
# Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded)

Alex Tereshkin, Alex Matrosov, Adam ‘pi3’ Zabrocki

# Who are we?

NVIDIA

Alex Tereshkin
@AlexTereshkin

- BlackHat speaker and trainer
- Reverse engineer
- UEFI security researcher
- More...

Alex Matrosov
@matrosov

- Security REsearcher since 1997
- Conference speaker and trainer
- Breaking all shades of firmware
- codeXplorer and efiXplorer plugins
- Author "Bootkits and Rootkits" book
- More...

Adam ‘pi3’ Zabrocki
@Adam_pi3

- Phrack author
- Bughunter (Hyper-V, Linux kernel, OpenSSH, Apache, gcc SSP, more…)
- Creator and a developer of Linux Kernel Runtime Guard (LKRG)
- More…

NVIDIA Product Security Team

# Outline

- Research Motivation

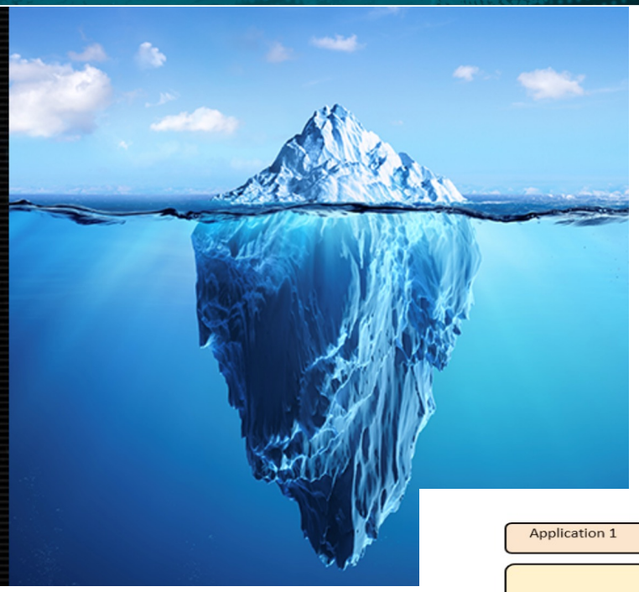
- UEFI Ecosystem Firmware Supply Chain is Hard(coded)
- Attacking (pre)EFI Ecosystem

- Safeguarding UEFI Ecosystem
- Summary

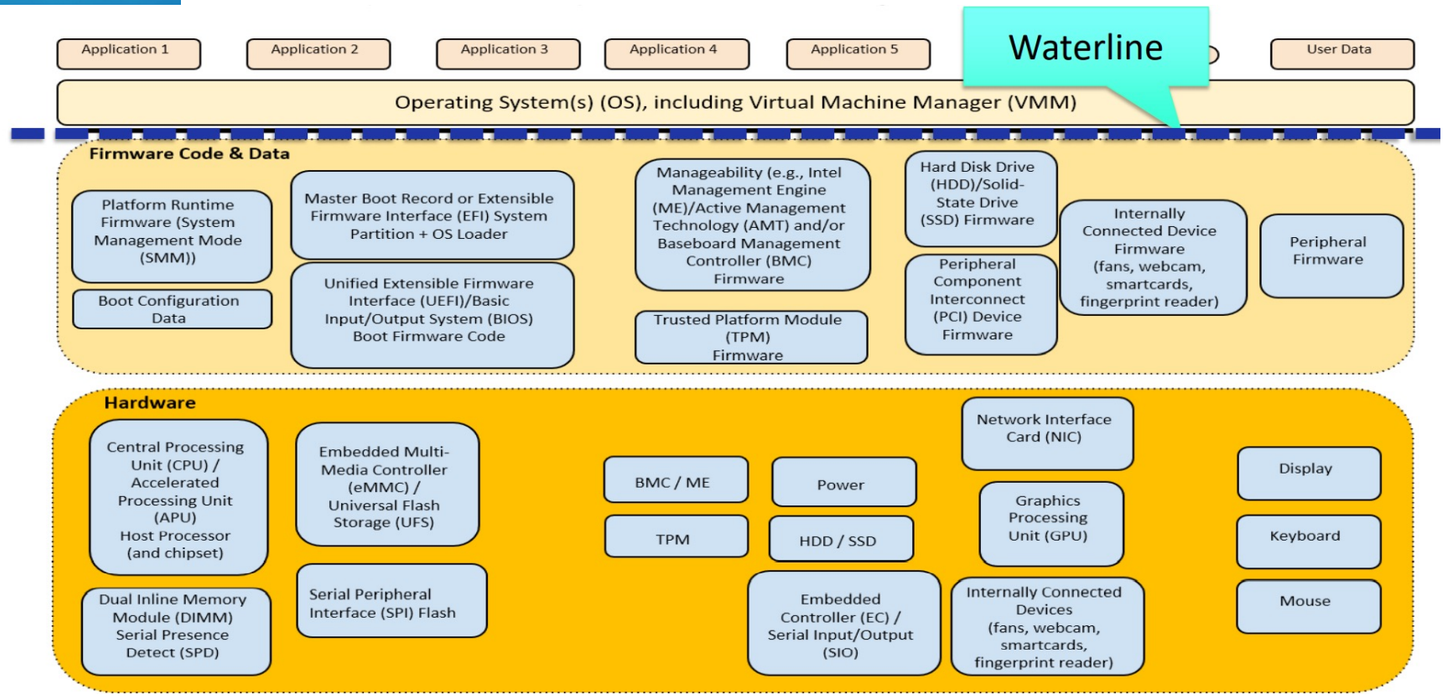
# Motivation for This Research

Security Industry Visibility Point

Modern Persistence Techniques

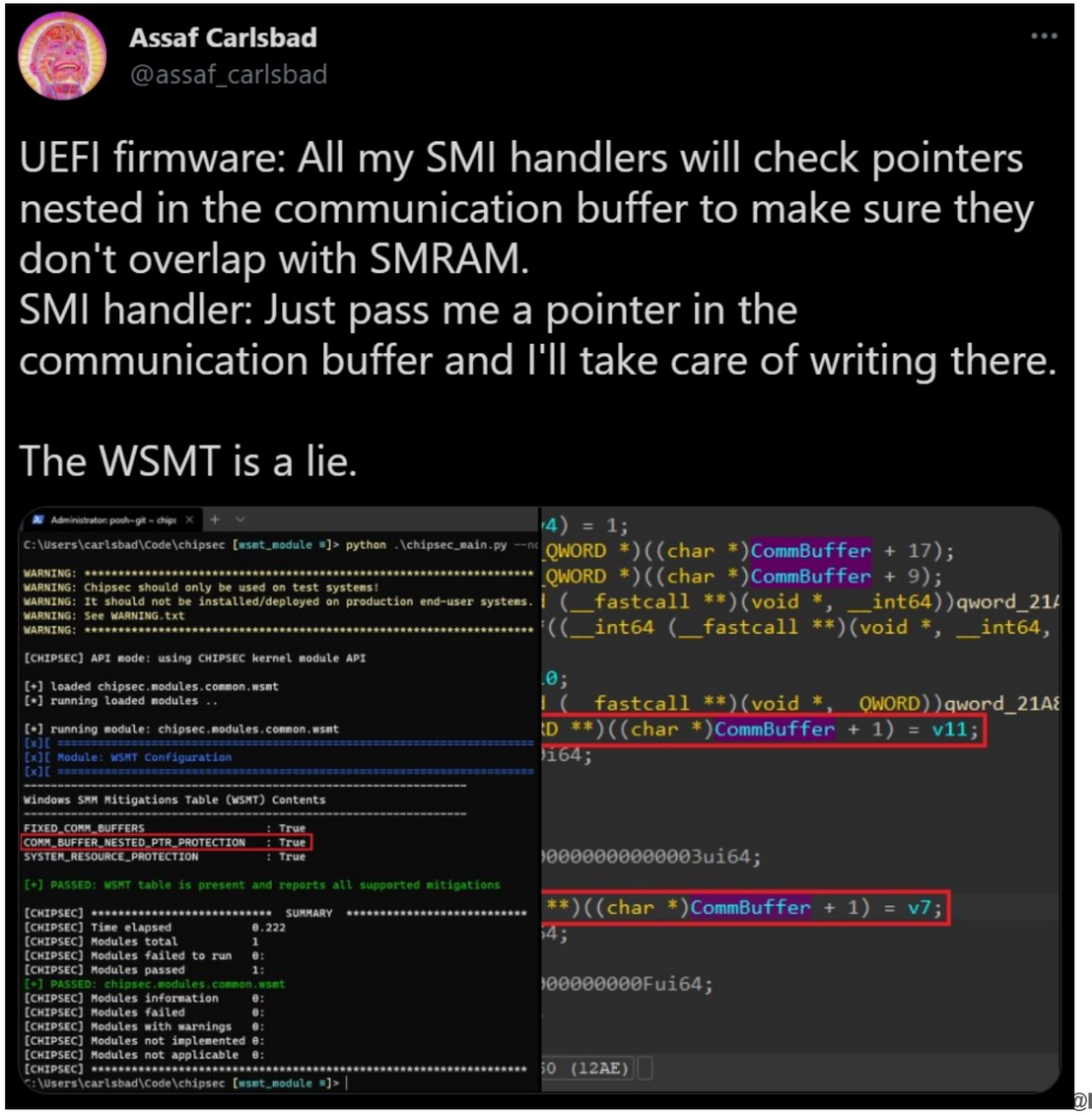


[The Evolution of Advanced Threats: REsearchers Arms Race]



