

SmmOEMInt15 SWSMI handler



- The SWSMI handler start by getting the content of *RSI* from the *saved state* (user input).
- This value is then used as a pointer on a structure.
- The first two bytes are used as an enum for a switch calling different handlers.
- I did a quick overview of the different handlers and then I arrived to the handler *0x3E00*.

Handler 0x3E00



- The handler gets two other fields from the *RSI* struct and combines them to create a pointer:

```
controlled = ((16 * *(rsi_struct + 0x1C)) + *(rsi_struct + 0x10));
```

- Then it calls an internal functions with the following code:

```
if (! *(controlled + 2)) {
  // [...]
  result = gBootServices->LocateHandleBuffer(
     ByProtocol, &stru_1720, 0i64, &NoHandles, &Buffer);
  // [...] // if result is an error just return
}
```

The vulnerability

Callout of SMRAM

- The *BootServices* are services located in "normal" world.
- We can control the address of *LocateHandleBuffer*.
- A simple callout of SMRAM.
- In the past (~2 years ago) we could just have put a shellcode in normal world and get SMM code execution.
- But `SMM_CODE_CHK_EN` was introduced.

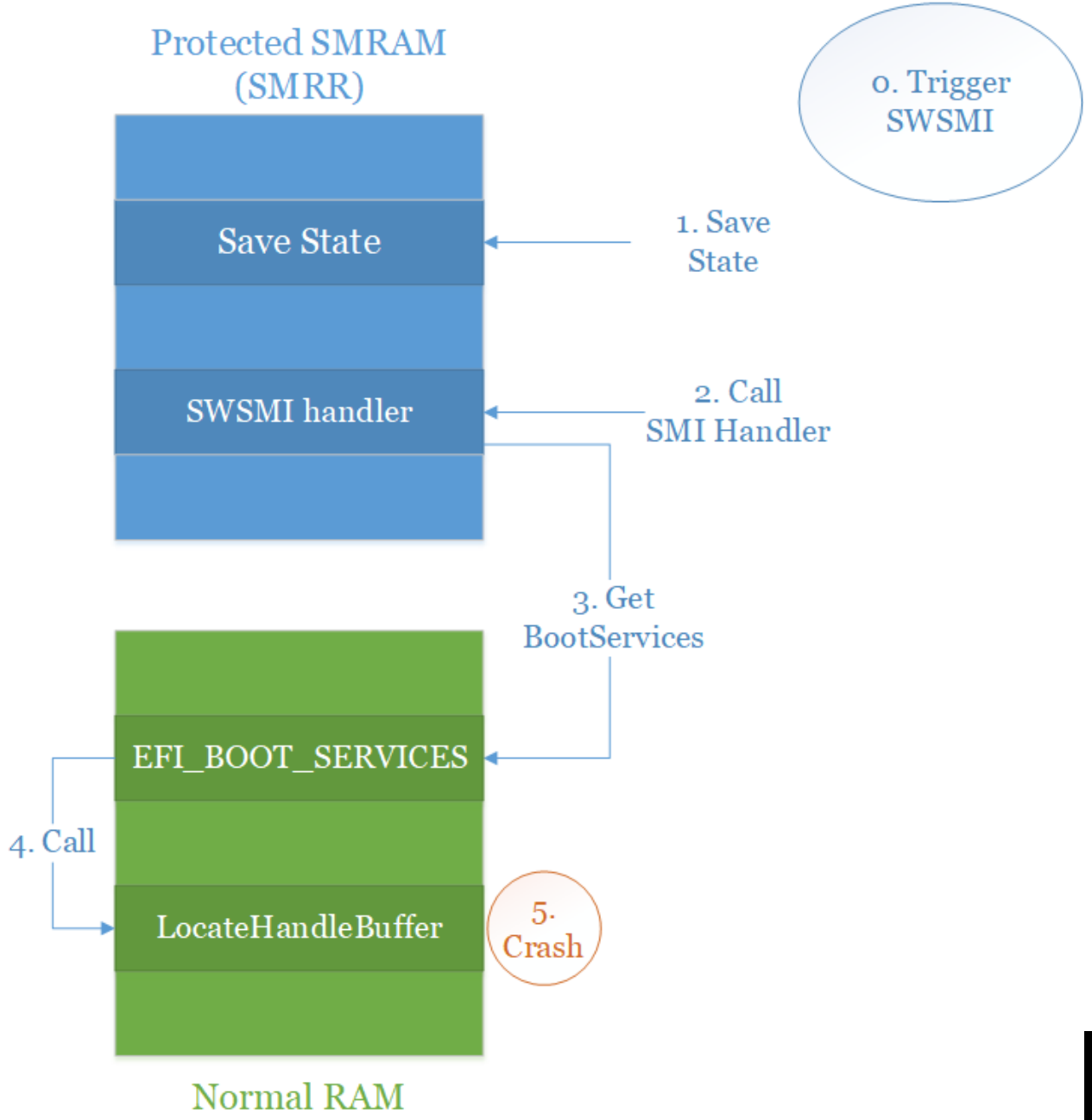
SMM_CODE_CHK_EN

- `SMM_CODE_CHK_EN` is a MSR which can be locked.
- Equivalent of *SMEP* for SMM.
- Easy to check as it can be read from normal world.
- Can't be disabled if locked (even from SMM).

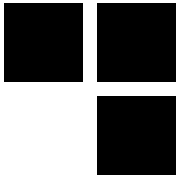
Exploitation

"It takes all the running you can do, to keep in the same place"

Callout of SMRAM

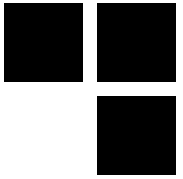


Exploitation problem



- `SMM_CODE_CHK_EN` is a SMEP-like feature for SMM.
- Usual kernel bypass tricks work, but:
  - SMM is a big blackbox and getting control of data is not always obvious.
  - There is no ASLR but in practice address depends of computers and firmware version.
  - Is there something which does not move (too much) and is controlled ?

