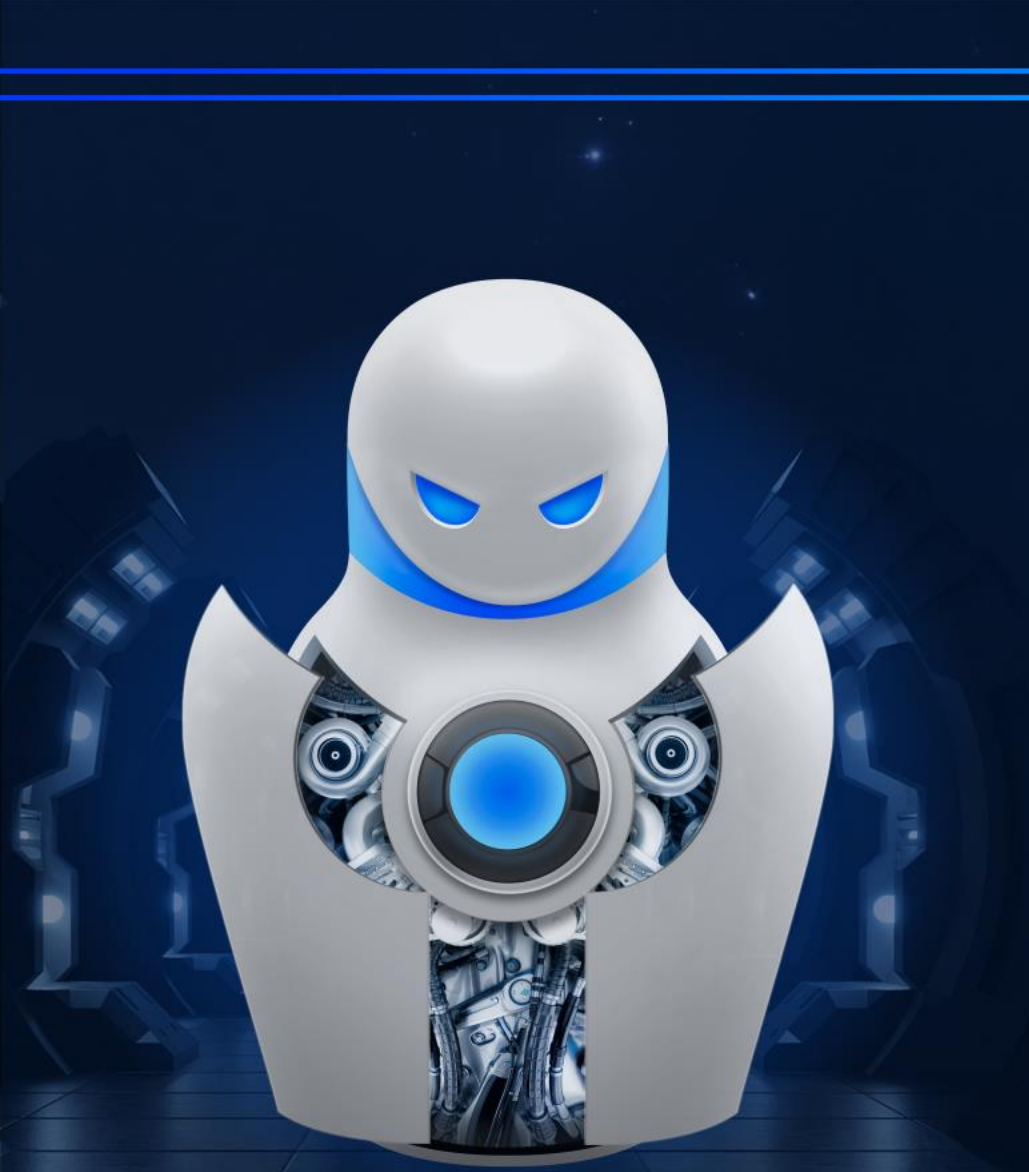


DATA-ONLY ATTACKS AGAINST UEFI BIOS

Alexander Ermolov

#whoami

- Intel Management Engine
  - Intel AMT. Stealth Breakthrough
- Intel Boot Guard
  - Safeguarding rootkits: Intel Boot Guard
  - Bypassing Intel Boot Guard
- UEFI BIOS
  - UEFI BIOS holes. So Much Magic, Don’t Come Inside
  - NUClear explotion
  - Microcode downgrade
  - Untrusted Roots: Exploiting vulnerabilities in Intel ACMs

#agenda

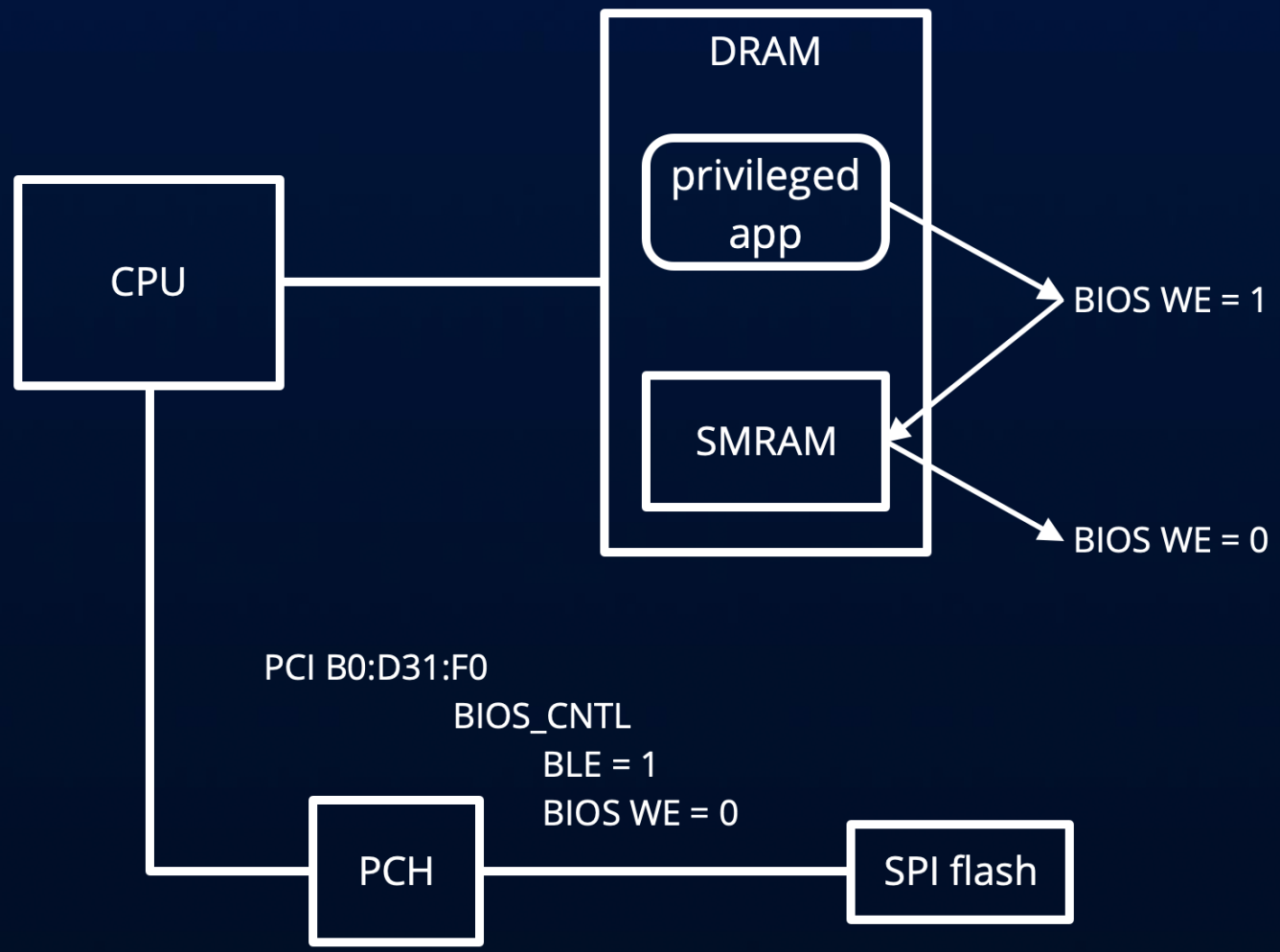
- UEFI BIOS Security
- SMM Mitigations
- Is that enough?
- Vulns
- NVRAM stories

UEFI BIOS SECURITY

UEFI BIOS common security

BIOS Lock Enable (BLE)

If BLE = 1, any attempt to set BIOS WE = 1 (required before starting flash operation cycles in SPIBAR) will trigger hardware SMI handler to set BIOS WE = 0.



OpenSecurityTraining by Xeno Kovah & Corey Kallenberg, LegbaCore LLC

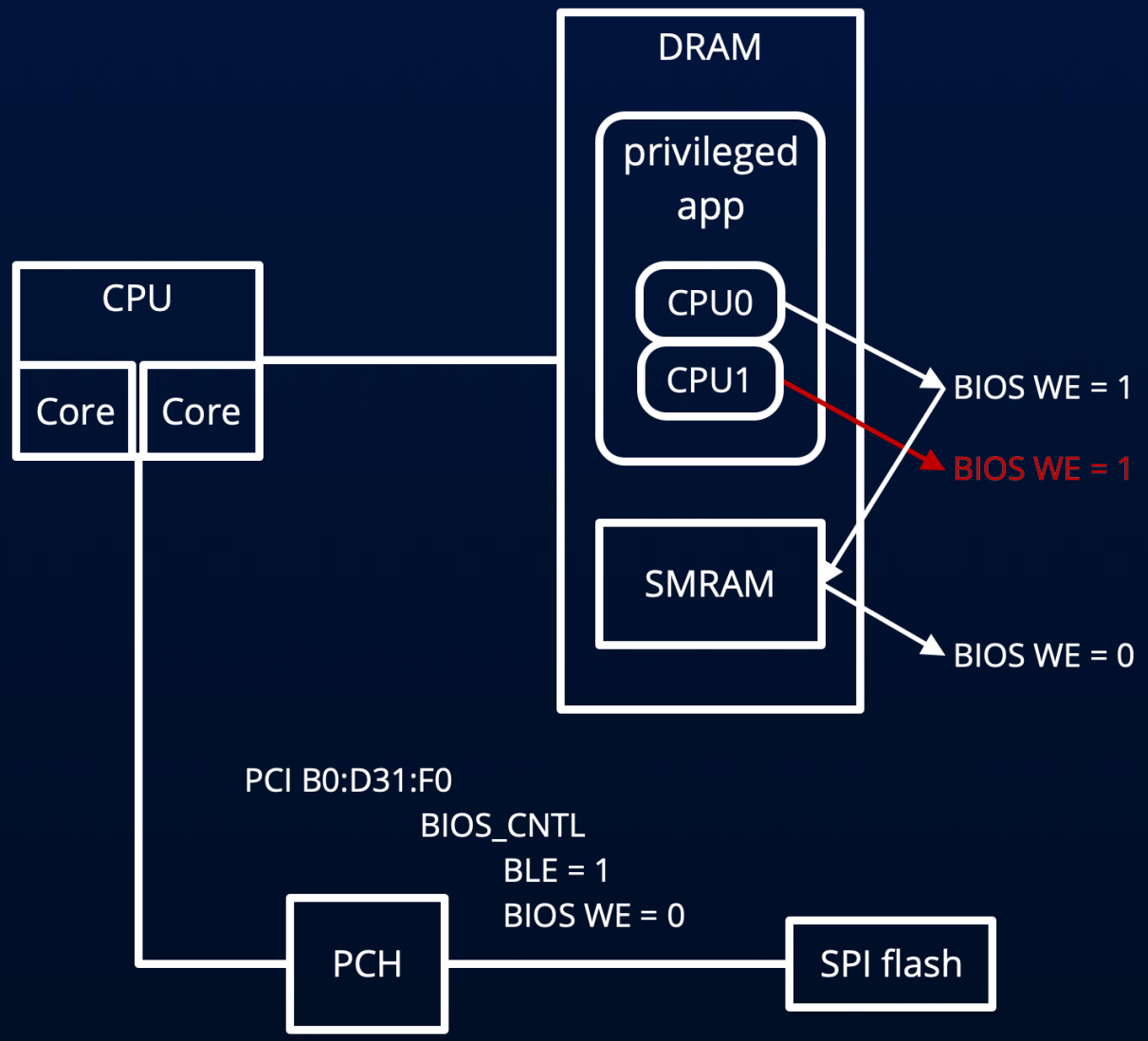
BTW check this out! OpenSecurityTraining2 by Xeno Kovah

UEFI BIOS common security

BIOS Lock Enable (BLE)

On multicore systems this protection is vulnerable to a race condition attack (aka Speed Racer).