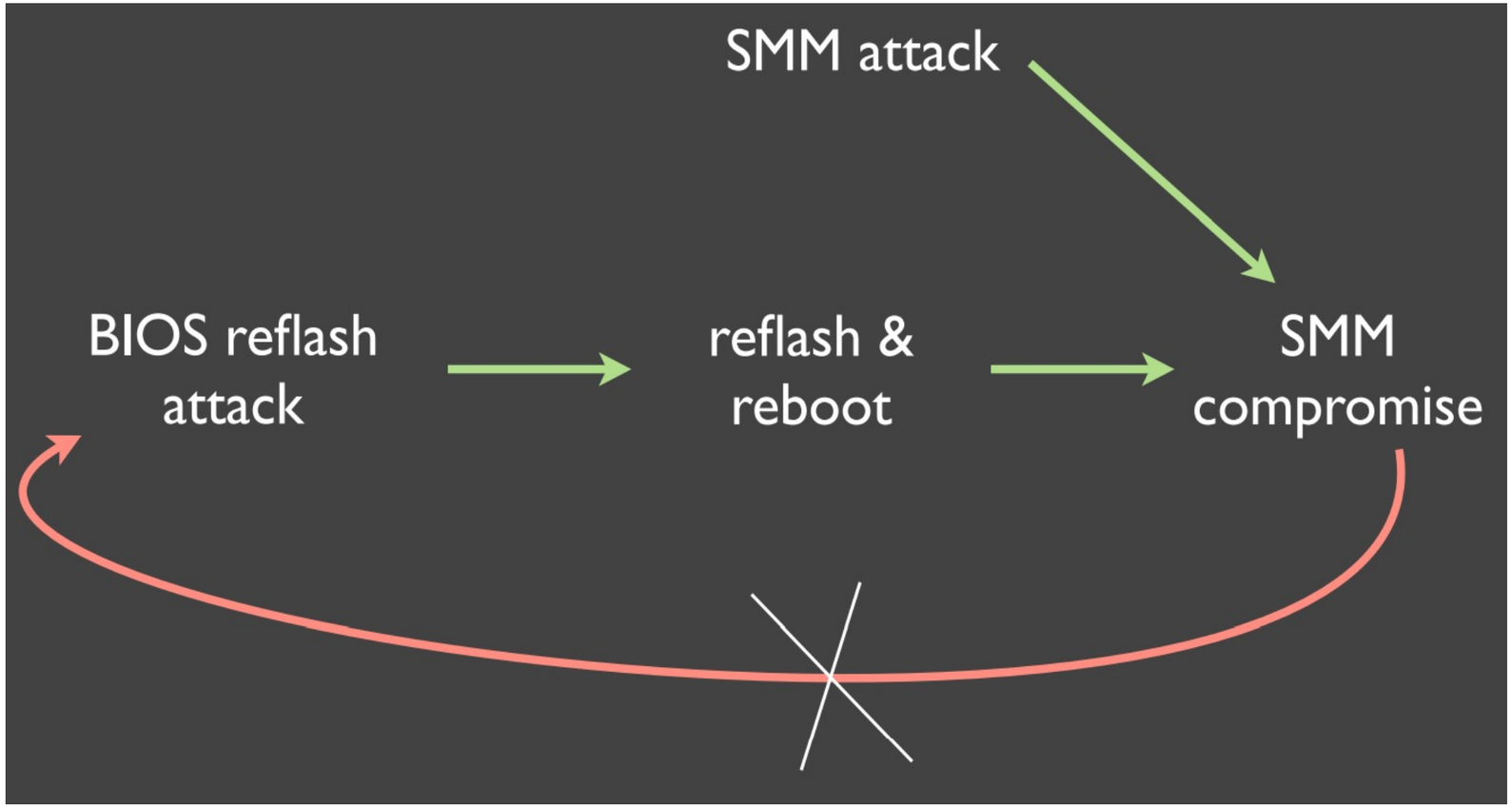

[DHS-CISA-Strategy-to-Fix-Vulnerabilities-Below-the-OS-Among-Worst-Offenders]

**WSMT (Windows SMM Security Mitigation Table) has a static nature by design. Having the mitigation enabled doesn't mean this mitigation is used or configured correctly.**



# Good Old Days of Attacking BIOS

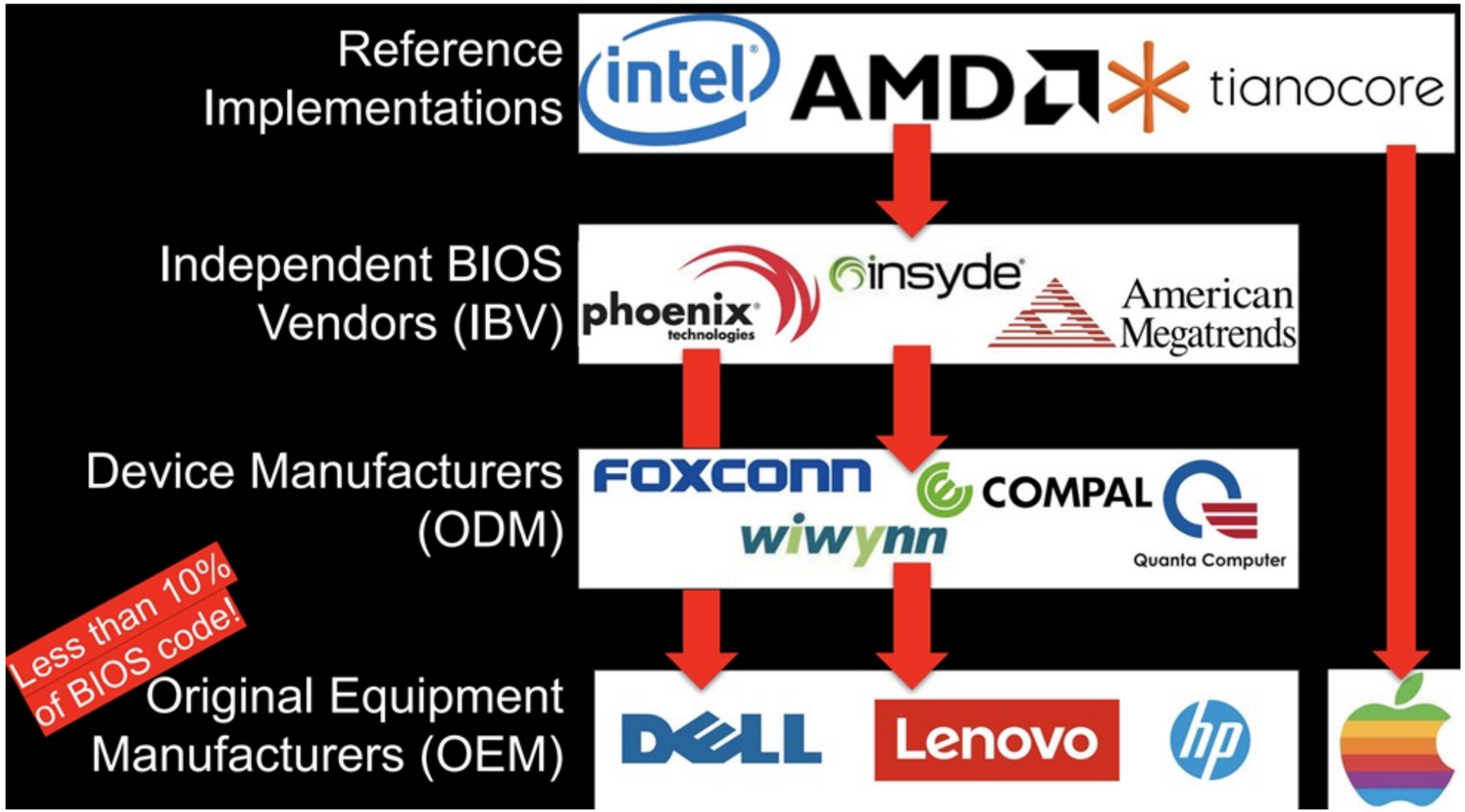
## Does anything change?




BHUSA09-Wojtczuk-AtkIntelBios-SLIDES

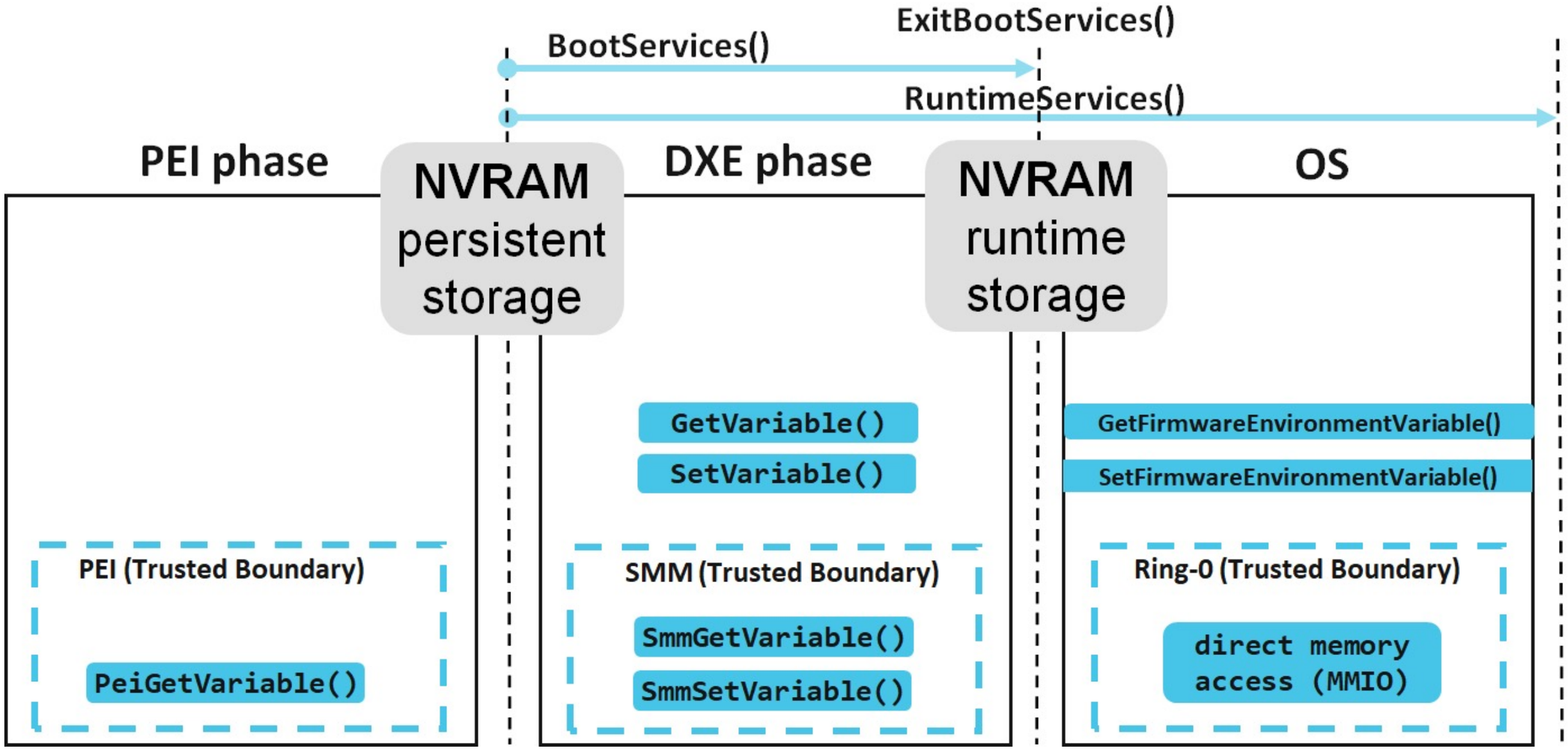
# UEFI Ecosystem Firmware Supply Chain is Hard(coded)