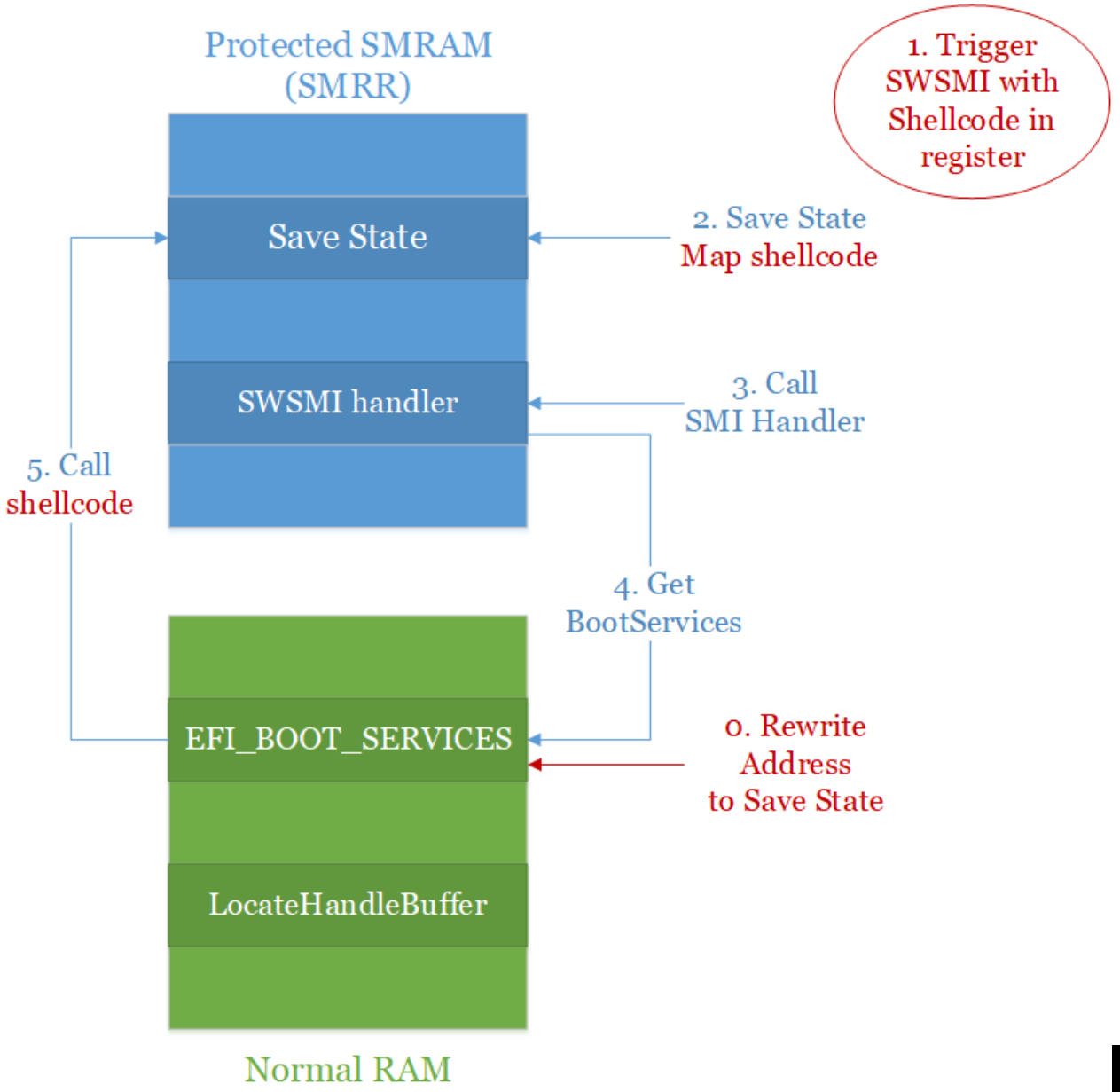
Saved state



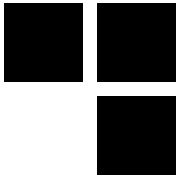
- The saved state when entering SMM contains registers.
- Always at `SMBASE + 0xFC00`.
- 0x80 bytes of registers in total!

```
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;
```

Bypassing CodeChk

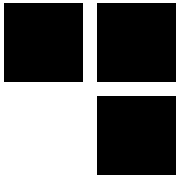


Getting SMBASE



- There is one little problem: SMBASE is unknown.
- One SMBASE by processor and firmware dependent…
- Usual techniques are not great:
  - (Educated) Guess (crash if you're wrong).
  - Bruteforce (will crash).
  - Reading MSR `IA32_SMMBASE` (SMM only, chicken and egg situation).
- Another way ?

