EMBEDI
(https://embedi.com)

Categories:

- Analytics (https://embedi.com/blog/categories/analytics/)
- Research (https://embedi.com/blog/categories/research/)

Tags:

#ATM (https://embedi.com/blog/tags/atm/) #CISCO (https://embedi.com/blog/tags/cisco/)

#cybersecurity (https://embedi.com/blog/tags/cybersecurity/)

#D-Link (https://embedi.com/blog/tags/d-link/) #DJI (https://embedi.com/blog/tags/dji/)

#exploitation (https://embedi.com/blog/tags/exploitation/)

#firmware-security (https://embedi.com/blog/tags/firmware-security/)

#hardware (https://embedi.com/blog/tags/hardware/)

#hijacking (https://embedi.com/blog/tags/hijacking/) #intel (https://embedi.com/blog/tags/intel/)

#Microsoft (https://embedi.com/blog/tags/microsoft/) #mobile (https://embedi.com/blog/tags/mobile/)

#Office (https://embedi.com/blog/tags/office/) #RCE (https://embedi.com/blog/tags/rce/)

#router (https://embedi.com/blog/tags/router/) #SCADA (https://embedi.com/blog/tags/scada/)

#vulnerabilities (https://embedi.com/blog/tags/vulnerabilities/)

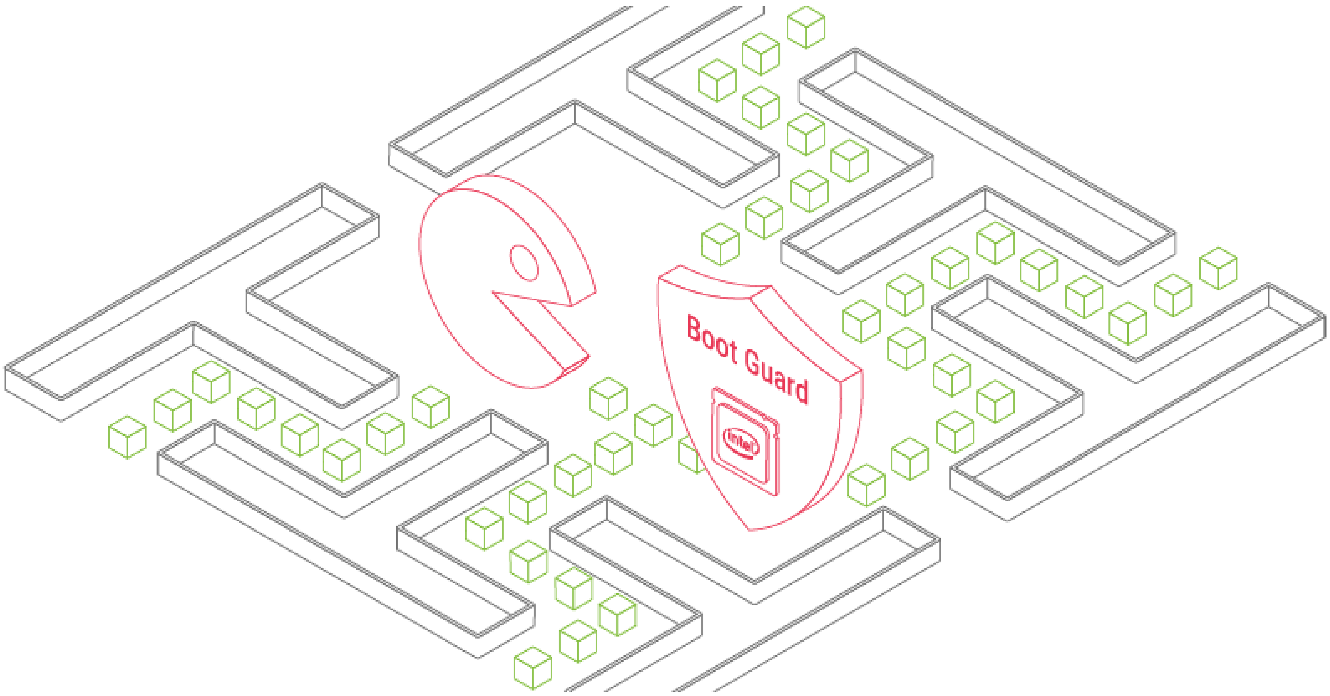
Popular articles:

Killchain of IoT Devices. Part 2 (https://embedi.com/blog/killchain-iot-devices-part-2/)

Killchain of IoT Devices. Part 1 (https://embedi.com/blog/killchain-iot-devices-part-1/)

5 October, 2017

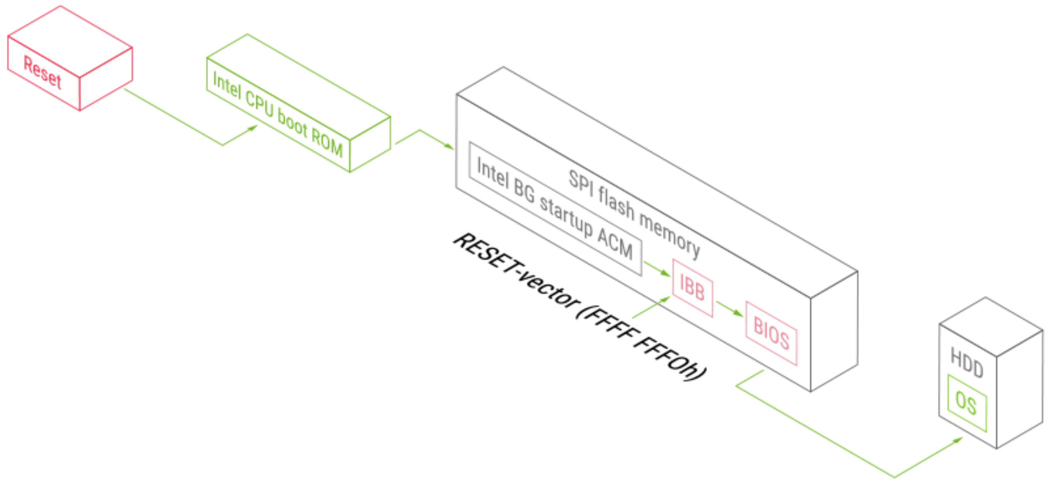
Bypassing Intel Boot Guard



Category: Research (https://embedi.com/blog/categories/research/)
Tags: #intel (https://embedi.com/blog/tags/intel/), #vulnerabilities (https://embedi.com/blog/tags/vulnerabilities/)

In recent years, there is an increasing attention to the UEFI BIOS security. As a result, there are more advanced technologies created to protect UEFI BIOS from illegal modifications. One of such technologies is Intel Boot Guard (BG) – a hardware-assisted BIOS integrity verification mechanism available since Haswell microarchitecture (2013). So-called «UEFI rootkits killer» this technology is designed to create a trusted boot chain (where a current boot component cryptographically measures/verifies the integrity of the next one) with Root-of-Trust locked into hardware.

How is that possible? Let's take a look at the figure:



The first boot component (the Root of Trust) in this chain is Intel CPU with the microcode (located inside the CPU boot ROM). It loads into the protected internal CPU memory called Authenticated Code RAM (ACRAM), verifies and executes an Intel-signed BG startup Authenticated Code Module (ACM). The hash of the RSA public key (which is used to verify the signature of the ACM) is hard-coded inside the CPU. This is an Intel-developed part of the technology implementation. Intel BG ACM acts before the execution of BIOS and is responsible for verifying its initial part called the Initial Boot Block (IBB). Usually, the IBB represents the contents of SEC/PEI volumes of UEFI BIOS.

The IBB must contain the final part of the technology implementation, which is developed by BIOS vendor (or the OEM): the code for verifying the remaining BIOS contents (usually the DXE volumes are the remaining part).

Intel BG technology is designed to be permanently configured (the configuration is to be written to one-time-programmable Intel chipset fuses) by the OEM during the manufacturing process. So turning this technology on makes it almost impossible to modify BIOS without knowing the private part of the OEM Root Key (Intel BG configuration includes the hash of the public part). However, the flexibility of the configuration allows setting Intel Boot Guard vulnerable making this technology easy to bypass in some cases.

Such a strong protection mechanism without any bugs?

Technical details describing how the CPU bootcode starts the BG ACM, and how the BG ACM verifies the IBB prior to allowing the CPU to execute it have been reverse-engineered and originally published here:

Intel Boot Guard research (https://github.com/flothrone/bootguard)

The referenced research also mentions that a few OEMs produced (and some of them are still producing) computer systems with Intel BG configuration not set properly (with chipset fuses left in an undefined state). This brings an amazing opportunity for an attacker with capabilities to inject program code into BIOS: to turn Intel BG technology on manually making any modifications in BIOS permanent. It only requires configuring Intel BG by programming the chipset fuses (via pure software way) after the modification is done.