# UEFI Ecosystem Firmware Supply Chain is Hard(coded)




[Roots of Trust and Attestation | Trammell Hudson]

# NVRAM Variables Access During Boot Flow



# NVRAM PEI/DXE/SMM Threat Model

**Attacker Model:**

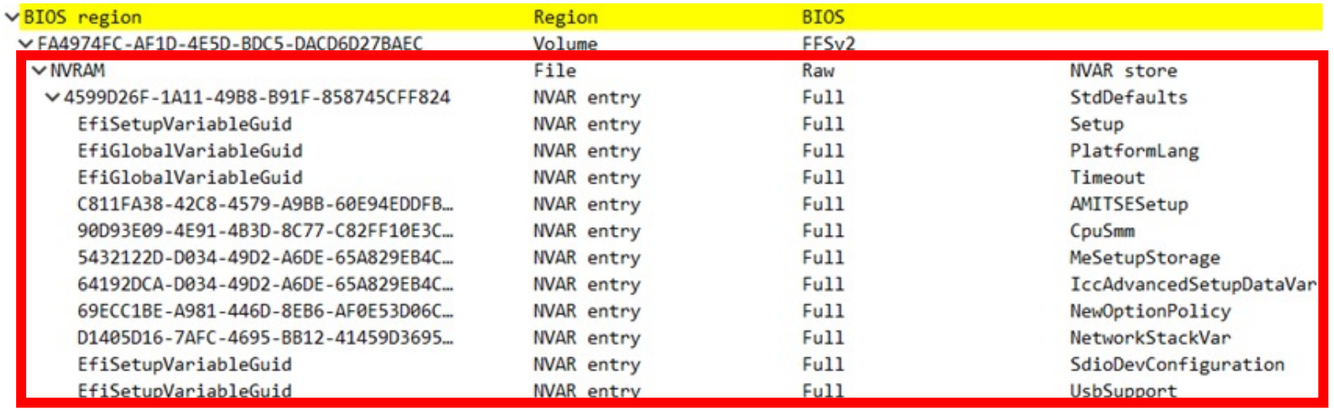
The local attacker uses privileged host OS access to trigger the vulnerability gaining PEI/DXE stage code execution in System Management Mode (SMM).

**Potential Impact:**

PEI/DXE code execution in SMM context allows potential installation of **persistent implants** in the NVRAM SPI flash region. Implant persistence across OS installations, can further **bypass Secure Boot** attacking guest VM's in bare metal cloud deployments.

# NVRAM Persistence on SPI Flash

- NVRAM region is not protected by Intel Boot Guard and can be abused by attacker with physical access (supply chain vector).



| BIOS region | Region | BIOS | |
|---|---|---|---|
| FA4974FC-AF1D-4E5D-BDC5-DACD6D27BAEC | Volume | FFSv2 | |
| NVRAM | File | Raw | NVAR store |
| 4599D26F-1A11-49B8-B91F-858745CFF824 | NVAR entry | Full | StdDefaults |
| EfiSetupVariableGuid | NVAR entry | Full | Setup |
| EfiGlobalVariableGuid | NVAR entry | Full | PlatformLang |
| EfiGlobalVariableGuid | NVAR entry | Full | Timeout |
| C811FA38-42C8-4579-A9BB-60E94EDDFB… | NVAR entry | Full | AMITSESetup |
| 90D93E09-4E91-4B3D-8C77-C82FF10E3C… | NVAR entry | Full | CpuSmm |
| 5432122D-D034-49D2-A6DE-65A829EB4C… | NVAR entry | Full | MeSetupStorage |
| 64192DCA-D034-49D2-A6DE-65A829EB4C… | NVAR entry | Full | IccAdvancedSetupDataVar |
| 69ECC1BE-A981-446D-8EB6-AF0E53D06C… | NVAR entry | Full | NewOptionPolicy |
| D1405D16-7AFC-4695-BB12-41459D3695… | NVAR entry | Full | NetworkStackVar |
| EfiSetupVariableGuid | NVAR entry | Full | SdioDevConfiguration |
| EfiSetupVariableGuid | NVAR entry | Full | UsbSupport |

- Arbitrary code execution via *GetVariable() and memory leak over SetVariable()* is common, attacker can modify persistent NVRAM storage and install fileless DXE/SMM/PEI implant (shellcode).

**Most security solutions inspect only UEFI drivers!**

# UEFI RE Methodology: Hex-Rays+ efiXplorer

```
__int64 (__fastcall *__fastcall sub_2BC(void *a1, EFI_SYSTEM_TABLE *a2))()
{
  EFI_RUNTIME_SERVICES *v2; // rax
  EFI_BOOT_SERVICES *v3; // rbx
  __int64 i; // rax
  char v6; // cl
  char v7; // bl
  __int64 v8; // rax
  __int64 v9; // rdx
  __int64 (__fastcall *result)(); // rax
  UINTN DataSize; // [rsp+50h] [rbp+20h] BYREF
  __int64 v12; // [rsp+58h] [rbp+28h] BYREF

  v2 = a2->RuntimeServices;
  v3 = a2->BootServices;
  AgentHandle = a1;
  gST_13488 = a2;
  gBS_13490 = v3;
  gRT_134A0 = v2;
  if ( sub_38D0(&EFI_TSC_FREQUENCY_GUID_110E0, &DataSize) )
  {
    (v3->AllocatePool)(4i64, 8i64, &DataSize);
    *DataSize = sub_3C7C();
    gBS_13490->InstallConfigurationTable(&EFI_TSC_FREQUENCY_GUID_110E0, DataSize);
    v3 = gBS_13490;
  }
  qword_134A8 = *DataSize;
  (v3->LocateProtocol)(&EFI_HII_STRING_PROTOCOL_GUID_10F50, 0i64, &qword_134C0);
  gBS_13490->LocateProtocol(&EFI_HII_DATABASE_PROTOCOL_GUID_BD00, 0i64, &Interface);
  gBS_13490->LocateProtocol(&EFI_HII_CONFIG_ROUTING_PROTOCOL_GUID_BD10, 0i64, &qword_134D0);
  gBS_13490->LocateProtocol(&EFI_HII_FONT_PROTOCOL_GUID_BCD0, 0i64, &qword_134B8);
  gBS_13490->LocateProtocol(&EFI_HII_IMAGE_PROTOCOL_GUID_BD80, 0i64, &qword_134B0);
  sub_38D0(&EFI_HOB_LIST_GUID_BD60, &qword_134D8);
  sub_38D0(&DXE_SERVICES_TABLE_GUID_10EE0, &qword_134E0);
  qword_14350 = sub_B2B0(&EFI_PHYSICAL_PRESENCE_DATA_GUID_10F20, a1, &unk_BDE0, 0i64);
  dword_13CA8 = *(qword_134D8 + 12);
```

```
[efiXplorer] ----------------------------------------------------------
[efiXplorer] Looking for GetVariable stack/heap overflow
[efiXplorer] GetVariable_1: 0x0000000000000374, GetVariable_2: 0x00000000000004ff
[efiXplorer] GetVariable_1: 0x00000000000004ff, GetVariable_2: 0x000000000000050c
[efiXplorer] GetVariable_1: 0x000000000000050c, GetVariable_2: 0x0000000000000565
[efiXplorer] GetVariable_1: 0x0000000000000565, GetVariable_2: 0x00000000000006f3
[efiXplorer] GetVariable_1: 0x00000000000006f3, GetVariable_2: 0x0000000000000736
[efiXplorer]      overflow can occur here: 0x0000000000000736
[efiXplorer] GetVariable_1: 0x0000000000000736, GetVariable_2: 0x0000000000000960
[efiXplorer] GetVariable_1: 0x0000000000000960, GetVariable_2: 0x0000000000000c4f
[efiXplorer] GetVariable_1: 0x0000000000000c4f, GetVariable_2: 0x0000000000000c5c
[efiXplorer] GetVariable_1: 0x0000000000000c5c, GetVariable_2: 0x0000000000000c69
[efiXplorer] GetVariable_1: 0x0000000000000c69, GetVariable_2: 0x0000000000000d58
[efiXplorer] GetVariable_1: 0x0000000000000d58, GetVariable_2: 0x0000000000000ef9
[efiXplorer] GetVariable_1: 0x0000000000000ef9, GetVariable_2: 0x0000000000001337
[efiXplorer] GetVariable_1: 0x0000000000001337, GetVariable_2: 0x0000000000001344
[efiXplorer] GetVariable_1: 0x0000000000001344, GetVariable_2: 0x0000000000001351
[efiXplorer] GetVariable_1: 0x0000000000001351, GetVariable_2: 0x0000000000001396
[efiXplorer] GetVariable_1: 0x0000000000001396, GetVariable_2: 0x000000000000149b
[efiXplorer] GetVariable_1: 0x000000000000149b, GetVariable_2: 0x0000000000001530
[efiXplorer] GetVariable_1: 0x0000000000001530, GetVariable_2: 0x00000000000015d3
[efiXplorer] GetVariable_1: 0x00000000000015d3, GetVariable_2: 0x00000000000016d8
[efiXplorer] GetVariable_1: 0x00000000000016d8, GetVariable_2: 0x0000000000001729
[efiXplorer] GetVariable_1: 0x0000000000001729, GetVariable_2: 0x000000000000181d
[efiXplorer] ----------------------------------------------------------
[efiXplorer] Looking for SmmGetVariable stack/heap overflow
[efiXplorer] gSmmVar->SmmGetVariable calls finding via EFI_SMM_VARIABLE_PROTOCOL_GUID
[efiXplorer] gSmmVar->SmmGetVariable function finding from 0x0000000000001A60 to 0x0000000000001EE0
[efiXplorer] can't find a EFI_SMM_VARIABLE_PROTOCOL_GUID guid
[efiXplorer] less than 2 GetVariable calls found
[efiXplorer] ----------------------------------------------------------
```



binarly-io/efiXplorer: IDA plugin for UEFI firmware analysis and reverse engineering automation

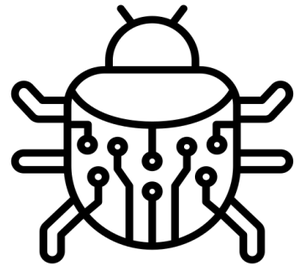
# DSA-2021-103: The Story of Two Buffer Overflows

- MirrorRequest:

NVRAM variable insecure memset leads to stack overflow.