SMBASE initialization

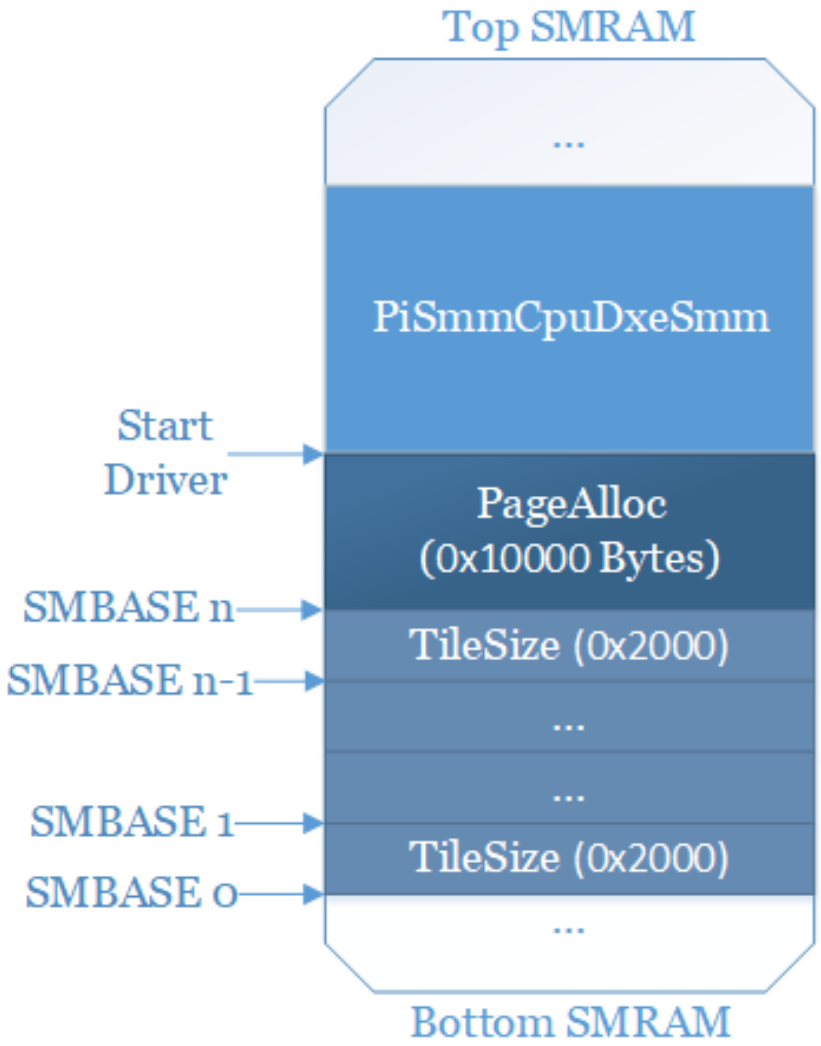


- SMBASEs is initialized by the *PiSmmCpuDxeSMM* driver.
- Open-source driver in edk2.
- For RAM space optimization does not reserve 0x10000 for each CPU but just shift enough for not rewriting the Saved State.
- Calculation of a TileSize in the driver which is always at 0x2000 (of what I have seen).
- If we got one SMBASE we got them all.

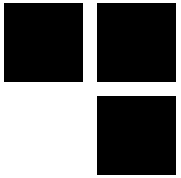
SMBASE Allocation

- The PiSmmCpuDxeSMM needs to reserve 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 mapped is PiSmmCpuDxeSMM.

PiSmmCpuDxeSMM & SMBASE



PiSmmCpuDxeSMM Leak

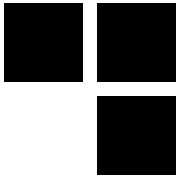


- If we got an address in PiSmmCpuDxeSMM we know SMBASE.
- PiSmmCpuDxeSMM registers a « normal » world protocol:

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

- `gSmmCpuPrivate->SmmConfiguration` is located in *PiSmmCpuDxeSMM* (in SMM).
- But the GUID and pointer are saved in normal world.
- We can get it with `LocateProtocol` and deduce the base address of *PiSmmCpuDxeSMM*.

SMBASE leak: recap



1. Get leak `gSmmCpuPrivate->SmmConfiguration` and calculate the *PiSmmCpuDxeSMM*. (`base = leak - offset`)
2. Deduce the SMBASE address:

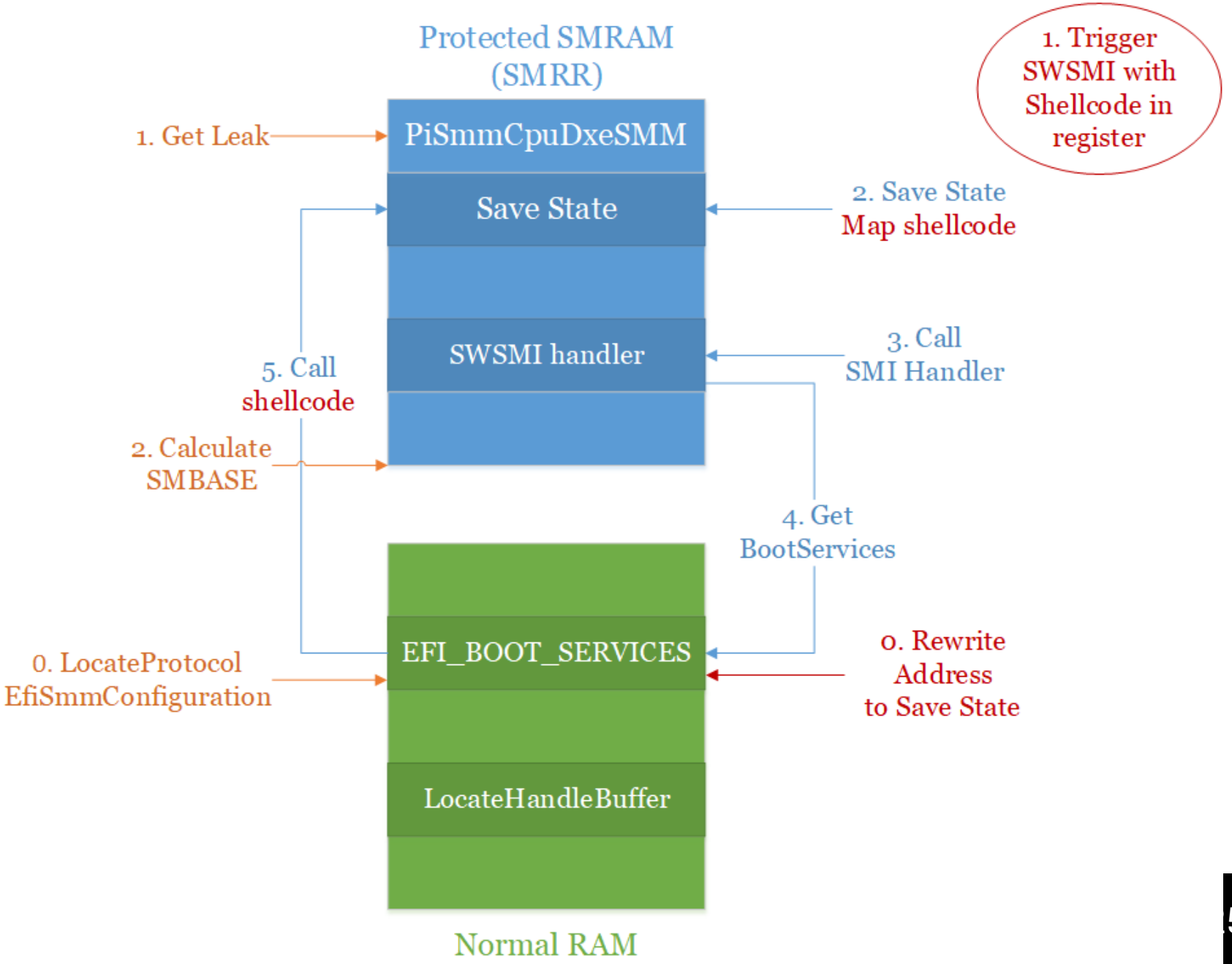
   ```
   base - 0x10000 - tilesize * (numcpu - 1)
   ```

   - `tilesize` is always `0x2000`.
   - `numcpu` can be retrieved using the `EfiPiMpServicesProtocol`.
   - Some firmwares do not correctly set the number of logical processors…
3. Once we have the SMBASE we can calculate the saved state address.

Shellcode ?

- Usual shellcodes for SMM try to:
  - Disable the SMRR: invalidate SMRAM protection.
  - Modify SMBASE: change completely where the SMRAM is located.
- Those do not work with `SMM_CODE_CHK_EN`.
- Maybe possible to modify SMRR and SMBASE.
- In practice 2/3 registers are way enough for dumping and/or rewriting whatever we want in SMRAM.
- Usually rewriting a SWSMI handler is a good way to keep SMM code execution after a shellcode is loaded.

Full exploitation



Conclusion

"When you've once said a thing, that fixes it, and you must take the consequences"

Conclusion

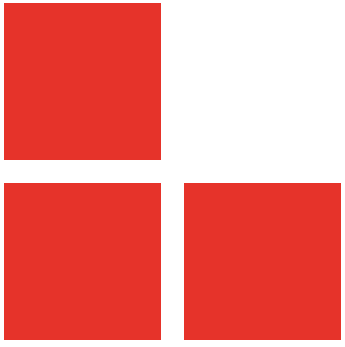
Patch

- This bug was silently patch on Lenovo P51s in August.
- The handler for the command `0x3E00` has just been deleted.
- As the base code crashes it may not have been considered a security fix.

- `SMM_CODE_CHK_EN` is pretty easy to bypass as long as we have SMBASE.
- But because of it callouts of SMRAM are dying because BIOS developers can't use them ;)
- Recent development (ACM) introduce code signing in the firmware: an SMM vulnerability is not enough for getting persistence anymore.

DO YOU HAVE QUESTIONS?

"It's ridiculous to leave all conversation to the pudding!"



THANK YOU FOR YOUR ATTENTION

SYNACKTIV
DIGITAL SECURITY