Our experimental research revealed that the verification of IBB is not performed upon every boot. For example, if the PC, once powered on and is never shut down, so only the Sleep (S3) mode is used, the verification is done only once every 12 times a device is powered up.

However, the most interesting thing here is a set of a few logical mistakes, made by the BIOS vendor and OEMs enabling an attacker to bypass the technology and make any changes to a huge part of BIOS. Alex Matrosov discovered a few vulnerabilities (not only in Intel BG but also in Intel BIOS Guard):

Betraying the BIOS: Where the Guardians of the BIOS are Failing

(https://github.com/REhints/BlackHat_2017)

The following text describes another one that we have discovered.

Killing the killer

To slice and dice it, let’s take a look at the IBB on Gigabyte GA-H170-D3H motherboard with UEFI BIOS version F04. Though it does not have Intel BG enabled, the components of this technology are present: BG ACM, BG key manifests, IBB (SEC/PEI) with a verification routines for DXE. An important notice: the motherboard’s firmware is based on AMI Aptio UEFI BIOS – a very popular product used by many OEMs (for example, Gigabyte, MSI, Asus, Acer, Dell, HP, ASRock). Therefore, the part of code we are going to talk about can be found in any system (produced since 2013) with BIOS based on that codebase (because it is written by AMI).

The present IBB should contain the code for verifying the integrity of the remaining part of BIOS, and it does: the BootGuardPei module (GUID: B41956E1-7CA2-42DB-9562-168389F0F066).

Firstly, the BootGuardPei entrypoint routine checks the compatibility with Intel BG (through the MSRs) and registers the notification callback procedure (Start) for EFI_END_OF_PEI_PHASE_PPI to be called in the end of PEI phase when the operating memory will be available to use.

```
                rdmsr
                and     edx, 1
                xor     eax, eax
                or      eax, edx
                jnz     short loc_FFEB7137

loc_FFEB7133:                           ; CODE XREF: _ModuleEntryPoint+2F↓j
                xor     eax, eax
                jmp     short exit
; ---------------------------------------------------------------------------

loc_FFEB7137:                           ; CODE XREF: _ModuleEntryPoint+1D↑j
                mov     ecx, 13Ah
                rdmsr
                mov     [ebp+var_4], edx
                test    eax, eax
                jz      short loc_FFEB7133
                mov     ecx, 13Ah
                rdmsr
                mov     [ebp+var_4], edx
                test    al, 1
                jz      short loc_FFEB7168
                test    al, 20h
                jz      short loc_FFEB7168
                test    al, 8
                jz      short loc_FFEB7168
                mov     eax, [esi]
                push    offset gUknownPpi ; PpiList
                push    esi             ; PeiServices
                call    [eax+EFI_PEI_SERVICES.InstallPpi]
                pop     ecx
                pop     ecx

loc_FFEB7168:                           ; CODE XREF: _ModuleEntryPoint+3D↑j
                                        ; _ModuleEntryPoint+41↑j ...
                call    GetPeiServices
                mov     ecx, [eax]
                push    offset gEfiPeiEndOfPeiPhasePpi ; NotifyList
                push    eax             ; PeiServices
                call    [ecx+EFI_P
                pop     ecx
                pop     ecx

exit:
                pop     esi
                leave
                retn
_ModuleEntryPoint endp

; =============== S U B R O U T I

; EFI_PEI_NOTIFY_DESCRIPTOR gEfiPeiEndOfPeiPhasePpi
gEfiPeiEndOfPeiPhasePpi EFI_PEI_NOTIFY_DESCRIPTOR <80000020h, \
                                        ; DATA XREF: _ModuleEntryPoint+5B↓o
                                         offset gEfiPeiEndOfPeiPhasePpiGuid, \
                                         offset Start>
gUnknownPpiGuid dd 6EE1B483h            ; Data1 ; DATA XREF: .text:gUknownPpi↓o
                dw 0A9B8h               ; Data2
                dw 4EAFh                ; Data3
                db 9Ah, 0E1h, 3Bh, 28h, 0C5h, 0CFh, 0F3h, 6Bh; Data4
; EFI_PEI_PPI_DESCRIPTOR gUknownPpi
gUknownPpi      EFI_PEI_PPI_DESCRIPTOR <80000010h, offset gUnknownPpiGuid, 0>
                                        ; DATA XREF: _ModuleEntryPoint+49↓o
```

Therefore, the Start routine is called further, which does the following.