- AepErrorLog:

NVRAM variable mistake in parsing routine leads to heap overflow.



DSA-2021-103: Dell PowerEdge Server Security Update for BIOS Vulnerabilities

# CVE-2021-21555 (DSA-2021-103): AepErrorLog NVRAM variable

**mEraseRecordShare** buffer is allocated on heap.

**AepErrorLog** NVRAM variable is controlled by attacker.

A mistake in variable parsing leads to heap overflow resulting in execution of an attacker controlled payload.

```
1__int64 __fastcall GetEraseLog__vuln()
2{
3  unsigned __int8 i; // [rsp+30h] [rbp-28h]
4  __int64 Status; // [rsp+38h] [rbp-20h]
5  UINTN DataSize; // [rsp+40h] [rbp-18h] BYREF
6  char Tries; // [rsp+48h] [rbp-10h]
7  BOOL v5; // [rsp+4Ch] [rbp-Ch]
8
9  Tries = 3;
10 DataSize = 0i64;
11 if ( is_debug() && is_not_zero(64) )
12   dbgprint(64, "%a(): Start\n", "GetEraseLog");
13 do
14 {
15   Status = gRuntimeServices->GetVariable(L"AepErrorLog", &VendorGuid, 0i64, &DataSize, mEraseRacordShare);
16   --Tries;
17   v5 = Status < 0;
18 }
19 while ( Status < 0 && Tries );
20 if ( Status >= 0 )
21 {
22   mEraseRacordShare[48].ch_ = 0;
23   for ( i = 0; i < 0x30u; ++i )
24     mEraseRacordShare[i].ResetNeeded = 0;
25 }
26 if ( is_debug() && is_not_zero(64) )
27   dbgprint(64, "%a(): Status = %r\n", "GetEraseLog", Status);
28 if ( is_debug() && is_not_zero(64) )
29   dbgprint(64, "%a(): End\n", "GetEraseLog");
30 return Status;
31}
```

HEAP OVERFLOW if var length is > 964 bytes

**DXE: CrystalRidge (C4EB3614-4986-42B9-8C0D-9FE118278908)**

# CVE-2021-21555: AepErrorLog NVRAM variable

**mEraseRecordShare** buffer is allocated on heap.

**AepErrorLog** NVRAM variable is controlled by attacker.

A mistake in variable parsing leads to heap overflow resulting in execution of an attacker controlled payload.

**The payload is not measured and TPM PCR's are not extended.**

**Remote health attestation will not detect the exploitation.**

HEAP OVERFLOW if var length is > 964 bytes

```
1  __int64 __fastcall GetEraseLog__vuln()
2  {
3    unsigned __int8 i; // [rsp+30h] [rbp-28h]
4    __int64 Status; // [rsp+38h] [rbp-20h]
5    UINTN DataSize; // [rsp+40h] [rbp-18h] BYREF
6    char Tries; // [rsp+48h] [rbp-10h]
7    BOOLEAN v5; // [rsp+4Ch] [rbp-Ch]
8
9    Tries = 3;
10   DataSize = 0i64;
11   if ( is_debug() && is_not_zero(64) )
12     dbgprint(64, "%a(): Start\n", "GetEraseLog");
13   do
14   {
15     Status = gRuntimeServices->GetVariable(L"AepErrorLog", &VendorGuid, 0i64, &DataSize, mEraseRacordShare);
16     --Tries;
17   }
18   while ( Status < 0 && Tries );
19   if ( Status >= 0 )
20   {
21     mEraseRacordShare[48].ch_ = 0;
22     for ( i = 0; i < 0x30u; ++i )
23       mEraseRacordShare[i].ResetNeeded = 0;
24   }
25   if ( is_debug() && is_not_zero(64) )
26     dbgprint(64, "%a(): Status = %r\n", "GetEraseLog", Status);
27   if ( is_debug() && is_not_zero(64) )
28     dbgprint(64, "%a(): End\n", "GetEraseLog");
29   return Status;
30 }
```

DXE: CrystalRidge (C4EB3614-4986-42B9-8C0D-9FE118278908)

# DSA-2021-103 (CVE-2021-21556) / INTEL-SA-00463 (CVE-2020-24486) MirrorRequest NVRAM variable

If **MirrorRequest var length** > 5, a subsequent **memset** will overwrite PEI stack with zeroes in PEI phase.

An attacker controls the length of overwritten buffer and can modify parts of saved return addresses to change execution flow which may lead to arbitrary code execution in PEI phase.

```
-000000ED                db ? ; undefined
-000000EC                db ? ; undefined
-000000EB var_EB         db ?
-000000EA var_EA         dd 5 dup(?)
-000000D6 var_D6         db ?
-000000D5 var_D5         db ?
-000000D4 var_D4         dw ?
-000000D2 var_D2         db ?
-000000D1 _MirrorRequest db ?
-000000D0                db ? ; undefined
-000000CF                db ? ; undefined
-000000CE                db ? ; undefined
-000000CD                db ? ; undefined
```

```
DataSize = 5;
if ( ReadOnlyVariable2->GetVariable(
       ReadOnlyVariable2,
       L"MirrorRequest",
       &stru_FFD1F114,
       0,
       &DataSize,
       _MirrorRequest) )
{                                       // char _MirrorRequest[10]; // [esp+6AB3h] [ebp-D1h] BYREF
  memset(_MirrorRequest, 0, DataSize);
}
```

Stack buffer: will overwrite saved EIP if length is > 0xD1 bytes

PEI: UncoreInitPeim (D71C8BA4-4AF2-4D0D-B1BA-F2409F0C20D3)

# CVE-2021-21556/CVE-2021-24486 (INTEL-SA-00463)

## MirrorRequest NVRAM variable

If **MirrorRequest var length** > 5, a subsequent **memset** will overwrite PEI stack with zeroes, in PEI phase.

An attacker controls the length of the overwrite and can modify parts of saved return addresses to change execution flow which may lead to arbitrary code execution in PEI phase.

```
-000000ED                db ? ; undefined
-000000EC                db ? ; undefined
-000000EB var_EB         db ?
-000000EA var_EA         dd 5 dup(?)
-000000D6 var_D6         db ?
-000000D5 var_D5         db ?
-000000D4 var_D4         dw ?
-000000D2 var_D2         db ?
-000000D1 _MirrorRequest db ?
-000000D0                db ? ; undefined
-000000CF                db ? ; undefined
-000000CE                db ? ; undefined
-000000CD                db ? ; undefined
```

**The payload is not measured and TPM PCR's are not extended.**

**Remote health attestation will not detect the exploitation.**

```
DataSize = 5;
if ( ReadOnlyVariable2->GetVariable(
       ReadOnlyVariable2,
       L"MirrorRequest",
       &stru_FFD1F114,
       0,
       &DataSize,
       _MirrorRequest) )
{                                       // char _MirrorRequest[10]; // [esp+6AB3h] [ebp-D1h] BYREF
  memset(_MirrorRequest, 0, DataSize);
}
```

Stack buffer: will overwrite saved EIP if length is > 0xD1 bytes

**PEI:** UncoreInitPeim (D71C8BA4-4AF2-4D0D-B1BA-F2409F0C20D3)

# Attacking (pre)EFI Ecosystem

# Pre-EFI Initialization and NVRAM variables

- PEI code also reads configuration data from NVRAM variables
- By design, NVRAM is considered read only in PEI stage
- EFI variables are as good attack surface for PEI as they are for DXE
- The API for reading EFI vars is a bit different for PEI though

```
#define EFI_PEI_READ_ONLY_VARIABLE2_PPI_GUID \
  { 0x2ab86ef5, 0xecb5, 0x4134, { 0xb5, 0x56, 0x38, 0x54, 0xca, 0x1f,
0xe1, 0xb4 } }

typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_VARIABLE2)(
  IN CONST  EFI_PEI_READ_ONLY_VARIABLE2_PPI *This,
  IN CONST  CHAR16                          *VariableName,
  IN CONST  EFI_GUID                        *VariableGuid,
  OUT       UINT32                          *Attributes,
  IN OUT    UINTN                           *DataSize,
  OUT       VOID                            *Data OPTIONAL
  );
```

Scan for this GUID to quickly locate PEI code that reads NVRAM variables (and potentially has vulns that have to be fixed)

# Pre-EFI Initialization and NVRAM variables

```
(*PeiServices)->LocatePpi(PeiServices, &gEfiPeiReadOnlyVariable2PpiGuid, 0, 0, &ReadOnlyPpi);
ZeroMem(syscg_stack, 2048);
ReadOnlyPpi->GetVariable(ReadOnlyPpi, L"syscg", &gSsaBiosVariablesGuid, 0, &DataSize, syscg_stack);
syscg = AllocatePool(DataSize);
memcpy_0(syscg, syscg_stack, DataSize);
```

This is an example of EFI var reading from UncoreInitPeim.efi running on Grantley, and there are problems.