While one CPU Core is in SMM and the other still not, there is a small time window to write to BIOS region before it will be switched to SMM.

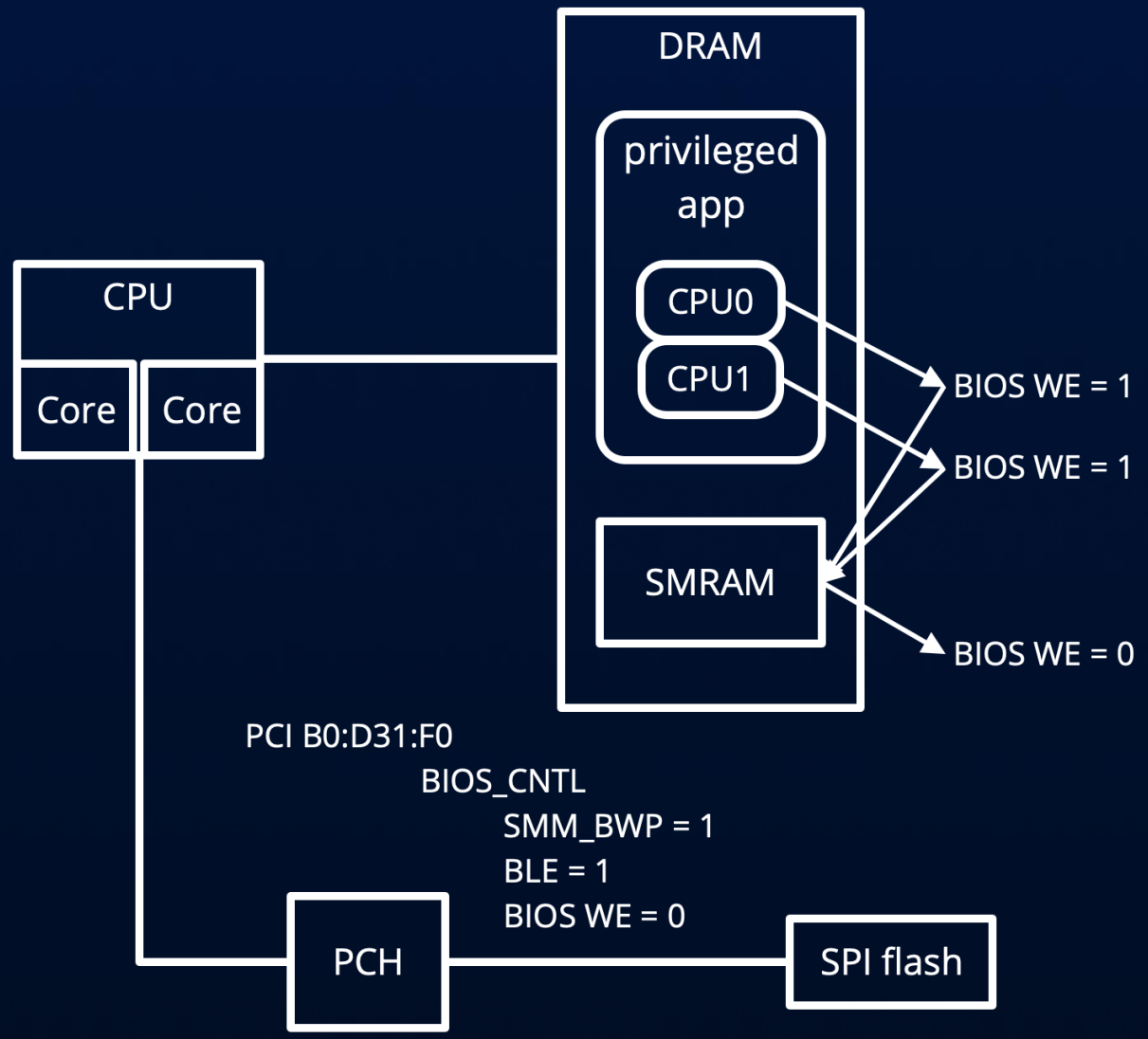


Speed Racer: Exploiting an Intel Flash Protection Race Condition by Corey Kallenberg & Rafal Wojtczuk

UEFI BIOS common security

SMM BIOS Write Protect (SMM_BWP)

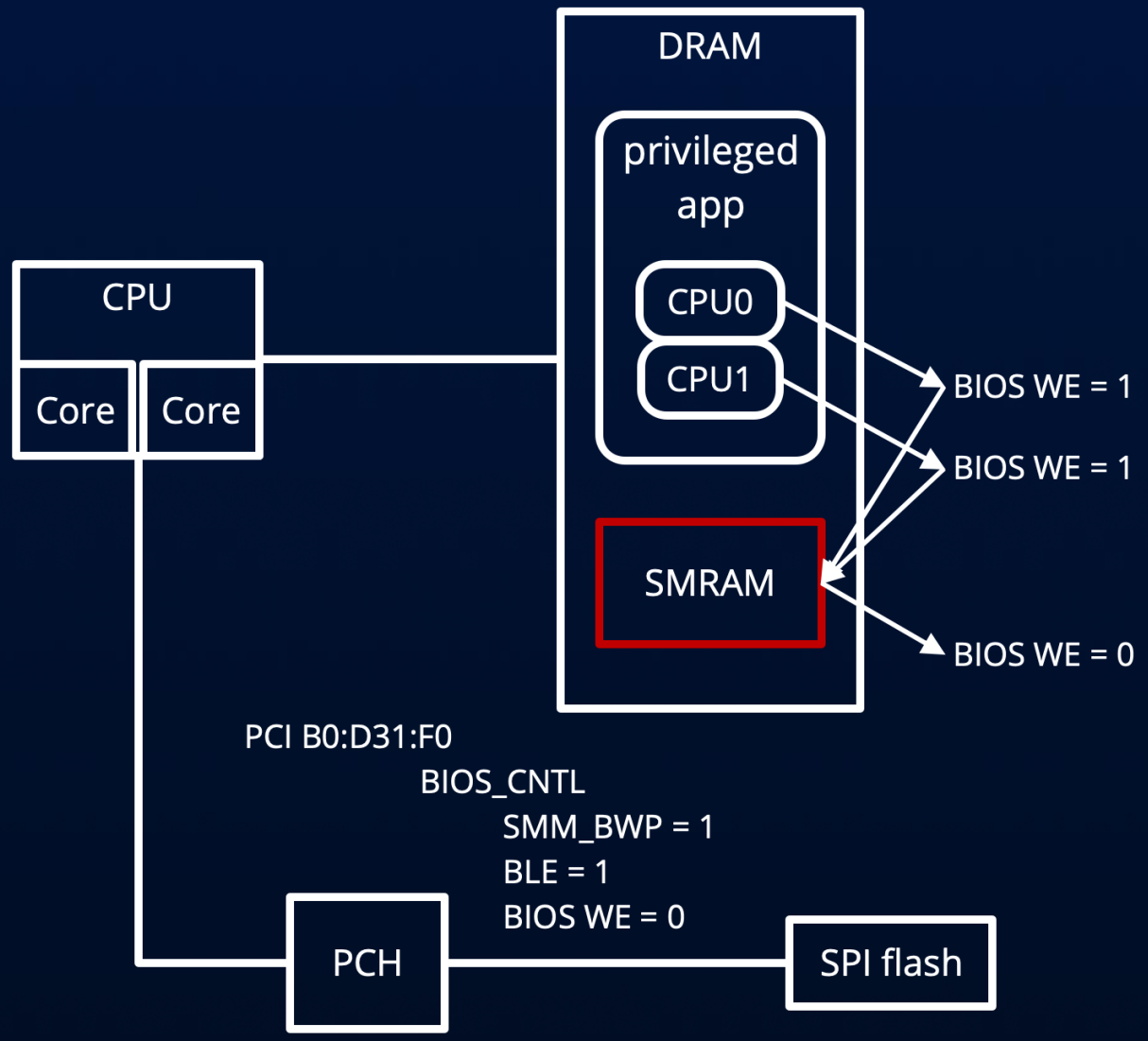
If SMM_BWP = 1, BIOS is not writable unless all processors are in SMM.