1. Check the booting mode.

```
Start           proc near               ; DATA XREF: .text:gEfiPeiEndOfPeiPhasePpi↑o

Block1CalculatedHash= byte ptr -60h
Block0CalculatedHash= byte ptr -40h
BootGuardPeiHobGuid= EFI_GUID ptr -20h
HashContainerBuffer= dword ptr -10h
BootMode        = dword ptr -0Ch
Hob             = dword ptr -8
Sha256Buffer    = dword ptr -4
PeiServices     = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 60h
                push    ebx
                mov     eax, 498Dh
                mov     [ebp+BootGuardPeiHobGuid.Data2], ax
                push    esi
                mov     eax, 429Dh
                push    edi
                mov     [ebp+BootGuardPeiHobGuid.Data1], 0B60AB175h
                mov     [ebp+BootGuardPeiHobGuid.Data3], ax
                mov     [ebp+BootGuardPeiHobGuid.Data4], 0ADh
                mov     [ebp+BootGuardPeiHobGuid.Data4+1], 0BAh
                mov     [ebp+BootGuardPeiHobGuid.Data4+2], 0Ah
                mov     [ebp+BootGuardPeiHobGuid.Data4+3], 62h
                mov     [ebp+BootGuardPeiHobGuid.Data4+4], 2Ch
                mov     [ebp+BootGuardPeiHobGuid.Data4+5], 58h
                mov     [ebp+BootGuardPeiHobGuid.Data4+6], 16h
                mov     [ebp+BootGuardPeiHobGuid.Data4+7], 0E2h
                call    GetPeiServices
                mov     ecx, [eax]
                lea     edx, [ebp+BootMode]
                push    edx             ; BootMode
                push    eax             ; PeiServices
                call    [ecx+EFI_PEI_SERVICES.GetBootMode]
                pop     ecx
                pop     ecx
                test    eax, eax
                jl      exit
                cmp     [ebp+BootMode], BOOT_IN_RECOVERY_MODE
                jz      exit
                cmp     [ebp+BootMode], BOOT_ON_FLASH_UPDATE
                jz      exit
                cmp     [ebp+BootMode], BOOT_ON_S3_RESUME
                jz      exit
```

It is quite obvious that the DXE code will never be verified while booting from the Sleep (S3) mode. However, that is not the point; this aspect is reasonable enough (to minimize the impact on system performance). Let's go ahead.

2. Create HOB with only one byte of data (which will represent the result of verification routine).

```
mov     ebx, [ebp+PeiServices]
mov     eax, [ebx]
lea     ecx, [ebp+Hob]
push    ecx             ; Hob
push    19h             ; Length
push    EFI_HOB_TYPE_GUID_EXTENSION ; Type
push    ebx             ; PeiServices
call    [eax+EFI_PEI_SERVICES.CreateHob]
add     esp, 10h
test    eax, eax
jl      exit
mov     edi, [ebp+Hob]
add     edi, 8
lea     esi, [ebp+BootGuardPeiHobGuid]
movsd
movsd
movsd
movsd
mov     eax, [ebp+Hob]
mov     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 0 ; VerificationResult
```

3. Search the hash container (which holds the hash of DXE code) inside the PEI volumes (by GUID: 389CC6F2-1EA8-467B-AB8A-78E769AE2A15) and verify the first block of DXE pointed by this container.

```
FindHashContainerAndVerifyBlock0:       ; CODE XREF: Start+D0↑j
                lea     eax, [ebp+HashContainerBuffer]
                push    eax
                mov     esi, ebx
                call    FindBootGuardDxeHashContainer
                mov     edi, eax
                pop     ecx
                test    edi, edi
                jl      exit
                push    [ebp+Sha256Buffer]
                mov     esi, [ebp+HashContainerBuffer]
                call    Sha256Init
                push    [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block0Size]
                push    [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block0BaseAddress]
                push    [ebp+Sha256Buffer]
                call    Sha256Calc
                lea     eax, [ebp+Block0CalculatedHash]
                push    eax
                push    [ebp+Sha256Buffer]
                call    Sha256Calc2
                lea     eax, [ebp+Block0CalculatedHash]
                add     esp, 18h
                cmp     esi, eax
                jz      short VerifyBlock1
                push    SHA256_DIGEST_SIZE
                push    eax
                push    esi
                call    CompareMemory
                add     esp, 0Ch
                test    eax, eax
                jnz     short ReturnError
```