# How to know when your Uncore is up to no good

```
(*PeiServices)->LocatePpi(PeiServices, &gEfiPeiReadOnlyVariable2PpiGuid, 0, 0, &ReadOnlyPpi);
ZeroMem(syscg_stack, 2048);
ReadOnlyPpi->GetVariable(ReadOnlyPpi, L"syscg", &gSsaBiosVariablesGuid, 0, &DataSize, syscg_stack);
syscg = AllocatePool(DataSize);
memcpy_0(syscg, syscg_stack, DataSize);
```

- No status check after `GetVariable()`
- `DataSize` may become larger than 2048 bytes if `GetVariable()` call fails with `EFI_BUFFER_TOO_SMALL` (that is, if “syscg” NVRAM variable is longer than 2048 bytes)
- No check for a return value of `AllocatePool()`
- `memcpy()` will not overflow `syscg`, but may copy some stack memory to it

This is not particularly useful but looks like this code may be full of surprises. Let’s see what it does *next*.

# Hey Uncore, GetVariable() API is dangerous when called in a loop

It may change DataSize and you overflow your buffer on the next iteration with an attacker-controlled data if you don't check the returned state:

This code reads a chain of EFI variables. Each var contains a name of the next one in the beginning of the data.

```
CHAR16 ChunkName[6]; // [esp+810h] [ebp-14h] BYREF
ReadOnlyPpi->GetVariable(ReadOnlyPpi, SourceVarName, SourceVarGuidPtr, 0, &DataSize, NextChunkNameAndData);

ZeroMem(ChunkName, 12);
memcpy_0(ChunkName, NextChunkNameAndData, 10);
memcpy_0(ImageBase, &NextChunkNameAndData[10], DataSize - 10);
for ( result = StrCmp(ChunkName, &Zero); result; result = StrCmp(ChunkName, &Zero) )
{
  ImageBase = ImageBase + DataSize - 10;
  ReadOnlyPpi->GetVariable(ReadOnlyPpi, ChunkName, SourceVarGuidPtr, 0, &DataSize, NextChunkNameAndData);
  ZeroMem(ChunkName, 12);
  memcpy_0(ChunkName, NextChunkNameAndData, 10);
  if ( DataSize != 10 )
  {
    memcpy_0(ImageBase, &NextChunkNameAndData[10], DataSize - 10);
    FlushFlashRegion();
  }
}
```

# Hey Uncore, GetVariable() API is dangerous when called in a loop

It may change DataSize and you overflow your buffer on the next iteration with an attacker-controlled data if you don't check the returned state:

DataSize is not reinitialized before GetVariable(). Therefore, DataSize value is controlled by an attacker.

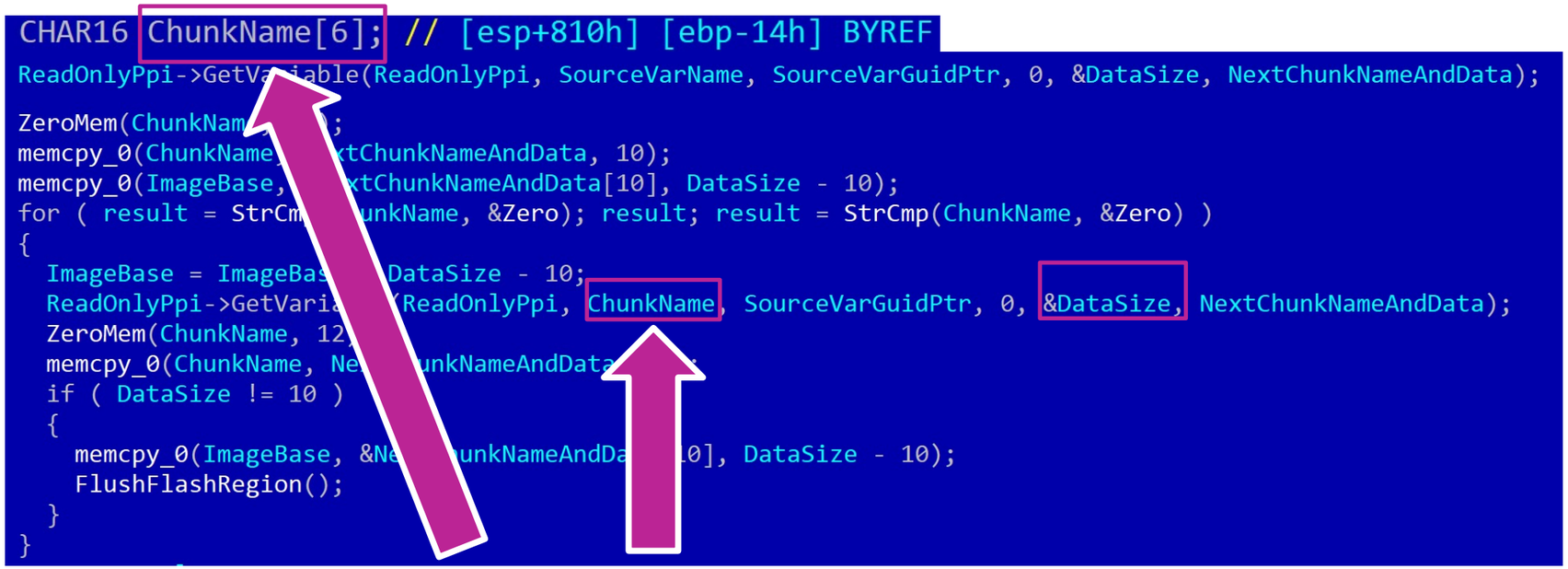
```
CHAR16 ChunkName[6]; // [esp+810h] [ebp-14h] BYREF
ReadOnlyPpi->GetVariable(ReadOnlyPpi, SourceVarName, SourceVarGuidPtr, 0, &DataSize, NextChunkNameAndData);

ZeroMem(ChunkName, 12);
memcpy_0(ChunkName, NextChunkNameAndData, 10);
memcpy_0(ImageBase, &NextChunkNameAndData[10], DataSize - 10);
for ( result = StrCmp(ChunkName, &Zero); result; result = StrCmp(ChunkName, &Zero) )
{
  ImageBase = ImageBase + DataSize - 10;
  ReadOnlyPpi->GetVariable(ReadOnlyPpi, ChunkName, SourceVarGuidPtr, 0, &DataSize, NextChunkNameAndData);
  ZeroMem(ChunkName, 12);
  memcpy_0(ChunkName, NextChunkNameAndData, 10);
  if ( DataSize != 10 )
  {
    memcpy_0(ImageBase, &NextChunkNameAndData[10], DataSize - 10);
    FlushFlashRegion();
  }
}
```

This may be large

# Hey Uncore, GetVariable() API is dangerous when called in a loop

It may change DataSize and you overflow your buffer on the next iteration with an attacker-controlled data if you don't check the returned state:

ChunkName is a stack buffer.
When GetVariable encounters a large var (length > DataSize), DataSize is overwritten with the actual length.
On the next iteration a stack overflow occurs when GetVariable() assumes ChunkName length is at least DataSize bytes.




This stack buffer is overflowed. We don't have stack cookies or ASLR in PEI code.

**We don’t even need to exploit this stack overflow to make our unsigned code execute during PEI.**

This *feature* was designed to run arbitrary unsigned code blobs stored in EFI variables!

**We don’t even need to exploit this stack overflow to make our unsigned code execute during PEI.**

This *feature* was designed to run arbitrary unsigned code blobs stored in EFI variables!

Yes, this is a reference code.

# Uncore features unsigned module loading

This walks the EFI var chain starting from variable “toolh” and builds a contiguous 32bit PE image.

The payload may be 100kb in size or even more, available NVRAM space is the limit.

This executes the PE entry point.

```
TotalConfigs = *(syscg + 0x10);
EvLoadTool(host, syscg, &ConfigIndex, &ImageBase);
if ( TotalConfigs )
{
  ConfigIndex = 0;
  do
  {
    EvLoadConfig(ConfigIndex, host, syscg, TotalConfigs, &v14);

    Entry = GetPEEntry(host, ImageBase);
    Entry(Ppi, v6);
    sub_FFE6667E(host);
    result = ++ConfigIndex;
  }
  while ( ConfigIndex < TotalConfigs );
}
else
{
  Entry = GetPEEntry(host, ImageBase);
  Entry(Ppi, 0);
  return sub_FFE6667E(host);
}
```

# Uncore features unsigned module loading

This walks the EFI var chain starting from variable “toolh” and builds a contiguous 32bit PE image.

The payload may be 100kb in size or even more, available NVRAM space is the limit.

This executes the PE entry point.

```
TotalConfigs = *(syscg + 0x10);
EvLoadTool(host, syscg, &ConfigIndex, &ImageBase);
if ( TotalConfigs )
{
  ConfigIndex = 0;
  do
  {
    EvLoadConfig(ConfigIndex, host, syscg, TotalConfigs, &v14);

    Entry = GetPEEntry(host, ImageBase);
    Entry(Ppi, v6);
    sub_FFE6667E(host);
    result = ++ConfigIndex;
  }
  while ( ConfigIndex < TotalConfigs );
}
else
{
  Entry = GetPEEntry(host, ImageBase);
  Entry(Ppi, 0);
  return sub_FFE6667E(host);
}
```

# Uncore features unsigned module loading

This walks the EFI var chain starting from variable “toolh” and builds a contiguous 32bit PE image.

The payload may be 100kb in size or even more, available NVRAM space is the limit.

This executes the PE entry point.

```
TotalConfigs = *(syscg + 0x10);
EvLoadTool(host, syscg, &ConfigIndex, &ImageBase);
if ( TotalConfigs )
{
  ConfigIndex = 0;
  do
  {
    EvLoadConfig(ConfigIndex, host, syscg, TotalConfigs, &v14);

    Entry = GetPEEntry(host, ImageBase);
    Entry(Ppi, v6);
    sub_FFE6667E(host);
    result = ++ConfigIndex;
  }
  while ( ConfigIndex < TotalConfigs );
}
else
{
  Entry = GetPEEntry(host, ImageBase);
  Entry(Ppi, 0);
  return sub_FFE6667E(host);
}
```

# Uncore features unsigned module loading

This walks the EFI var chain starting from variable “toolh” and builds a contiguous 32bit PE image.

The payload may be 100kb in size or even more, available NVRAM space is the limit.