UEFI BIOS common security

BLE & SMM_BWP

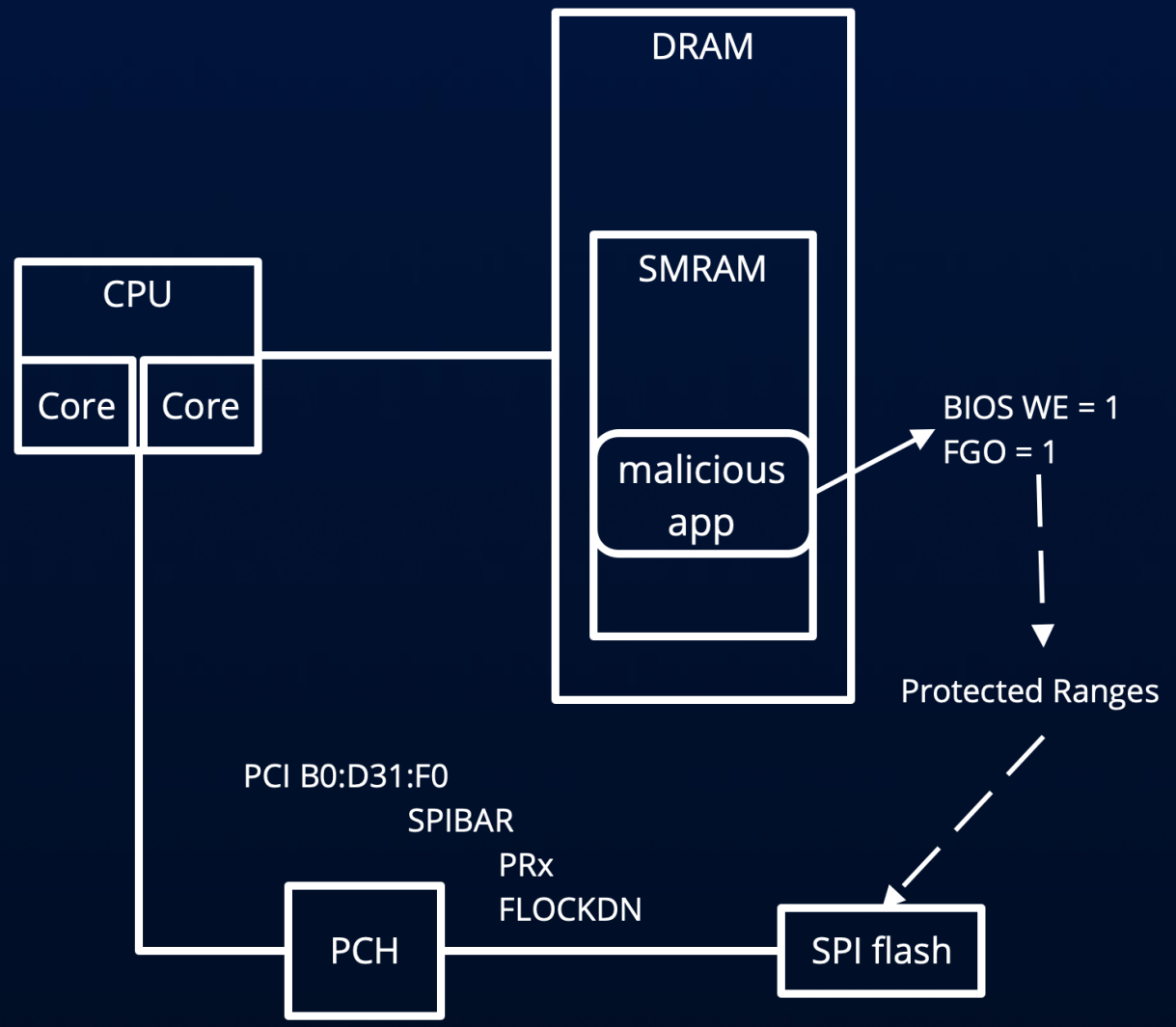
SMM-based protections makes SMM code as a main target, gaining arbitrary code execution will bypass the protections.



UEFI BIOS common security

Protected Ranges

PRx registers allows to mark certain BIOS regions as read-only. This configuration is locked by Flash Configuration Lock Down (FLOCKDN) bit.

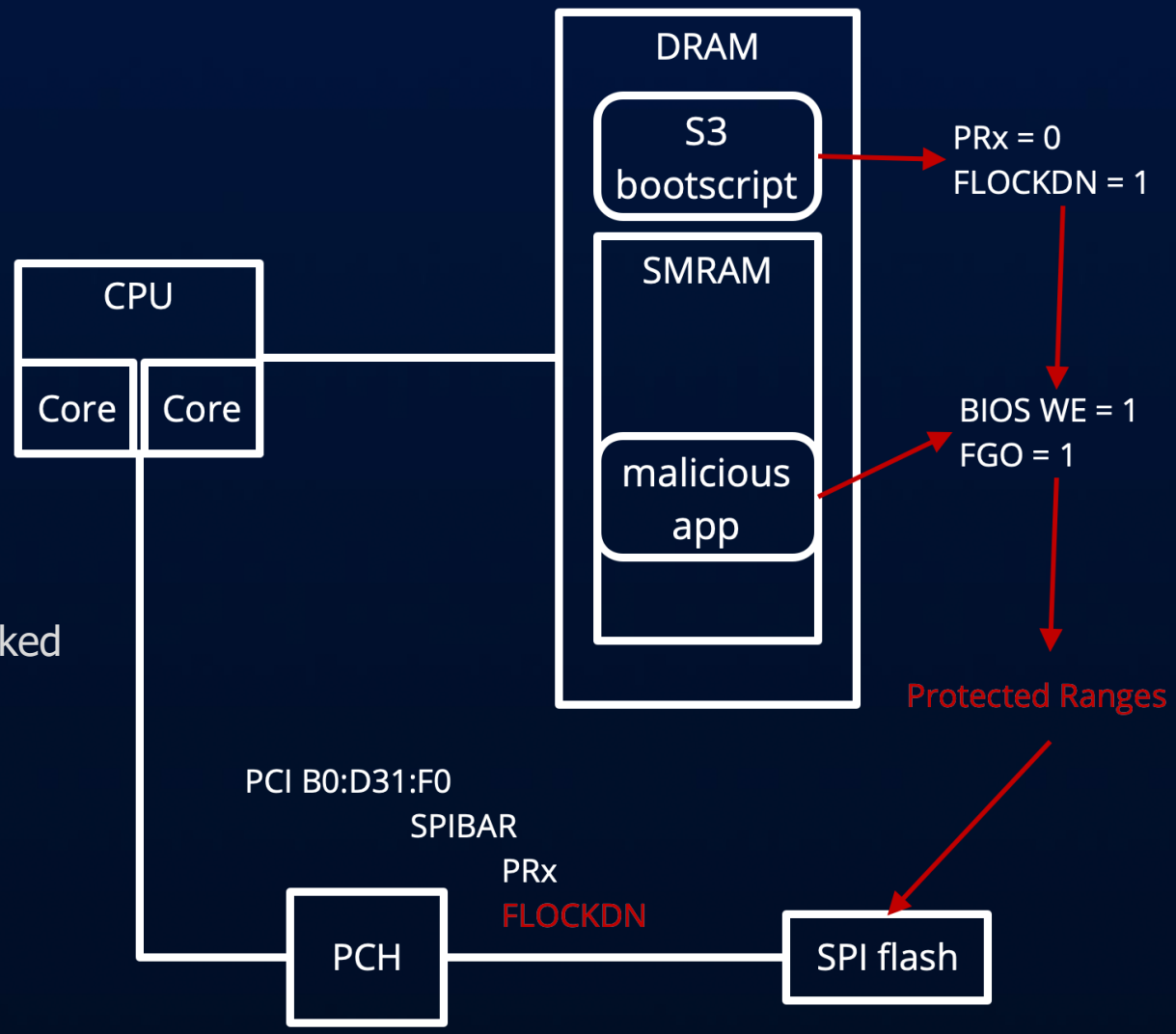


UEFI BIOS common security

Protected Ranges

Potential bypasses:

- Misconfigurations
  - FLOCKDN not set
  - Not all code-containing BIOS regions are marked
- S3 bootscript attacks
- S3 boot path attacks (SMM)
- BIOS update procedure



Adventures From UEFI Land: the Hunt For the S3 Boot Script by By Assaf Carlsbad & Itai Liba

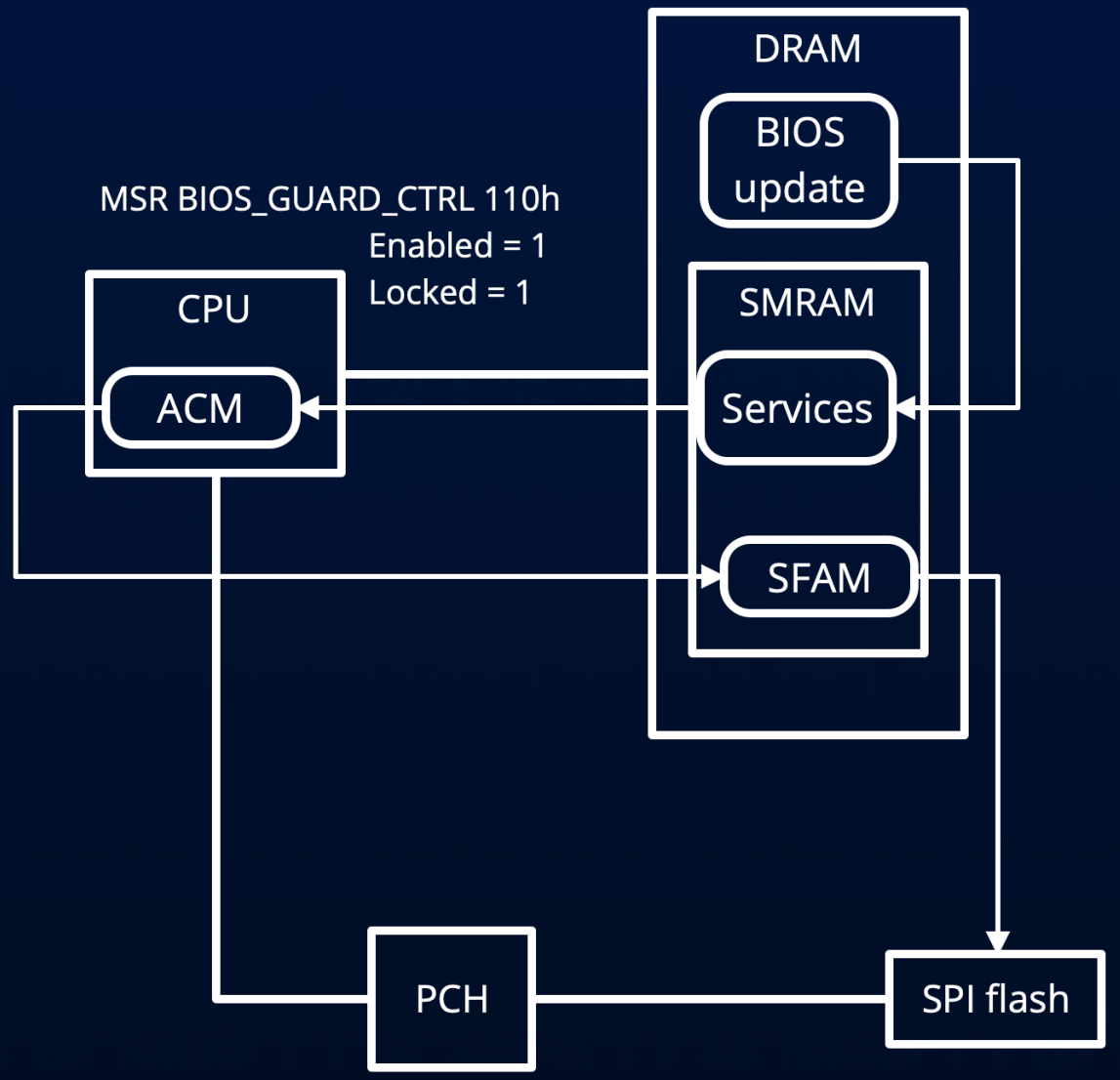
UEFI BIOS hardened security

Intel BIOS Guard

Aka Platform Flash Armoring Technology (PFAT), small trusted component granted for write operations onto SPI.

It is called Intel BIOS Guard Authenticated Code Module (ACM).

Signed Flash Address Map (SFAM) structure describes what regions are marked as Signed – means, a valid certificate is required to update them.