The hash container format is as follows:

```
00000000 BOOT_GUARD_DXE_HASH_CONTAINER struc ; (sizeof=0x50, mappedto_15)
00000000                                 ; XREF: AllocateBufferForHashContainer+F/o
00000000 Block0Sha256Hash db 32 dup(?)
00000020 Block0BaseAddress dd ?          ; XREF: Start+FD/r
00000024 Block0Size      dd ?            ; XREF: Start+FA/r
00000028 Block1Sha256Hash db 32 dup(?)   ; XREF: Start+163/o
00000048 Block1BaseAddress dd ?          ; XREF: Start+135/r Start+14C/r
0000004C Block1Size      dd ?            ; XREF: Start+13B/r Start+149/r
00000050 BOOT_GUARD_DXE_HASH_CONTAINER ends
```

4. After that verify the second block (if it is described by the hash container like the first one).

```
VerifyBlock1:                           ; CODE XREF: Start+11C↑j
                mov     eax, [ebp+Hob]
                mov     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 1 ; VerificationResult
                cmp     [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block1BaseAddress], 0
                jz      short ReturnOkGoOn
                cmp     [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block1Size], 0
                jz      short ReturnOkGoOn
                push    [ebp+Sha256Buffer]
                call    Sha256Init
                push    [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block1Size]
                push    [esi+BOOT_GUARD_DXE_HASH_CONTAINER.Block1BaseAddress]
                push    [ebp+Sha256Buffer]
                call    Sha256Calc
                lea     eax, [ebp+Block1CalculatedHash]
                push    eax
                push    [ebp+Sha256Buffer]
                call    Sha256Calc2
                add     esi, BOOT_GUARD_DXE_HASH_CONTAINER.Block1Sha256Hash
                lea     eax, [ebp+Block1CalculatedHash]
                add     esp, 18h
                cmp     esi, eax
                jz      short ReturnOk
                push    SHA256_DIGEST_SIZE
                push    eax
                push    esi
                call    CompareMemory
                add     esp, 0Ch
                test    eax, eax
                jnz     short ReturnError

ReturnOk:                               ; CODE XREF: Start+16E↑j
                mov     eax, [ebp+Hob]
                mov     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 1 ; VerificationResult

ReturnOkGoOn:                           ; CODE XREF: Start+139↑j
                                        ; Start+13F↑j
                mov     eax, edi
                jmp     short exit
; ---------------------------------------------------------------------------

ReturnError:                            ; CODE XREF: Start+12C↑j
                                        ; Start+17E↑j
                mov     eax, [ebp+Hob]
                mov     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 0 ; VerificationResult
                xor     eax, eax

exit:                                   ; CODE XREF: Start+55↑j
                                        ; Start+5F↑j ...
                pop     edi
                pop     esi
                pop     ebx
                leave
                retn
```

In case the verification was not successful (for example, because the DXE code was illegally modified), the zero value will be written into HOB. If the DXE were not modified, the positive value (1) would be written. So, the illegal modification of the DXE code will be detected by the BootGuardPei (the hash of the DXE code would not fit one in the hash container).

Good! However, the funny thing is, instead of immediately applying the Intel BG enforcement policy (usually it is an immediate shutdown), this module still allows further execution of BIOS, hence running the modified DXE code. Why?

Inside the DXE volume there is another BG-related module: the BootGuardDxe (GUID: 1DB43EC9-DF5F-4CF5-AAF0-0E85DB4E149A). It is responsible for analyzing the results (previously saved in HOB by BootGuardPei) and shutting down the system if the DXE code does not pass the integrity check (if HOB contains the zero value, as we have already discovered).