This executes the PE entry point.

```
TotalConfigs = *(syscg + 0x10);
EvLoadTool(host, syscg, &ConfigIndex, &ImageBase);
if ( TotalConfigs )
{
  ConfigIndex = 0;
  do
  {
    EvLoadConfig(ConfigIndex, host, syscg, TotalConfigs, &v14);
    Entry = GetPEEntry(host, ImageBase);
    Entry(Ppi, v6);
    sub_FFE6667E(host);
    result = ++ConfigIndex;
  }
  while ( ConfigIndex < TotalConfigs );
}
else
{
  Entry = GetPEEntry(host, ImageBase);
  Entry(Ppi, 0);
  return sub_FFE6667E(host);
}
```

**The loaded unsigned code is not measured, TPM PCRs are not extended.**

**Remote health attestation will not detect the exploitation.**

# What is the purpose of this feature?

This was intended to be used for debugging or testing purposes.

Intel guidance is this feature is not supposed to be enabled if a physical presence has not been established.

BIOS vendors are supposed to implement their own code for establishing physical presence. For example, a physical jumper setting may be used.

If this feature is not used by OEM vendors then this feature should not be compiled in.

# What is the purpose of this feature?

This was intended to be used for debugging or testing purposes.

Intel guidance is this feature is not supposed to be enabled if a physical presence has not been established.

BIOS vendors are supposed to implement their own code for establishing physical presence. For example, a physical jumper setting may be used.

This is what a poor implementation looks like:

```
char IsPhysicalPresenceEstablished()
{
  return 1;
}
```

# Not all BIOS vendors are oblivious to the guidelines

Dell has permanently disabled a setup knob setting for this feature, making the module loading code unreachable.

Good job!

# How did it happen?

1. Intel implemented a dummy function for a physical presence which always returns TRUE. Should have put “return FALSE”.
2. IBVs have been told by Intel to implement the code which checks for physical jumpers etc.
3. Instead, IBVs just reused Intel’s reference code implementation without making any changes to the relevant code. Now Grantley+ server platforms have this presence check effectively disabled because of this.

"Reference implementations" often become the defacto implementation — due care with proper (safe) defaults should be the norm.

Actual bug is here

```
char IsPhysicalPresenceEstablished()
{
return 1;
}
```

# CVE-2021-0114: Intel’s response

- BSSA Target-Based Loader is a Design for Manufacturing (DFx) feature intended to be used on manufacturing lines.
- Intel's Implementation and Security Guidelines state this feature is not intended to be included in production use, but if it is included then it must be further secured. A suggested method is to confirm physical presence and would vary by different motherboards and systems each requiring their own implementation.
- Transparency is part of our security first pledge. Intel has published an Intel® BSSA DFT Advisory (INTEL-SA-00525) and has developed updated reference UEFI Firmware for potentially affected platforms which includes a physical presence check that always returns False thus necessitating IBV's to complete the implementation for a specific motherboard or system if they included this DFx feature in production.

# Looking up the details

Let’s look up the GUID for “toolh” and “syscg” EFI variables used by this feature.

https://edk2.groups.io › devel › message

Re: [edk2-platforms][PATCH V1 01/37] CoffeelakeSiliconPkg: Add ...

+gSsaBiosVariablesGuid = {**0x43eeffe8**, 0xa978, 0x41dc, {0x9d, 0xb6, 0x54, 0xc4, 0x27, 0xf2 0x7e, 0x2a}} +gSsaBiosResultsGuid = {0x8f4e928, 0xf5f, 0x46d4, ...

# Looking up the details

https://edk2.groups.io/g/devel/message/46002

```
+## Include/SsaCommonConfig.h
+gSsaPostcodeHookGuid = {0xADF0A27B, 0x61A6, 0x4F18, {0x9E, 0xAC, 0x46, 0x87, 0xE7, 0x9E, 0x6F, 0xBB}}
+gSsaBiosVariablesGuid = {0x43eeffe8, 0xa978, 0x41dc, {0x9d, 0xb6, 0x54, 0xc4, 0x27, 0xf2, 0x7e, 0x2a}}
+gSsaBiosResultsGuid = {0x8f4e928, 0xf5f, 0x46d4, {0x84, 0x10, 0x47, 0x9f, 0xda, 0x27, 0x9d, 0xb6}}
...
+##
+## SystemAgent
+##
+gSsaBiosCallBacksPpiGuid  =  {0x99b56126, 0xe16c, 0x4d9b, {0xbb, 0x71, 0xaa, 0x35, 0x46, 0x1a, 0x70, 0x2f}}
+gSsaBiosServicesPpiGuid   =  {0x55750d10, 0x6d3d, 0x4bf5, {0x89, 0xd8, 0xe3, 0x5e, 0xf0, 0xb0, 0x90, 0xf4}}
```

So, the *feature* is called "SSA".

And the payload is probably called "SystemAgent"