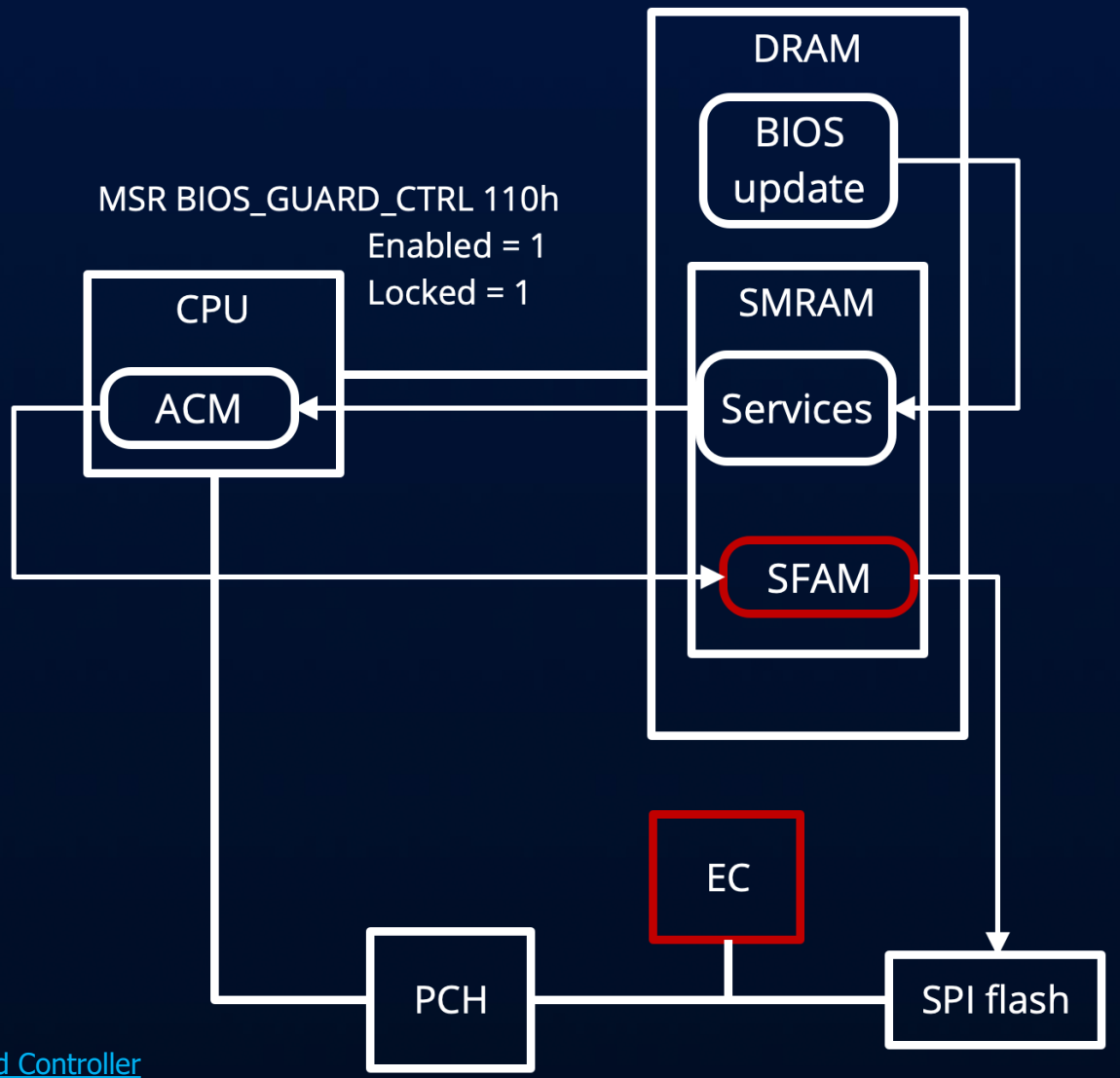
Untrusted Roots: Exploiting Vulnerabilities in Intel ACMs by Alexander Ermolov & Dmitriy Frolov

UEFI BIOS hardened security

Intel BIOS Guard

Potential bypasses:

- SFAM misconfigurations
- ACPI Embedded Controller (EC) with access to SPI
- Downgrade ACM to vulnerable version (1day)

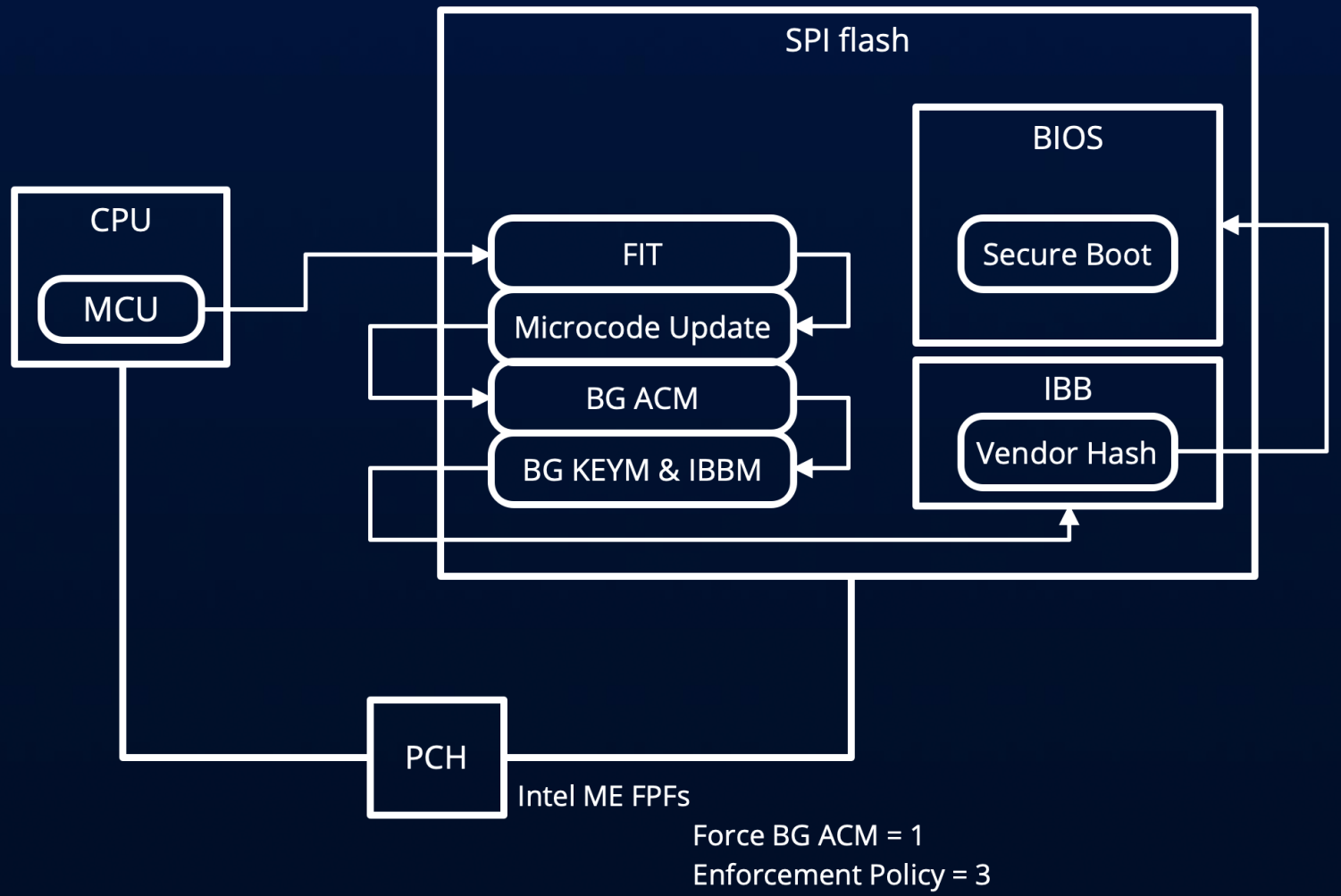


Breaking Through Another Side. Bypassing Firmware Security Boundaries from Embedded Controller by Alex Matrosov & Alexandre Gazet

UEFI BIOS hardened security

Intel Boot Guard

Hardware-supported BIOS verification mechanism.



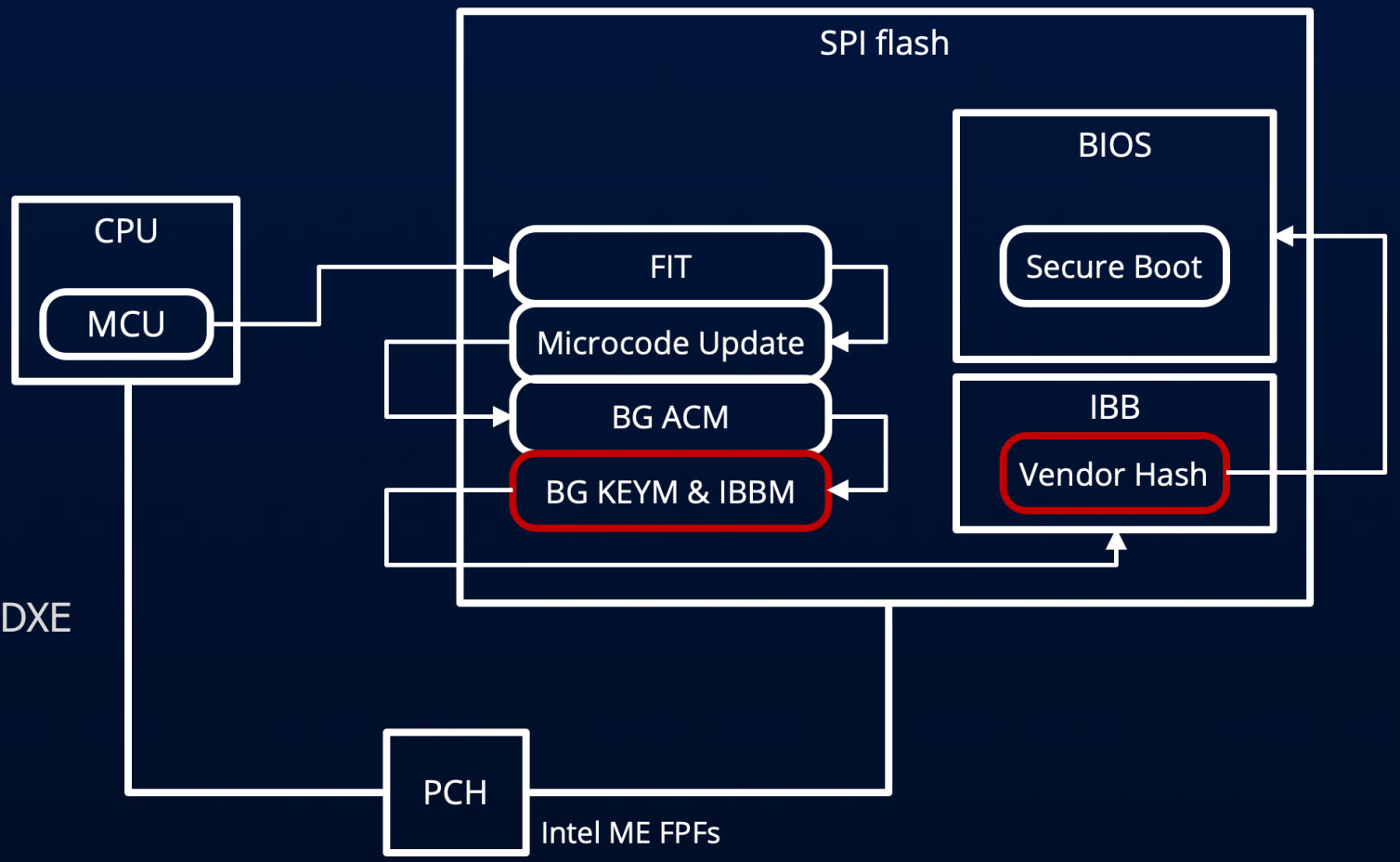
[Betraying the BIOS: where the guardians of the BIOS are failing by Alex Matrosov]

UEFI BIOS hardened security

Intel Boot Guard

Potential bypasses:

- Misconfigurations
  - Fuses are not set
  - IBB hash is not covering all SEC/PEI modules
  - Vendor hash is not covering all DXE modules
  - Downgrade MCU to vulnerable version



Force BG ACM = 1
Enforcement Policy = 3

Betraying the BIOS: where the guardians of the BIOS are failing by Alex Matrosov

Bypassing Intel Boot Guard by Alexander Ermolov

TOCTOU Attacks Against BootGuard by Peter Bosch & Trammell Hudson

[BRLY-2021-002] LENOVO SYSTEM FIRMWARE HAS MISSING COVERAGE WITH BOOT GUARD PROTECTED RANGES (IBB) FOR UEFI MODULES

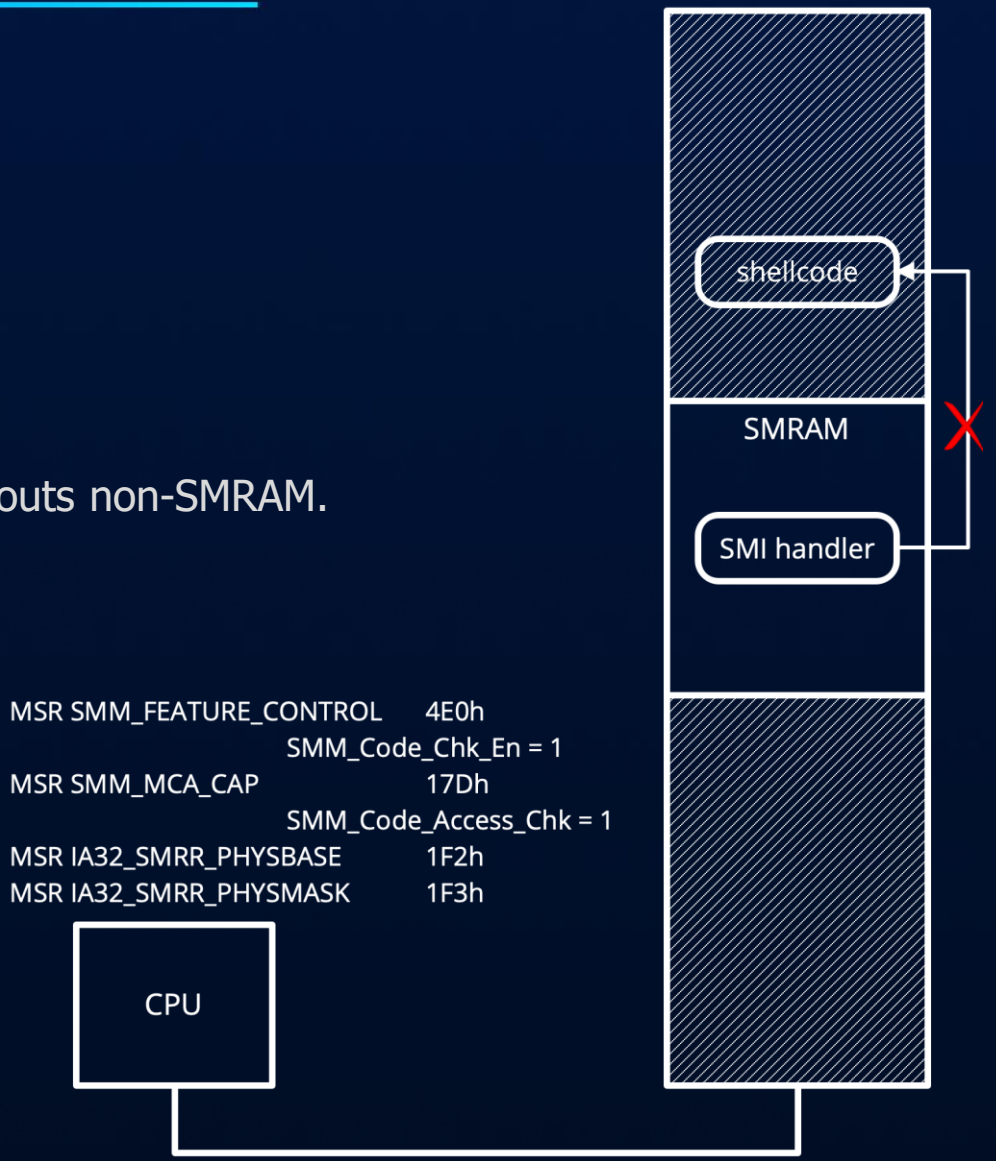
SMM MITIGATIONS

UEFI BIOS mitigated security

SMM Nx-bit & Sanity Checks

The most known and widely-used mitigation, protects from SMM call-outs non-SMRAM.

- ValidateMemoryBuffer()
- ValidateMmioBuffer()
- ValidateSmramBuffer()



[UEFI Firmware – Securing SMM by Dick Wilkins]

UEFI BIOS mitigated security

IA32 (SMM) Stack/Heap Guard

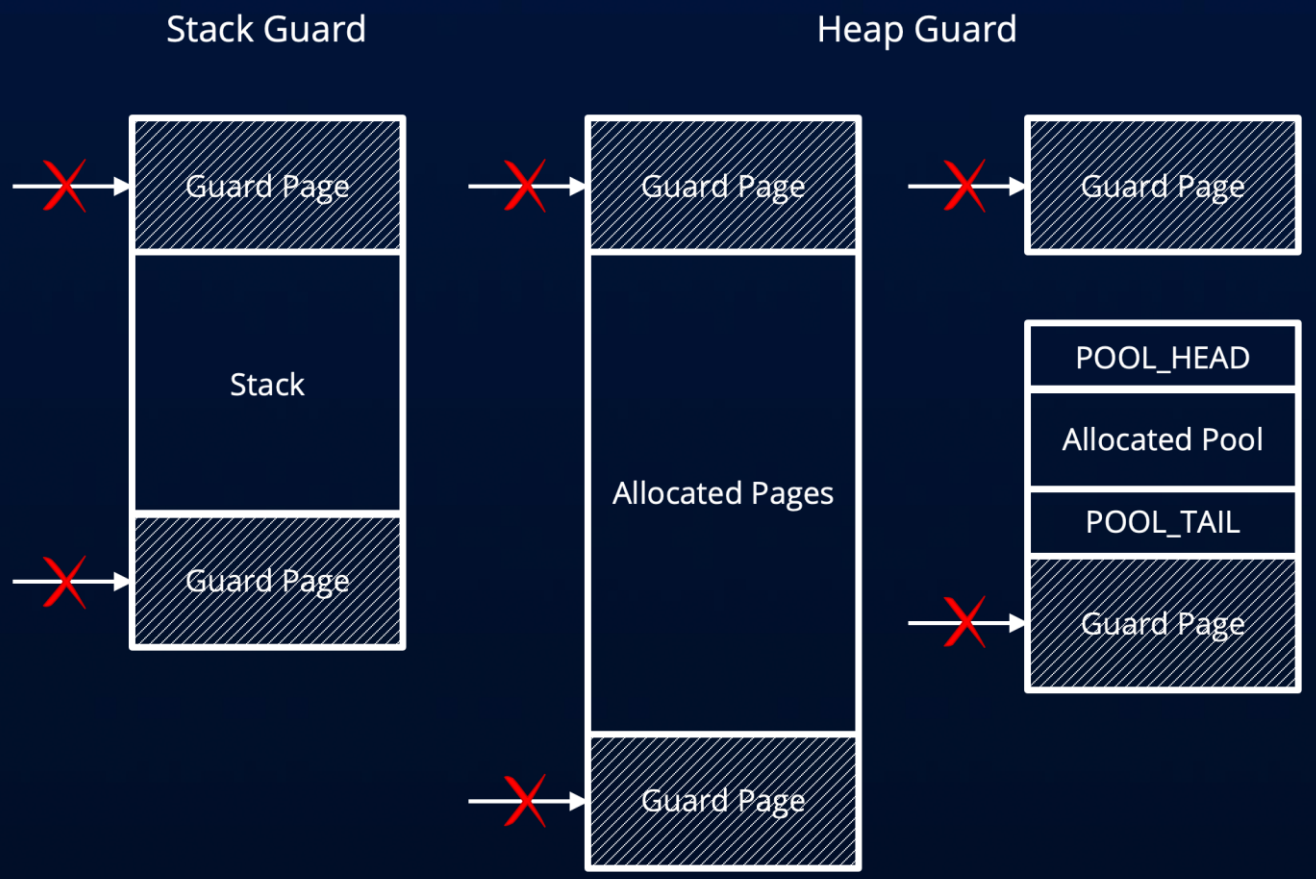
Guard Page is created before and after the allocation, accessing them will cause Page Fault.

- For Stack it is a production-feature

  Disabled by default

- For Heap it is a debug-feature

  Two different builds required for Pool to get Guard page close to POOL_HEAD and POOL_TAIL).



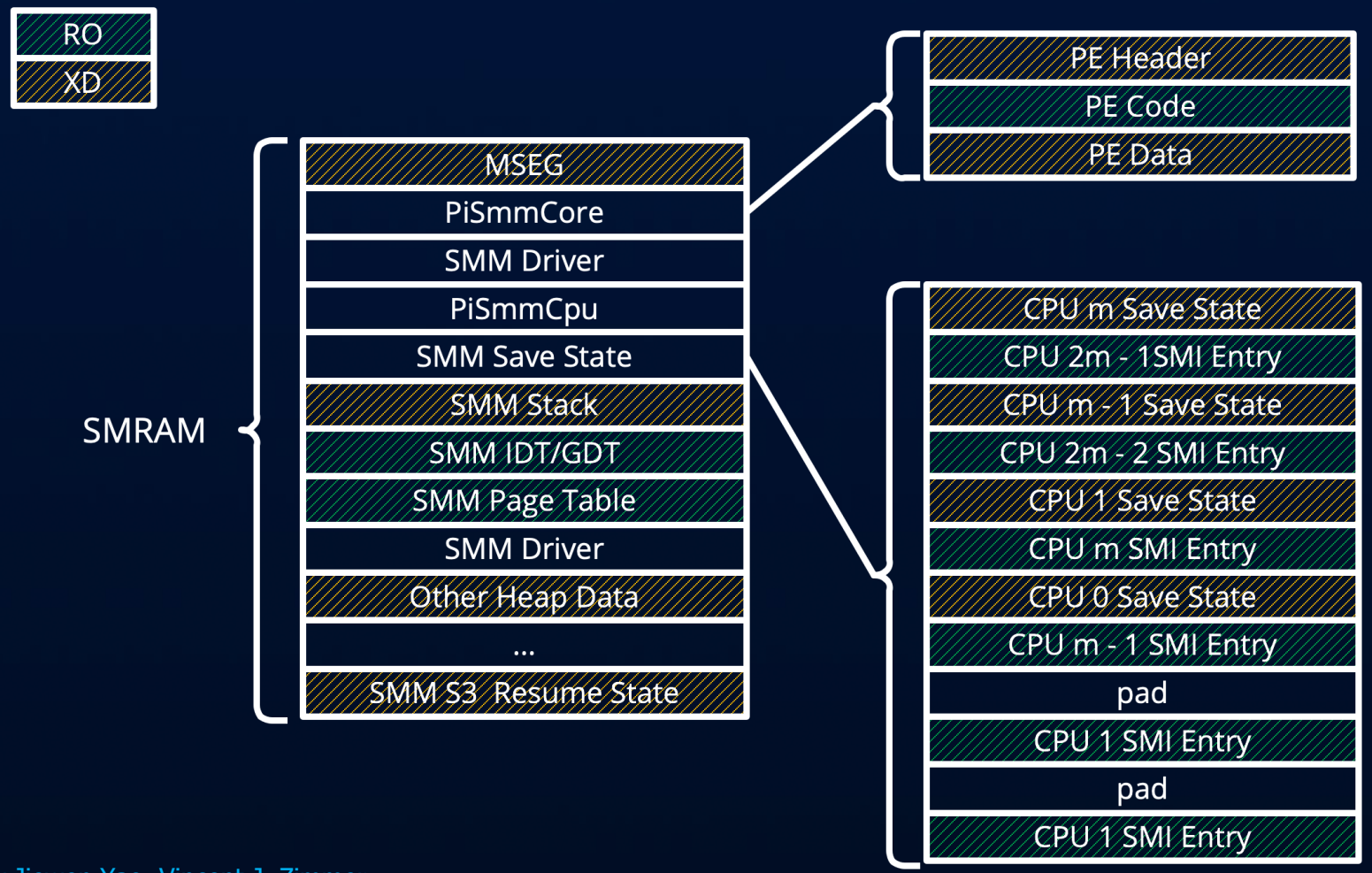
SMM Protection in EDK II by Jiewen Yao

UEFI BIOS mitigated security

SMRAM memory protection (RO/XD)

Violating Read-Only / Execute-Disable attributes will cause Segmentation Fault.