Here it is entrypoint routine:

```
                public start
start           proc near               ; DATA XREF: HEADER:00000000000000E0↑o

arg_0           = qword ptr  8

                mov     [rsp+arg_0], rbx
                push    rdi
                sub     rsp, 20h
                mov     rax, [rdx+EFI_SYSTEM_TABLE.BootServices]
                mov     rdi, rcx
                mov     cs:gImageHandle, rcx
                mov     cs:gEfiBootServices, rax
                mov     rax, [rdx+EFI_SYSTEM_TABLE.RuntimeServices]
                mov     cs:gEfiSystemTable, rdx
                mov     rbx, rdx
                lea     rcx, gDxeServicesTableGuid
                lea     rdx, gDxeServicesTable
                mov     cs:gEfiRuntimeServices, rax
                call    FindObjectByGuid
                lea     rdx, gHobList
                lea     rcx, gHobListGuid
                call    FindObjectByGuid
                mov     rdx, rbx
                mov     rcx, rdi
                call    BootGuardCheck
                mov     rbx, [rsp+28h+arg_0]
                add     rsp, 20h
                pop     rdi
                retn
start           endp
```

It calls the BootGuardCheck routine, which gains access to the HOB list, then searches for BootGuardPeiHob.

```
BootGuardCheck  proc near               ; CODE XREF: start+60↑p

var_28          = qword ptr -28h
BootGuardPeiHobGuid= EFI_GUID ptr -18h
buffer          = qword ptr  18h
Registration    = qword ptr  20h

                push    rbx
                sub     rsp, 40h
                mov     eax, 498Dh
                mov     [rsp+48h+BootGuardPeiHobGuid.Data1], 0B60AB175h
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4], 0ADh ; '¡'
                mov     [rsp+48h+BootGuardPeiHobGuid.Data2], ax
                mov     eax, 429Dh
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+1], 0BAh ; '¦'
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+2], 0Ah
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+3], 62h ; 'b'
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+4], 2Ch ; ','
                mov     [rsp+48h+BootGuardPeiHobGuid.Data3], ax
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+5], 58h ; 'X'
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+6], 16h
                mov     [rsp+48h+BootGuardPeiHobGuid.Data4+7], 0E2h ; 'G'
                mov     ecx, 13Ah
                rdmsr
                shl     rdx, 20h
                mov     rcx, 100000000h
                or      rax, rdx
                and     rax, rcx
                jz      ReturnOk
                lea     rdx, [rsp+48h+buffer]
                lea     rcx, gHobListGuid
                call    FindObjectByGuid
                mov     rdx, [rsp+48h+buffer]
                cmp     [rdx+EFI_HOB_GENERIC_HEADER.HobType], 1
                jnz     ReturnOk
                cmp     [rdx+EFI_HOB_HANDOFF_INFO_TABLE.BootMode], 20h ; ' '
                jz      ReturnOk
                cmp     [rdx+EFI_HOB_HANDOFF_INFO_TABLE.BootMode], 12h
                jz      ReturnOk
                mov     rax, cs:gHobList
                mov     rdx, qword ptr [rsp+48h+BootGuardPeiHobGuid.Data4]
                mov     r8, qword ptr [rsp+48h+BootGuardPeiHobGuid.Data1]
                xor     ebx, ebx
                mov     r9d, 0FFFFh
                jmp     short loc_457
```

```
loc_44A:                                ; CODE XREF: BootGuardCheck+CE↓j
                cmp     cx, 4
                jz      short loc_463

loc_450:                                ; CODE XREF: BootGuardCheck+DC↓j
                                        ; BootGuardCheck+E2↓j
                movzx   ecx, [rax+EFI_HOB_GENERIC_HEADER.HobLength]
                add     rax, rcx

loc_457:                                ; CODE XREF: BootGuardCheck+B8↑j
                movzx   ecx, [rax+EFI_HOB_GENERIC_HEADER.HobType]
                cmp     cx, r9w
                jnz     short loc_44A
                mov     rax, rbx

loc_463:                                ; CODE XREF: BootGuardCheck+BE↑j
                cmp     rax, rbx
                jz      short ReturnError
                cmp     r8, qword ptr [rax+EFI_HOB_GUID_TYPE.Name.Data1]
                jnz     short loc_450
                cmp     rdx, qword ptr [rax+EFI_HOB_GUID_TYPE.Name.Data4]
                jnz     short loc_450
                cmp     rax, rbx
                jz      short ReturnError
                cmp     [rax+(size EFI_HOB_GUID_TYPE)], bl ; VerificationResult
                jnz     short exit
                lea     rax, [rsp+48h+buffer]
                xor     r9d, r9d        ; NotifyContext
                lea     r8, PrintFailMessageAndResetSystem ; NotifyFunction
                mov     [rsp+48h+var_28], rax
                mov     rax, cs:gEfiBootServices
                lea     edx, [r9+8]     ; NotifyTpl
                mov     ecx, 200h       ; Type
                call    [rax+EFI_BOOT_SERVICES.CreateEvent]
                cmp     rax, rbx
                jl      short exit
                mov     rax, cs:gEfiBootServices
                mov     rdx, [rsp+48h+buffer] ; Event
                lea     r8, [rsp+48h+Registration] ; Registration
                lea     rcx, gBdsAllDriversConnectedProtocolGuid ; Protocol
                call    [rax+EFI_BOOT_SERVICES.RegisterProtocolNotify]
                jmp     short exit
; ---------------------------------------------------------------------------

ReturnError:                            ; CODE XREF: BootGuardCheck+D6↑j
                                        ; BootGuardCheck+E7↑j
                mov     rbx, 800000000000000Eh
                jmp     short exit
; ---------------------------------------------------------------------------

ReturnOk:                               ; CODE XREF: BootGuardCheck+65↑j
                                        ; BootGuardCheck+85↑j ...
                xor     ebx, ebx

exit:                                   ; CODE XREF: BootGuardCheck+EC↑j
                                        ; BootGuardCheck+118↑j ...
                mov     rax, rbx
                add     rsp, 40h
                pop     rbx
                retn
BootGuardCheck  endp
```