# Looking up the details

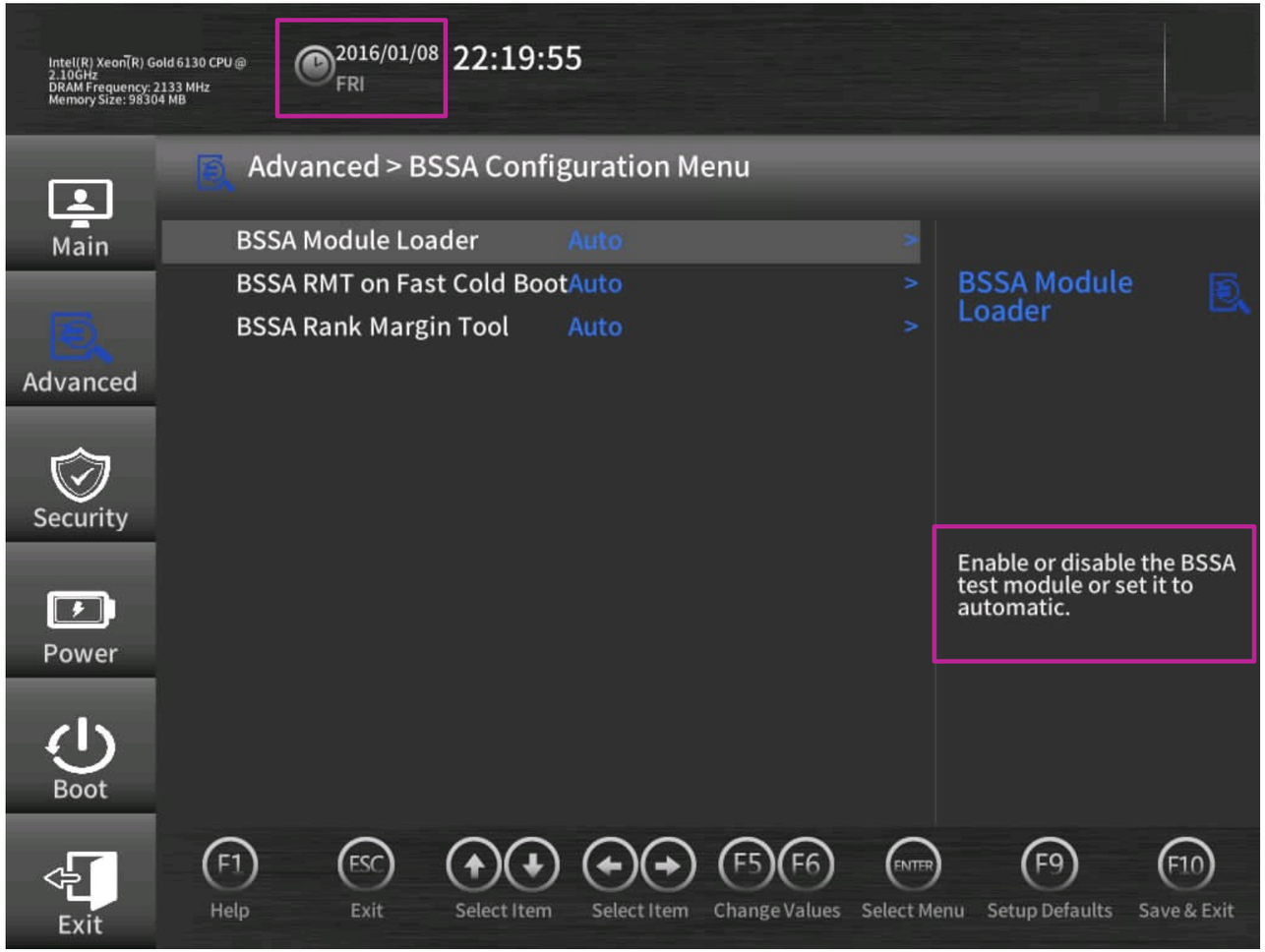



Table 50. MRC Progress Codes

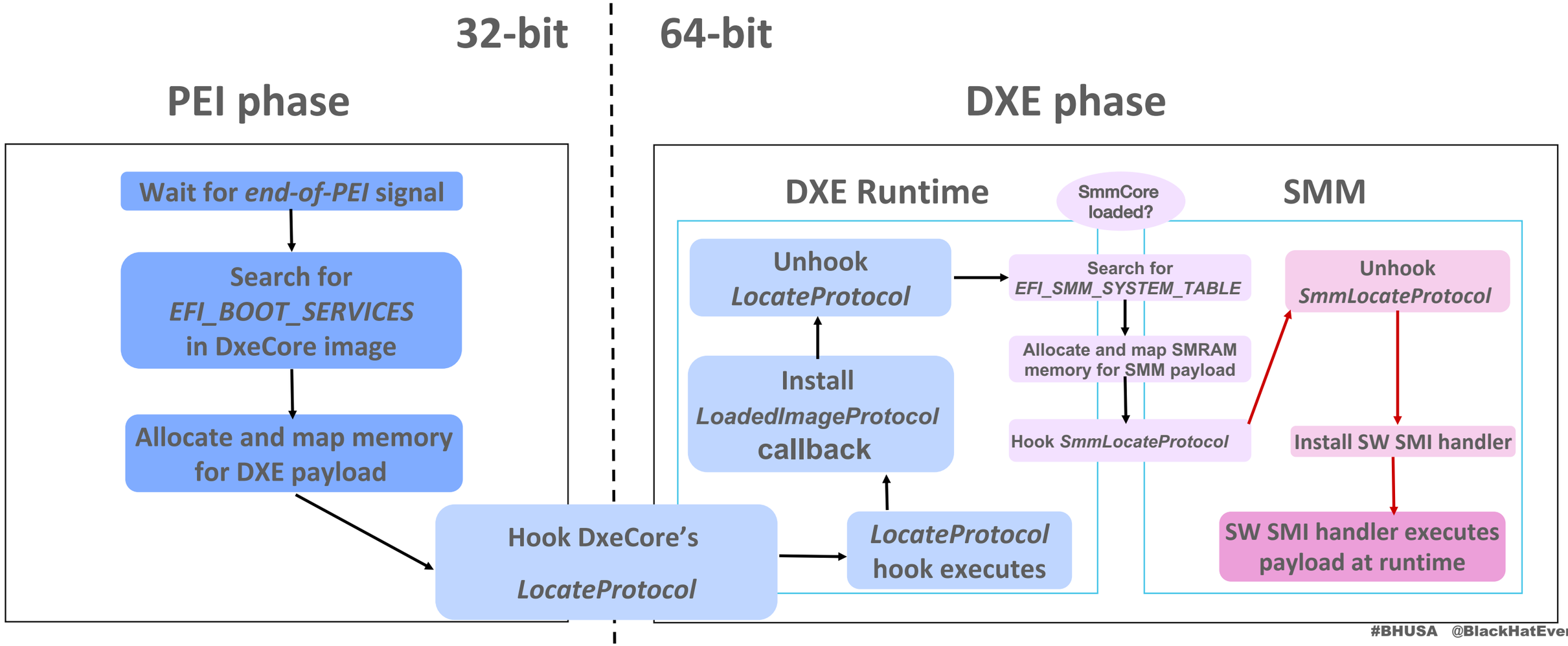
| Post Code (Hex) | Upper Nibble | | | | Lower Nibble | | | | Description |
|---|---|---|---|---|---|---|---|---|---|
| | 8h | 4h | 2h | 1h | 8h | 4h | 2h | 1h | |
| B0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | Detect DIMM population |
| B1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Set DDR4 frequency |
| B2 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Gather remaining SPD data |
| B3 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Program registers on the memory controller level |
| B4 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | Evaluate RAS modes and save rank information |
| B5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | Program registers on the channel level |
| B6 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Perform the JEDEC defined initialization sequence |
| B7 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | Train DDR4 ranks |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Train DDR4 ranks |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Train DDR4 ranks – Read DQ/DQS training |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Train DDR4 ranks – Receive enable training |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Train DDR4 ranks – Write DQ/DQS training |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Train DDR4 ranks – DDR channel training done |
| B8 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | Initialize CLTT/OLTT |
| B9 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | Hardware memory test and init |
| BA | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | Execute software memory init |
| BB | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | Program memory map and interleaving |
| BC | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | Program RAS configuration |
| BE | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Execute BSSA RMT |
| BF | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | MRC is done |

116

BSSA was available since at least 2016

[Intel® Server System M50CYP1UR TPS]

# Payload Example: Escalate from PEI to SMM



# DEMO TIME

# Safeguarding UEFI Ecosystem

# Let’s briefly talk about

# EFI Development Kit (EDK II)…

Let’s briefly talk about

EFI Development Kit (EDK II)…

… I mean CVE-2021-28216 ;-)

# EFI Development Kit (EDK II)…

- An open-source implementation of the Unified Extensible Firmware Interface (UEFI) support by the community (TianoCore)
- EDK II is advertised as a modern, feature-rich, cross-platform firmware development environment for the UEFI and UEFI Platform Initialization (PI) specifications
- EDK II supports tons of platforms including:
  - Intel
  - AMD
  - ARM
  - Ampere
  - HiSilicon
  - BeagleBoard
  - Marvell
  - Raspbery Pi
  - RISC-V
  - Socionext
  - NXP
  - More…

# EFI Development Kit (EDK II)…

- An open-source implementation of the Unified Extensible Firmware Interface (UEFI) support by the community (TianoCore)
- EDK II is advertised as a modern, feature-rich, cross-platform firmware development environment for the UEFI and UEFI Platform Initialization (PI) specifications

- EDK II supports tons of platforms including:
  - Intel
  - AMD
  - ARM
  - Ampere
  - HiSilicon
  - BeagleBoard
  - Marvell
  - Raspbery Pi
  - RISC-V
  - Socionext
  - NXP
  - More…

One bug…
… to rule them all !

# EFI Development Kit (EDK II)…

- An open-source implementation of the Unified Extensible Firmware Interface (UEFI) support by the community (TianoCore)
- EDK II is advanced, modern, feature-rich, cross-platform firmware development environment for the UEFI and UEFI Platform Initialization (PI) specifications

- EDK II supports tons of platforms, including:
  - Intel
  - AMD
  - ARM
  - Ampere
  - HiSilicon
  - BeagleBoard
  - Marvell
  - Raspbery Pi
  - RISC-V
  - Socionext
  - NXP
  - More…

**For example, CVE-2021-28216**

One bug…
… to rule them all !

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

Arbitrary overwrite

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob   = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

Arbitrary overwrite

# CVE-2021-28216

```
  //
  // Update S3 boot records into the basic boot performance table.
  //
  VarSize = sizeof (PerformanceVariable);
  Status = VariableServices->GetVariable (
                        VariableServices,
                        EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                        &gEfiFirmwarePerformanceGuid,
                        NULL,
                        &VarSize,
                        &PerformanceVariable
                        );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
  // Dump PEI boot records
  //
  FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
  GuidHob    = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
  while (GuidHob != NULL) {
    FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
    PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
    CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),

(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries));
    GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
    FirmwarePerformanceTablePtr +=
(UINTN)(PeiPerformanceLogHeader->SizeOfAllEntries);
  }  //
  // Update Table length.
  //
  ((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

Arbitrary overwrite

Semi-controllable write-what-where

# CVE-2021-28216