- Page Table is dynamic by default
- Disabled by default
- ROP-attackable



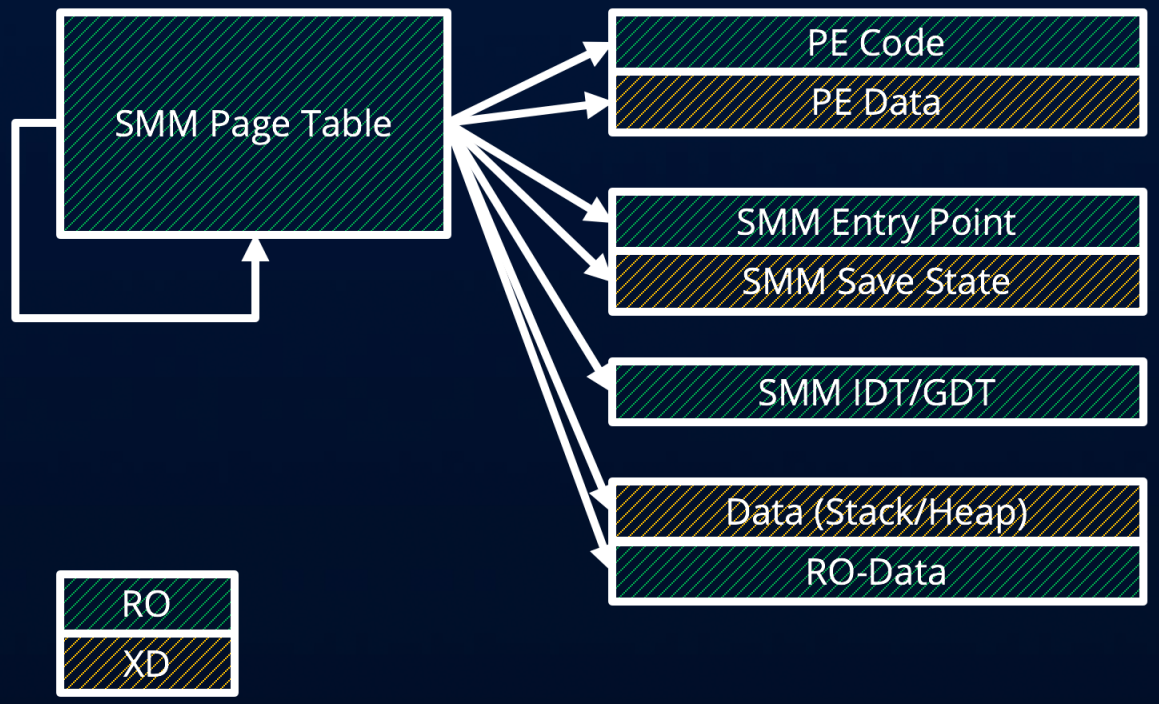
A Tour Beyond BIOS - Memory Protection in UEFI BIOS by Jiewen Yao, Vincent J. Zimmer

UEFI BIOS mitigated security

SMM static page table (RO)

SMRAM Protection + Static Page Table.

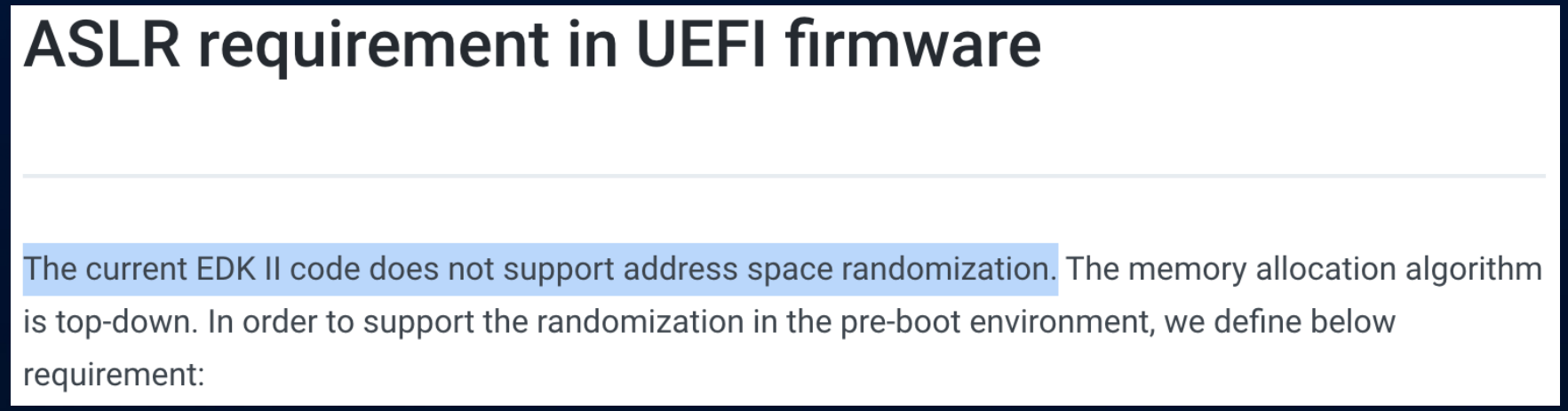
- Not compatible with Heap Guard
- Disabled by default
- ROP-attackable



A Tour Beyond BIOS - Memory Protection in UEFI BIOS by Jiewen Yao, Vincent J. Zimmer

UEFI BIOS mitigated security

- Address Space Layout Randomization (ASLR)



ASLR requirement in UEFI firmware

The current EDK II code does not support address space randomization. The memory allocation algorithm is top-down. In order to support the randomization in the pre-boot environment, we define below requirement:

A Tour Beyond BIOS - Security Enhancement to Mitigate Buffer Overflow in UEFI by Jiewen Yao, Vincent J. Zimmer, Jian Wang

- Intel CET (Control-flow Enforcement Technology) exists only here:

A Technical Look at Intel's Control-flow Enforcement Technology by Baiju V Patel

UEFI BIOS mitigated security

Windows SMM Security Mitigation Table (WSMT)

This is a static ACPI table configured by vendor and stored in system memory.

https://twitter.com/matrosov/status/1401656449965596673?s=20

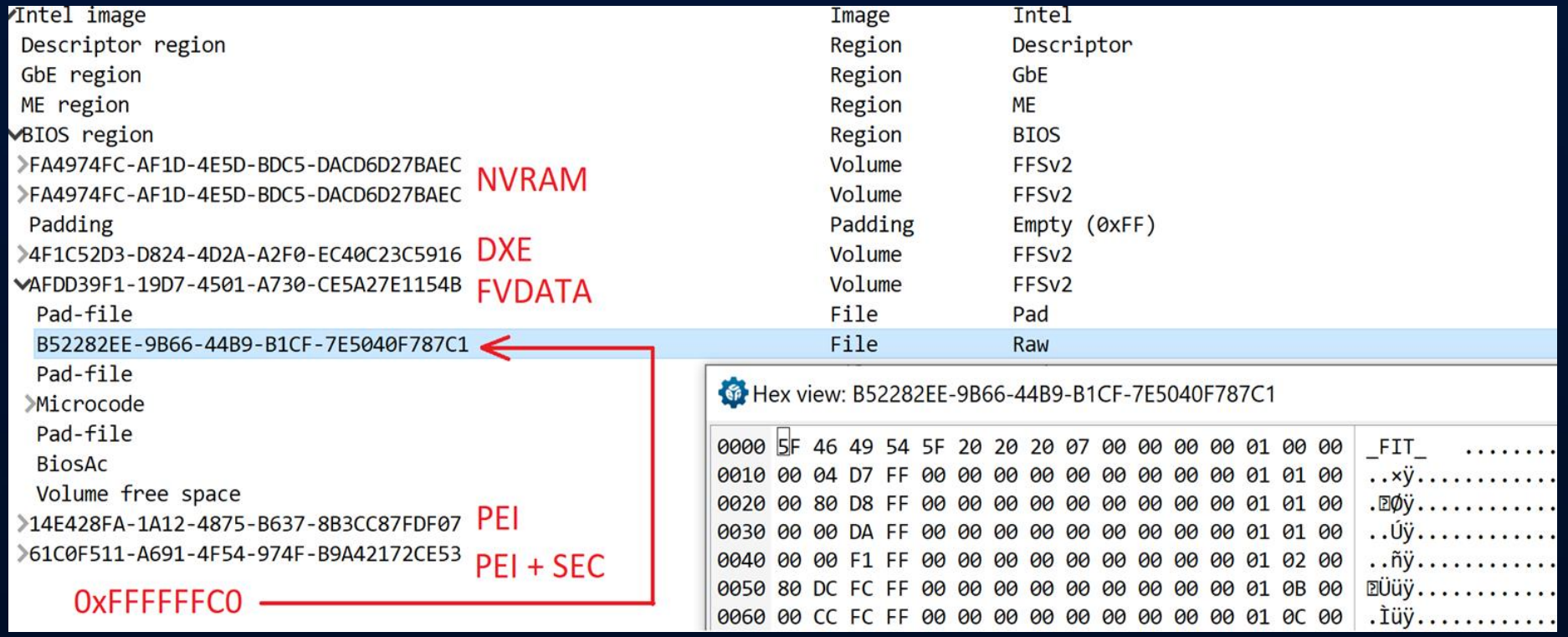
IS THAT ENOUGH?

Updated attack surface

NVRAM

Definitely not enough about secure operations with it.