By the way, if BootGuardDxe fails to find it, this routine does nothing but returns an error ☺ Therefore, any physical memory remote access attack (for example, via IEEE 1394 FireWire) would allow to delete this HOB from the list and allow bypassing the DXE code integrity check.

However, if there was found a HOB (created by BootGuardPei) which holds the zero value, the PrintFailMessageAndResetSystem will be called.

```
; void __cdecl PrintFailMessageAndResetSystem(EFI_EVENT Event, void *Context)
PrintFailMessageAndResetSystem proc near
                                        ; DATA XREF: BootGuardCheck+F6↑o

arg_10          = byte ptr  18h
arg_18          = qword ptr  20h

                push    rbx
                sub     rsp, 20h
                mov     rax, cs:gEfiSystemTable
                mov     rbx, rcx
                mov     rdx, [rax+EFI_SYSTEM_TABLE.ConOut]
                mov     rcx, rdx
                call    qword ptr [rdx+30h]
                mov     r11, cs:gEfiSystemTable
                lea     rdx, aBootGuardVerif ; "Boot Guard verified DXE that is fail\n\"
                mov     rax, [r11+EFI_SYSTEM_TABLE.ConOut]
                mov     rcx, rax
                call    qword ptr [rax+8]
                mov     r11, cs:gEfiSystemTable
                lea     rdx, aSystemWillShut ; "System will shutdown\n\r"
                mov     rax, [r11+EFI_SYSTEM_TABLE.ConOut]
                mov     rcx, rax
                call    qword ptr [rax+8]
                mov     r11, cs:gEfiSystemTable
                lea     rdx, aPressAnyKey ; "Press any key\n\r"
                mov     rax, [r11+EFI_SYSTEM_TABLE.ConOut]
                mov     rcx, rax
                call    qword ptr [rax+8]

GetChar:                                ; CODE XREF: PrintFailMessageAndResetSystem+7C↓j
                mov     rax, cs:gEfiSystemTable
                lea     rdx, [rsp+28h+arg_10]
                mov     r8, [rax+EFI_SYSTEM_TABLE.ConIn]
                mov     rcx, r8
                call    qword ptr [r8+8]
                test    rax, rax
                jnz     short GetChar
                lea     ecx, [rax+2]    ; ResetType
                mov     rax, cs:gEfiRuntimeServices
                xor     r9d, r9d        ; ResetData
                xor     r8d, r8d        ; DataSize
                xor     edx, edx        ; ResetStatus
                call    [rax+EFI_RUNTIME_SERVICES.ResetSystem]
                mov     [rsp+28h+arg_18], 0

loc_580:                                ; CODE XREF: PrintFailMessageAndResetSystem+A4↓j
                mov     rax, [rsp+28h+arg_18]
                test    rax, rax
                jz      short loc_580
                mov     rax, cs:gEfiBootServices
                mov     rcx, rbx        ; Event
                call    [rax+EFI_BOOT_SERVICES.CloseEvent]
                add     rsp, 20h
                pop     rbx
                retn
PrintFailMessageAndResetSystem endp
```

You might have already guessed, that if an attacker manages to delete the BootGuardDxe from the DXE volume, the protection of the DXE part will not work at all (there will be no code to check the results of the verification done by the IBB).