```
//
// Update S3 boot records into the basic boot performance table.
//
VarSize = sizeof (PerformanceVariable);
Status = VariableServices->GetVariable (
                    VariableServices,
                    EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
                    &gEfiFirmwarePerformanceGuid,
                    NULL,
                    &VarSize,
                    &PerformanceVariable
                    );
if (EFI_ERROR (Status)) {
  return Status;
}
BootPerformanceTable = (UINT8*) (UINTN)
PerformanceVariable.BootPerformanceTablePointer;  //
// Dump PEI boot records
//
FirmwarePerformanceTablePtr = (BootPerformanceTable + sizeof
(BOOT_PERFORMANCE_TABLE));
GuidHob    = GetFirstGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid);
while (GuidHob != NULL) {
  FirmwarePerformanceData = GET_GUID_HOB_DATA (GuidHob);
  PeiPerformanceLogHeader = (FPDT_PEI_EXT_PERF_HEADER *)
FirmwarePerformanceData;
  CopyMem (FirmwarePerformanceTablePtr,
FirmwarePerformanceData + sizeof (FPDT_PEI_EXT_PERF_HEADER),
(UINTN)PeiPerformanceLogHeader->SizeOfAllEntries));
  GuidHob = GetNextGuidHob
(&gEdkiiFpdtExtendedFirmwarePerformanceGuid, GET_NEXT_HOB
(GuidHob));
  FirmwarePerformanceTablePtr +=
(UINTN)PeiPerformanceLogHeader->SizeOfAllEntries);
} //
// Update Table length.
//
((BOOT_PERFORMANCE_TABLE *) BootPerformanceTable)-
>Header.Length = (UINT32)((UINTN)FirmwarePerformanceTablePtr -
(UINTN)BootPerformanceTable);
```

Arbitrary overwrite

The payload is not measured and TPM PCR's are not extended.

Remote health attestation will not detect the exploitation.

Semi-controllable write-what-where

Let’s briefly talk about exploitability…

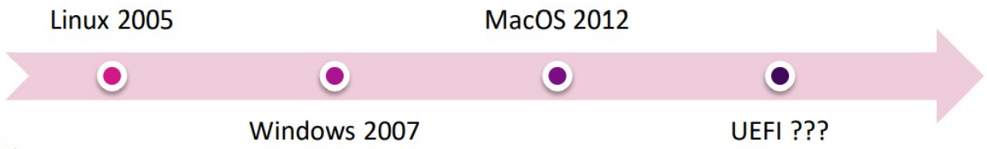
… Let’s talk about mitigations and hardening…

… below the OS

# “The popularity of UEFI and its lack of memory protection enforcements attract exploitation.”

**BAD NEWS**: There are no ways to apply Vulnerability Mitigations below the OS

**A**ddress
**S**pace
**L**ayout
**R**andomization




**MORE BAD NEWS**: Most criminal and advanced threat actors exploit Vulnerabilities Below the OS that affect UEFI

**EVEN MORE BAD NEWS**: In general, existing memory protections (NX) are not enforced due to vendor non-compliance



DHS-CISA-Strategy-to-Fix-Vulnerabilities-Below-the-OS-Among-Worst-Offenders

# EFI Development Kit (EDK II) documentation…

## Compatbility with other features

| | Stack Guard | NULL Ptr | Heap Guard | Mem Profile | NX Stack | NX/RO Mem | Image Protect | Static Paging | SMI Profile | SMM Profile |
|---|---|---|---|---|---|---|---|---|---|---|
| Stack Guard | N/A | V | V | V | V | V | V | V | V | V |
| NULL Pointer | V | N/A | V | V | V | V | V | V | V | V |
| Heap Guard | V | V | N/A | V | V | V | V | N | V | V |
| Mem Profile | V | V | V | N/A | V | V | V | V | V | V |
| NX Stack | V | V | V | V | N/A | V | V | V | V | V |
| NX/RO Memory | V | V | V | V | V | N/A | V | V | V | V |
| Image Protect | V | V | V | V | V | V | N/A | V | V | V |
| SMM Static Paging (*) | V | V | N | V | V | V | V | N/A | V | N |
| SMI Profile (*) | V | V | V | V | V | V | V | V | N/A | V |
| SMM Profile (*) | V | V | V | V | V | V | V | N | V | N/A |

Production Feature | Debug Feature

- **BLUE** means a production feature which might be enabled in the final production.
- **YELLOW** means a debug feature which need be disabled in the final production.
- **“V”** means 2 features can be enabled together.
- **“N”** means 2 feature must not be enabled together.
- **(*)** means this feature is for System Management Mode (SMM) only.
- **No (*)** means this feature can be enabled for DXE or SMM.

https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue/summary

# EFI Development Kit (EDK II) documentation…

## Compatbility with other features

| | Stack Guard | NULL Ptr | Heap Guard | Mem Profile | NX Stack | NX/RO Mem | Image Protect | Static Paging | SMI Profile | SMM Profile |
|---|---|---|---|---|---|---|---|---|---|---|
| Stack Guard | N/A | V | V | V | V | V | V | V | V | V |
| NULL Pointer | V | N/A | V | V | V | V | V | V | V | V |
| Heap Guard | V | V | N/A | V | V | V | V | V | V | V |
| Mem Profile | V | V | V | N/A | V | V | V | V | V | V |
| NX Stack | V | V | V | V | N/A | V | V | V | V | V |
| NX/RO Memory | V | V | V | V | V | N/A | V | V | V | V |
| Image Protect | V | V | V | V | V | V | N/A | V | V | V |
| SMM Static Paging (*) | V | V | N | V | V | V | V | N/A | V | N |
| SMI Profile (*) | V | V | V | V | V | V | V | V | N/A | V |
| SMM Profile (*) | V | V | V | V | V | V | V | N | V | N/A |

| Production Feature | Debug Feature |
|---|---|

- **BLUE** means a production feature which might be enabled in the final production.
- **YELLOW** means a debug feature which need be disabled in the final production.
- **“V”** means 2 features can be enabled together.
- **“N”** means 2 feature must not be enabled together.
- **(*)** means this feature is for System Management Mode (SMM) only.
- **No (*)** means this feature can be enabled for DXE or SMM.

**“May” → not mandatory**

**Moreover…**

https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue/summary

# EFI Development Kit (EDK II) documentation…

- Stack canaries
- DEP
- ASLR
- NULL Pointers
- Limitations

# EFI Development Kit (EDK II) documentation...

- **Stack canaries**

Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The stack check feature is disabled by default. The reason is that EDK II does not link against any compiler provided libraries. If `/G`

- **DEP**

- **ASLR**

- **NULL Pointers**

- **Limitations**

# EFI Development Kit (EDK II) documentation…

- **Stack canaries**

> Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The stack check feature is disabled by default. The reason is that EDK II does not link against any compiler provided libraries. If `/G`

- **DEP**

> The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] specification allows "Stack may be marked as non-executable in identity mapped page tables." UEFI also defines

- **ASLR**

- **NULL Pointers**

- **Limitations**

# EFI Development Kit (EDK II) documentation…

- **Stack canaries**

> Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The stack check feature is disabled by default. The reason is that EDK II does not link against any compiler provided libraries. If `/G`

- **DEP**

> The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] specification allows "Stack may be marked as non-executable in identity mapped page tables." UEFI also defines

- **ASLR**

> The current EDK II code does not support address space randomization. T

- **NULL Pointers**

- **Limitations**

# EFI Development Kit (EDK II) documentation…

- **Stack canaries**

Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The stack check feature is disabled by default. The reason is that EDK II does not link against any compiler provided libraries. If `/G`

- **DEP**

The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] specification allows "Stack may be marked as non-executable in identity mapped page tables." UEFI also defines

- **ASLR**

The current EDK II code does not support address space randomization. T

- **NULL Pointers**

Zero address is considered as an invalid address in most programs. However, in x86 systems, the zero address is valid address in legacy BIOS because the 16bit interrupt vector table (IVT) is at address zero. In current UEFI firmware, zero address is always mapped.

- **Limitations**

# EFI Development Kit (EDK II) documentation…

- **Stack canaries**

Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The s
disabled by default. The reason is that EDK II does not link against any compiler pr

- **DEP**

The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] specification allows "Stack may be
marked as non-executable in identity mapped page tables." UEFI also defines

- **ASLR**

The current EDK II code does not support address space random

- **NULL Pointers**

Zero address is considered as an invalid address in most prog
address is valid address in legacy BIOS because the 16bit interrupt vector table (IVT) is at address zero. In
current UEFI firmware, zero address is always mapped.

- **Limitations**

The guard in Pre-EFI Initialization (PEI) phase is not supported yet, because most Intel® Architecture (IA) platforms only supports 32bit PEI and paging is not enabled.
From technical perspective, we can add paging-based guard after the permanent memory is initialized in PEI. Stack guard, heap guard or NULL pointer detection can be enabled.

allocation need 12K memory. The heap guard feature will increase memory consumption and may cause memory out of resource. Especially, the System Management Mode (SMM) code runs in the limited System Management Mode RAM (SMRAM) (4M or 8M).

We have observed the performance downgrade in UEFI Shell,

For heap pool detection, we cannot enable both underflow and overflow detection in one image, because the guard page must be 4K aligned and the allocated pool is either adjacent to head guard page or tail guard page.

# EFI Development Kit (EDK II) documentation…

- **Stack canaries**

Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The s
disabled by default. The reason is that EDK II does not link against any compiler pr

- **DEP**

The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] specification allows "Stack may be
marked as non-executable in identity mapped page tables." UEFI also define

- **ASLR**

The current EDK II code does not support address space random

- **NULL Pointers**

Zero address is considered as an invalid address in most prog
address is valid address in legacy BIOS because the 16bit interrupt vector table (IVT) is at address zero. In
current UEFI firmware, zero address is always mapped.

- **Limitations**

The guard in Pre-EFI Initialization (PEI) phase is not supported yet, because most Intel® Architecture (IA) platforms only supports 32bit PEI and paging is not enabled.
From technical perspective, we can add paging-based guard after the permanent memory is initialized in PEI. Stack guard, heap guard or NULL pointer detection can be enabled.

allocation need 12K memo The heap guard feature will increase memory consumption and may
e memory out of resource. Especially, the System Management Mode (SMM) code runs in the
limited System Management Mode RAM (SMRAM) (4M or 8M).

We have observed the performance downgrade in UEFI Shell,

For heap pool detection, we cannot enable both underflow and overflow detection in one image, because the guard page must be 4K aligned and the allocated pool is either adjacent to head guard page or tail guard page.

**In practice, mitigations & hardening are rarely (if ever) enabled…**

# What about EDR / XDR / ATP ?

- **Dramatically limited comparing to OS solutions…**

**… however, it’s (very) slowly changing**

# What about EDR / XDR / ATP ?

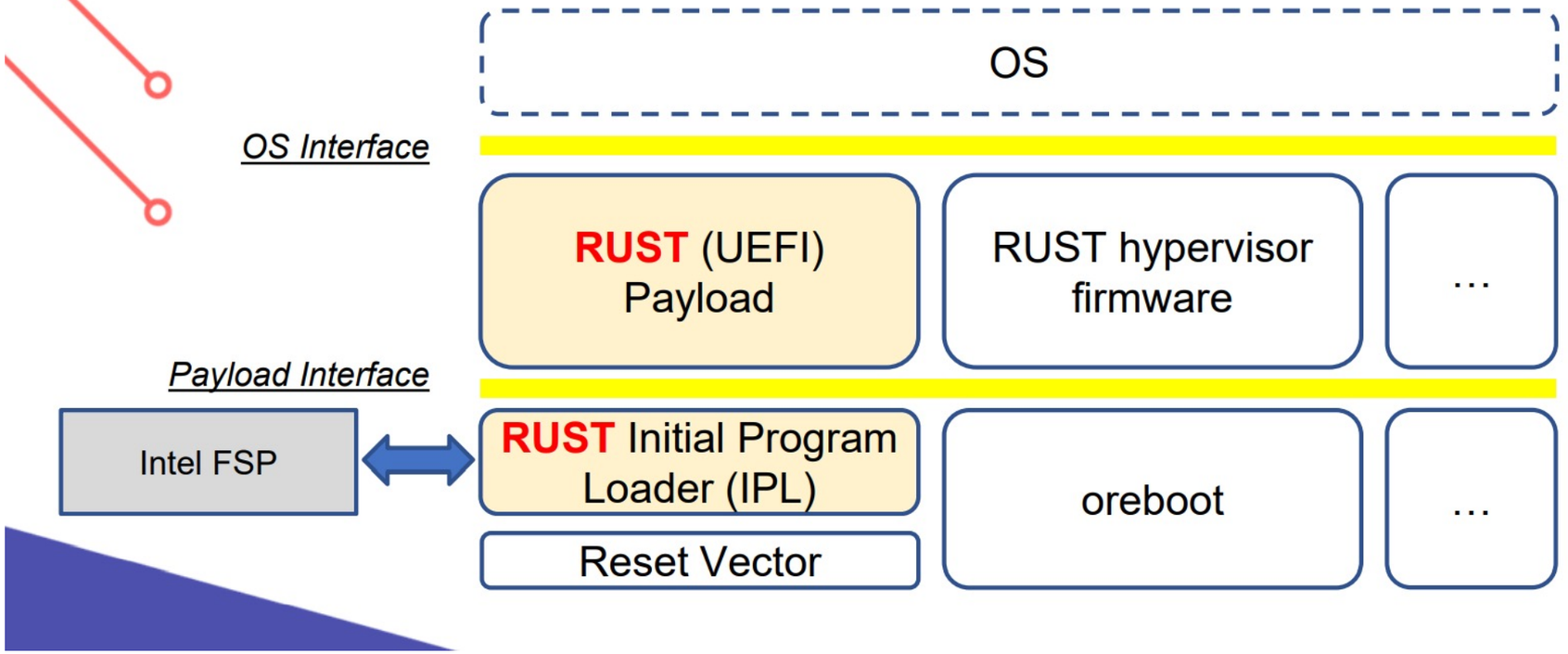
- **Dramatically limited comparing to OS solutions…**

**… however, it’s (very) slowly changing**

# Are we completely lost?

# Using Type Safe languages to Develop Critical Code

## Thought and Current work




OSFC2020_Rust_EFI_Yao_Zimmer_NDK4Dme.pdf

rust-osdev/uefi-rs: Rust wrapper for UEFI

edk2-staging/RustPkg at edkii-rust · tianocore/edk2-staging

# We would like to warmly thank:


- **NVIDIA Product Security, GPU System Software and PSIRT teams**
  - for supporting this research, assistance in coordinating disclosure and more…
- **Intel PSIRT**
  - for hard work on the fixes and active participation and support in this coordinated disclosure
- **RedHat and LKRG project**
  - for helping to discover at scale potentially vulnerable vendors and notify them
- **Dell**
  - for the great collaboration during disclosure process and very professional feedback

# Questions?