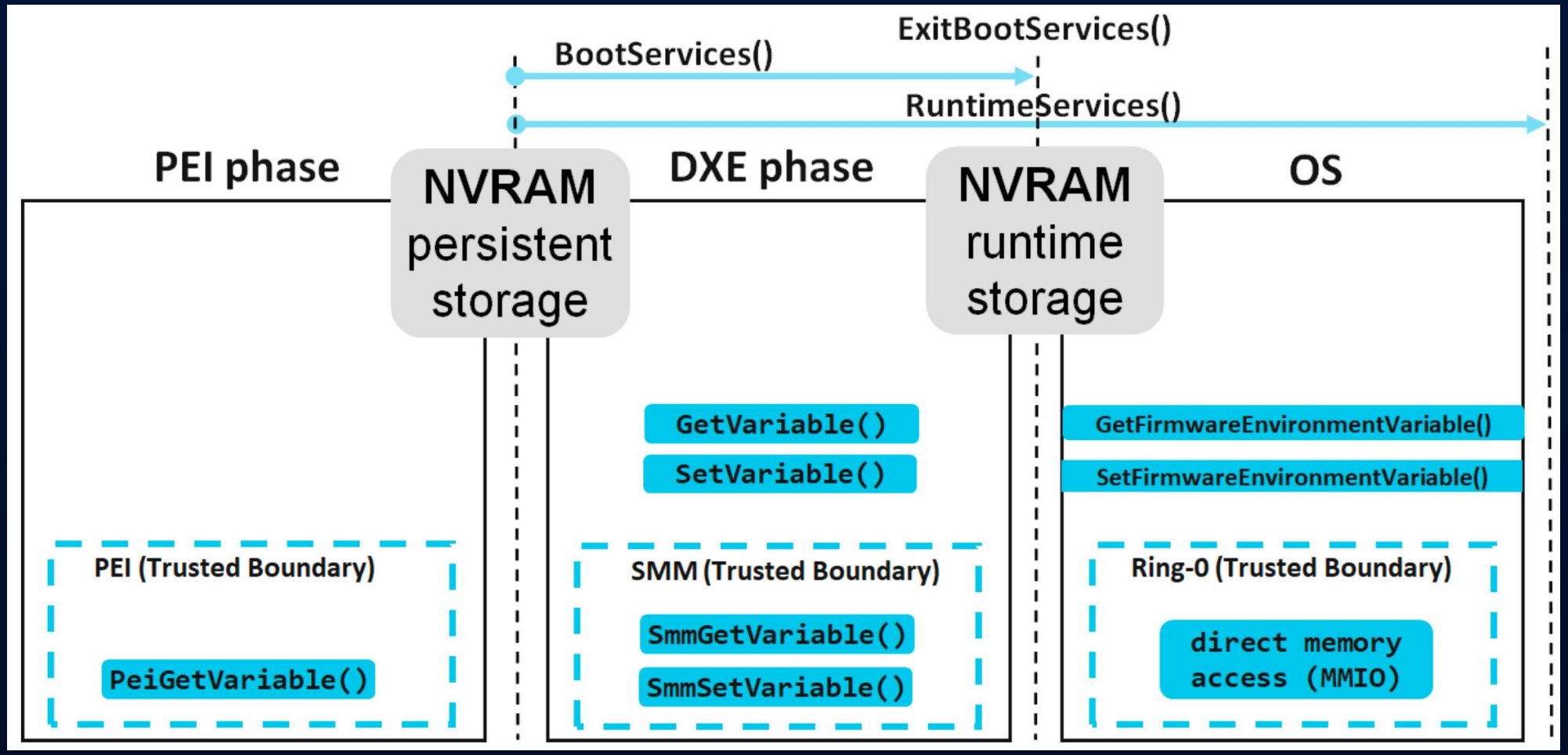
- R/W area
- Not verified
- Not measured



Updated attack surface

Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded) by Alex Tereshkin, Alex Matrosov, Adam 'pi3' Zabrocki

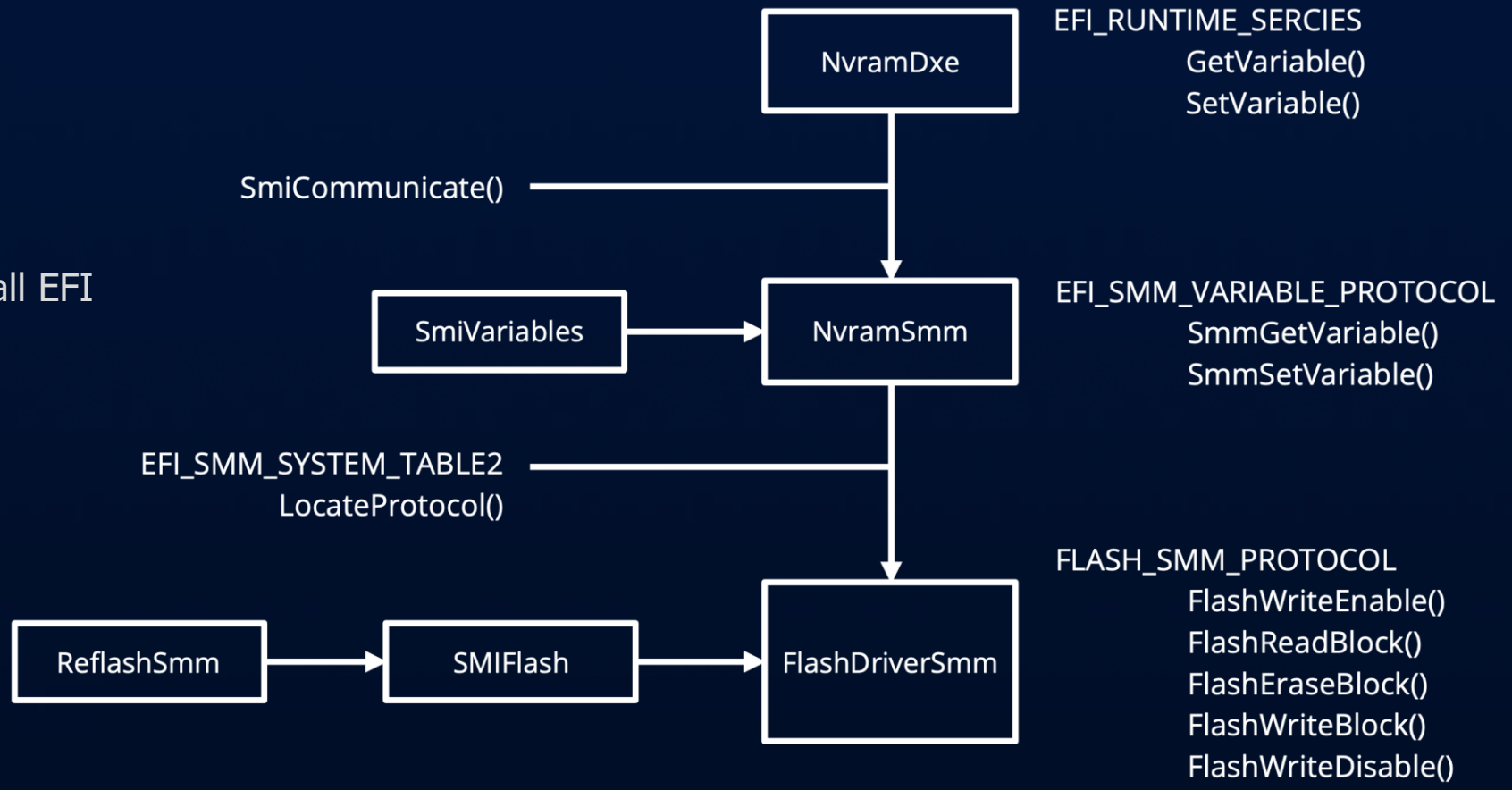
Updated Attack Surface



Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded) by Alex Tereshkin, Alex Matrosov, Adam 'pi3' Zabrocki

AMI NVRAM driven experience

This what happens when you call EFI NVAR R/W procedures.



INTEL-SA-00343

CVE-2020-0530

CVSS 7.8 High

https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00343.html

PEI module S3Resume2Pei {89E549B0-7CFE-449D-9BA3-10D8B2312D71}

```
// Retrieves 'FPDT_Variable_NV' EFI variable upon booting from S3 (Sleep) mode
DataSize = 4;
FDPT_ptr = 0;

if ( (*ReadOnlyVar)(ReadOnlyVar, L"FPDT_Variable_NV", &gFPDT_Variable_NV_Guid, 0, &DataSize, &FDPT_ptr) >= 0) {

  ptr = (_DWORD *) *FDPT_ptr;

  // Obviously, 'FPDT_Variable_NV' variable points to some structure, where the first field should be pointing to 'TP3S' signature
  if ( *(_DWORD *) *FDPT_ptr == 'TP3S' ) {
    if ( *((_QWORD *) FDPT_ptr + 2) ) {
      if ( !*((_WORD *) ptr + 4) ) {

        ...

        // After some basic validation a calculated values are written to the structure
        ptr[3] = v11;
        ptr[6] = result;
        ptr[7] = v12;
        ptr[4] = v7;
        ptr[5] = v9;
```

PEI module SmbiosPeim {968C1D9F-80C4-43B7-8CAE-668AA56C4E71}

```
// Retrieves 'WakeUpType' EFI variable upon booting from S3 (Sleep) mode
result = (*ReadOnlyVar)(ReadOnlyVar, L"WakeUpType", &gWakeUpTypeVarGuid, 0, &DataSize, &Data);

if ( !result )
{
  if ( (unsigned int) Data <= 0xF0000 )
  {

    ...

  }
  // The value is written at address pointed by 'WakeUpType' EFI variable
  else
  {
    v4 = sub_FFFAD0F2(v3);
    LOBYTE(result) = v4[1](6);

    if ( (unsigned __int8)result >= 9u )
      LOBYTE(result) = 6;

    *Data = result;
  }
}
```

Exploitation

- Craft the ‘FPDT_Variable_NV’ or ‘WakeUpType’ EFI Variable
- Write onto NVRAM (via SMM if hardened security or RT if common security)
- Restart or Sleep/Wakeup the system

The vulnerability will be triggered upon every boot.

However we need to craft the variable (bypass all watermark checks) very carefully to gain something. Complicated, let’s look into another vulnerability.

INTEL-SA-00343

CVE-2020-0526

CVSS 7.7 High

https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00343.html

SMM module NvramSmm {447A1B58-8F3E-4658-ABAA-9E7B2280B90A}

```
// Retrieves NvramMailBox EFI variable

UINTN Data[3];
UINTN DataSize = 0x18;

result = gEfiSystemTable_0->RuntimeServices->GetVariable(L"NvramMailBox", &NVRAM_MAILBOX_VARIABLE_GUID, 0, &DataSize, Data);

if ( result >= 0 )
{
  ...

  // Processes the first and last QWORD from NvramMailBox

  ...

  // Deletes NvramMailBox variable
  gEfiSystemTable_0->RuntimeServices->SetVariable(L"NvramMailBox", &NVRAM_MAILBOX_VARIABLE_GUID, 0, 0, 0);

  result = gSmst_1->SmmAllocatePool(EfiRuntimeServicesData, 0x10000ui64, &Buffer);
  if ( result >= 0 )
  {
    ...
    result = gSmst_1->SmiHandlerRegister(SmiHandler, &NVRAM_MAILBOX_VARIABLE_GUID, &DispatchHandle);
    if ( result >= 0 )
    {
      // Calls a function pointed by the second QWORD from NvramMailBox!
      result = ((__int64 (*)(void)) Data[1])();
```

Exploitation

It is very simple to craft this variable so that it would lead to gaining code execution in SMM.