This vulnerability was experimentally confirmed on the Dell XPS 13 9350 system with UEFI BIOS versions 1.4.4 – 1.4.10 that has the turned on Intel Boot Guard technology. As for Dell systems, this vulnerability is not dangerous as long as these systems have another BIOS protection mechanisms applied (PRx, SMM protection, verifying the integrity of UEFI BIOS updates, etc.). Therefore, for these systems the exploitation requires a hardware SPI flash programmer or an RCE/LPE vulnerability in SMM code (allowing to bypass the applied protections), to modify the BIOS.

However, there are still many systems having no additional protection mechanisms turned on which makes this vulnerability more dangerous.

Dell's vulnerability research team confirmed this security issue on their systems after some checking. We would like to thank them for the fruitful cooperation that made it possible to enhance the security level on Dell systems. The adequate and professional reaction to the reported security issues is still an incredibly rare commodity when it comes to computer system OEMs updating.

After some cooperation with us, Dell has patched out the vulnerability in the following way (found in UEFI BIOS version 1.4.18). The BootGuardPei module, if the verification fails just brings the system into the Recovery mode, so the DXE phase will not be executed.

```
ReturnError:                            ; CODE XREF: Start+12C↑j
                                        ; Start+17E↑j
                mov     eax, [ebp+BootGuardPeiHob]
                mov     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 0 ; VerificationResult
                call    GetPeiServices
                mov     ecx, [eax]
                push    BOOT_IN_RECOVERY_MODE ; BootMode
                push    eax             ; PeiServices
                call    [ecx+EFI_PEI_SERVICES.SetBootMode]
                pop     ecx
                pop     ecx
                call    InstallUnknown1Ppi
                xor     eax, eax

exit:                                   ; CODE XREF: Start+55↑j
                                        ; Start+5F↑j ...
                pop     edi
                pop     esi
                pop     ebx
                leave
                retn
Start           endp
```

Of course, we had notified **AMI** about this issue and learned that they had already mitigated this vulnerability and notified their customers (which are the OEMs) on how to mitigate the same. Also, AMI assured us that the latest AMI BIOS codebase available for their customers does not have this vulnerability.
This allows OEMs to release BIOS updates (fixing the vulnerability) for the end-users.

However, how exactly did they patched this issue? Let's take a look at the latest Gigabyte GA-H170-D3H motherboard's BIOS.

You better be kidding me…

That is how:

```
ReturnError:                            ; CODE XREF: Start+128↑j
                                        ; Start+173↑j
                and     byte ptr [eax+(size EFI_HOB_GUID_TYPE)], 10h ; VerificationResult
                xor     eax, eax

exit:                                   ; CODE XREF: Start+55↑j
                                        ; Start+5F↑j ...
                pop     edi
                pop     esi
                pop     ebx
                leave
                retn
Start           endp
```

In case of unsuccessful verification, they put the positive value inside the BootGuardPei HOB, allowing to bypass Intel BG protection for the DXE volumes without the need to delete the BootGuardDxe module because it resets the system only if the HOB stores the zero!

No protection mechanism works – no troubles with fixing bugs in it!

(https://www.linkedin.com/shareArticle?mini=true&url=https://embedi.com/blog/bypassing-intel-boot-guard/&title=Bypassing Intel Boot Guard)

(http://www.twitter.com/intent/tweet?url=https://embedi.com/blog/bypassing-intel-boot-guard/&text=Bypassing Intel Boot Guard)

() 86 12

Subscribe to our newsletter to stay in touch

Enter your email here Submit

Solutions (https://embedi.com/solutions/) Blog (https://embedi.com/blog/)
Our news (https://embedi.com/news/) Resources (https://embedi.com/resources/)
About (https://embedi.com/about/)

(https://www.facebook.com/Embedi/) (https://www.linkedin.com/company/embedi)
(https://twitter.com/_embedi_) (https://github.com/embedi/)
(https://www.youtube.com/channel/UC9XR2noZynNxvQcg0G9ooKA)