However at the moment of running NvramSmm this variable is already created in DXE module NvramDxe {1807040D-5934-41A2-A088-8E0F777F71AB}:

```
Data[0] = (__int64) &dword_8000E620;
Data[1] = (__int64) sub_80001C60;
Data[2] = (__int64) &byte_8000E924;

result = SetVariable_local(L"NvramMailBox", &NvramMailBoxVarGuid, 2, 0x18, Data);
```

But since the variable is deleted in NvramSmm (with all its attributes):

```
// Deletes NvramMailBox variable
gEfiSystemTable_0->RuntimeServices->SetVariable(L"NvramMailBox", &NVRAM_MAILBOX_VARIABLE_GUID, 0, 0, 0);
```

We can create the new one! Using just EFI_RUNTIME_SERVICES->SetVariable() with NON_VOLATILE | RUNTUME_ACCESS | AUTH_ACCESS attributes to lock it for NvramDxe.

Exploitation

- Build payload
- Write onto NVRAM (via SMM if hardened security or RT if common security)
- Craft the ‘NvramMailBox’ EFI Variable so its second QWORD will point to the payload inside NVRAM and with attributes to lock it for boot phase
- Restart the system

The payload will be executed upon every boot

Tested on Intel® NUC Kit NUC7i7BNH

Impact

- This vulnerability bypasses all known BIOS protection mechanisms since:
  - NVRAM is a R/W area by design not measured/verified
  - Lack of anti-exploit mitigations (no ASLR, no Memory Guards, no RO/XD etc.)
- Ring 3 admin no SMM required API
- Good point is:

Given different patch cycles across vendors, these vulnerabilities can stay unpatched on endpoint devices for 6-9 months (sometimes even longer).

Firmware Supply Chain is Hard(Coded) by Binarly team

- Even if BIOS stored in Apple T2 security chip (you can still exploit memory corruptions in NVRAM)

NVRAM STORIES

with efiXplorer

Thanks to Yegor Vasilenko @yeggorv for helping with this part

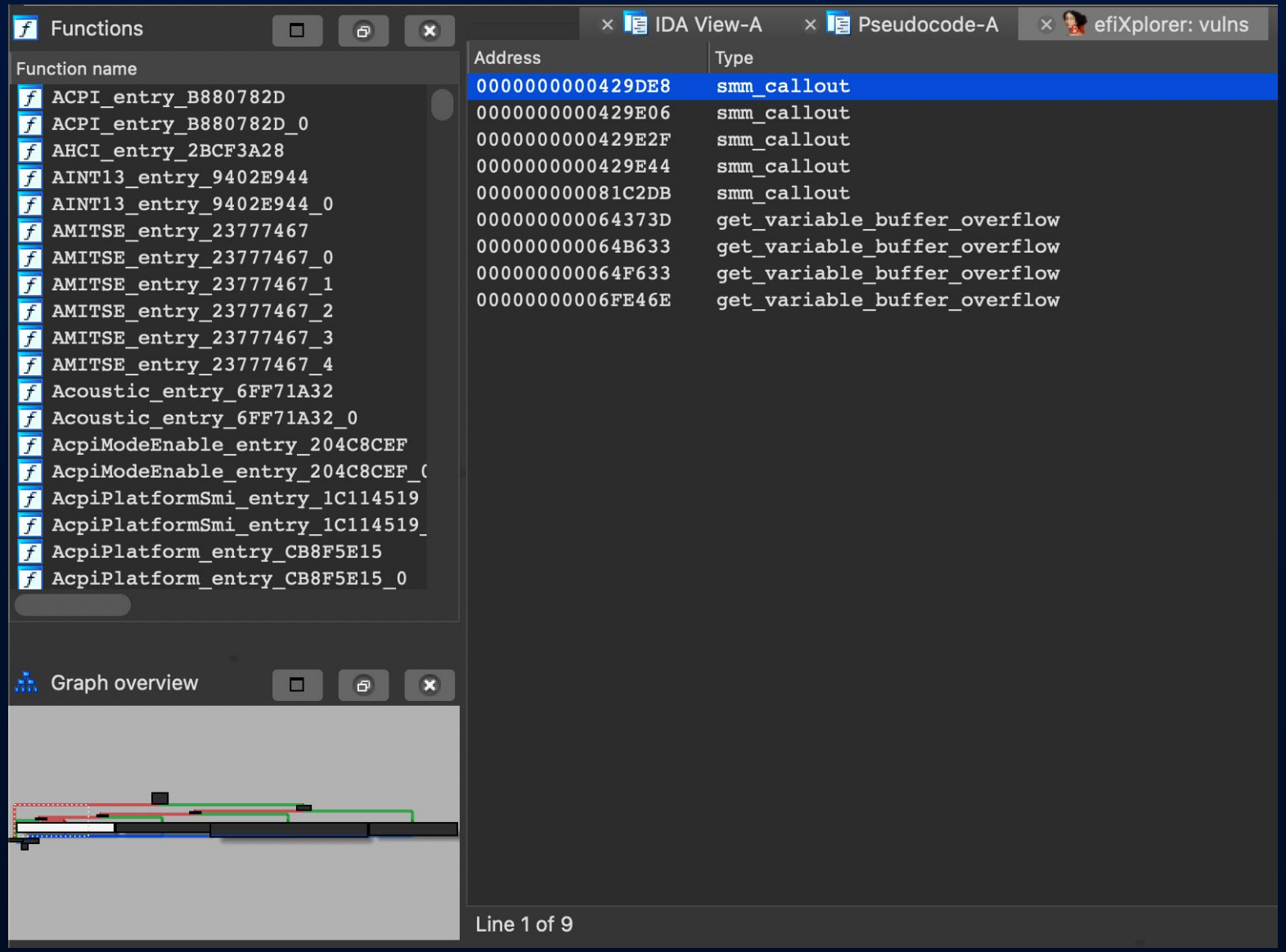
https://github.com/binarly-io/efiXplorer

Load & Scan

Testing efiXloader / efiXplorer on some old buggy firmware.

Results:

- a few SMM callouts
- a few GetVariable() buffer overflows



GetVariable() buffer overflow

GetVariable() used to retrieve the size of the variable, which further is ignored.

IDA View-A | Pseudocode-A | efiXplorer: vulns | Hex View-1 | Structures | Enums

```
__int64 __fastcall sub_64F590(unsigned __int16 a1, __int64 a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  DataSize = 0i64;
  v9 = 1280i64;
  gBS_6526B0->LocateProtocol(&EFI_HII_STRING_PROTOCOL_GUID_651C78, 0i64, &Interface);
  v4 = gRT_6526B8->GetVariable(L"PlatformLang", &EFI_SIMPLE_BOOT_FLAG_VARIABLE_GUID_651BB0, 0i64, &DataSize, Data);
  if ( v4 == EFI_BUFFER_TOO_SMALL )
    v4 = gRT_6526B8->GetVariable(L"PlatformLang", &EFI_SIMPLE_BOOT_FLAG_VARIABLE_GUID_651BB0, 0i64, &DataSize, Data);
  if ( v4 >= 0 )
    return (Interface->GetString)(Interface, Data, qword_652728, a1, a2, &v9, 0i64);
  else
    return EFI_NOT_FOUND;
}
```

SMM call-outs

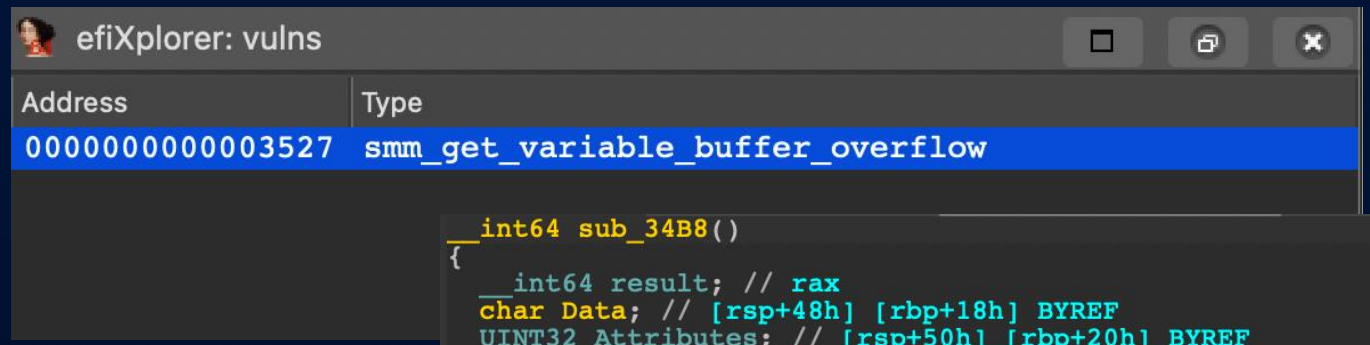
Usage of EFI_RUNTIME_SERVICES in SMI Handler.

IDA View-A | Pseudocode-A | efiXplorer: vulns | Hex View-1 | Structures

```
EFI_STATUS sub_429DAC()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v0 = qword_42AE08;
  if ( *qword_42AE08 != 'RAVN' )
    return EFI_DEVICE_ERROR;
  switch ( *(qword_42AE08 + 40) )
  {
    case 1:
      result = gRT_42AE28->GetVariable(
                 (qword_42AE08 + 72),
                 (qword_42AE08 + 8),
                 (qword_42AE08 + 4),
                 (qword_42AE08 + 24),
                 (qword_42AE08 + 72));
      break;
    case 2:
      result = gRT_42AE28->SetVariable(
                 (*(qword_42AE08 + 24) + qword_42AE08 + 72),
                 (qword_42AE08 + 8),
                 *(qword_42AE08 + 4),
                 *(qword_42AE08 + 24),
                 (qword_42AE08 + 72));
      break;
    case 3:
      result = gRT_42AE28->GetNextVariableName((qword_42AE08 + 24), (qword_42AE08 + 72), (qword_42AE08 + 8));
      break;
    case 4:
      result = gRT_42AE28->QueryVariableInfo(
                 *(qword_42AE08 + 4),
                 (qword_42AE08 + 48),
                 (qword_42AE08 + 56),
                 (qword_42AE08 + 64));
      break;
    default:
      return EFI_DEVICE_ERROR;
  }
  *(v0 + 32) = result;
  *(v0 + 40) = 0;
  *v0 = 0xADBEA9B1;
  return result;
}
```

SmmGetVariable() buffer overflow




```
__int64 sub_34B8()
{
  __int64 result; // rax
  char Data; // [rsp+48h] [rbp+18h] BYREF
  UINT32 Attributes; // [rsp+50h] [rbp+20h] BYREF
  UINTN DataSize; // [rsp+58h] [rbp+28h] BYREF

  DataSize = 0i64;
  result = gSmmVar_5B40->SmmGetVariable(gVariableName1, &gVendorGuid, &Attributes, &DataSize, 0i64);
  if ( result == EFI_BUFFER_TOO_SMALL )
  {
    result = gSmmVar_5B40->SmmGetVariable(gVariableName1, &gVendorGuid, &Attributes, &DataSize, &Data);
    if ( result >= 0 )
    {
      DataSize = 0i64;
      (gSmmVar_5B40->SmmSetVariable)(gVariableName1, &gVendorGuid, Attributes, 0i64, 0i64);
      DataSize = 1i64;
      return (gSmmVar_5B40->SmmSetVariable)(gVariableName2, &gVendorGuid, Attributes);
    }
  }
  else if ( result == 0x800000000000000Eui64 )
  {
    return 0i64;
  }
  return result;
}
```

Outcomes

- Vector to bypass all known protections
- UEFI BIOS attack surface keeps extending:
  - More NVRAM format & EFI variable parsing code
  - More UEFI API for the runtime (Redfish etc.)
- Even a signed firmware can’t be trusted (c) Binarly

Mitigations

- Enforce mitigations
- Type safe languages

THANK YOU