Untrusted Roots: exploiting vulnerabilities in Intel ACMs

Alexander Ermolov
@flothrone

Dmitry Frolov
@allowitsme

#WhoAmI

- Alexander Ermolov
- ZeroNights security conference

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

#WhoAmI

Dmitry Frolov - security researcher at Digital Security

Has a major experience in UEFI firmware researching and vulnerability hunting (over 15 CVEs for Intel systems).

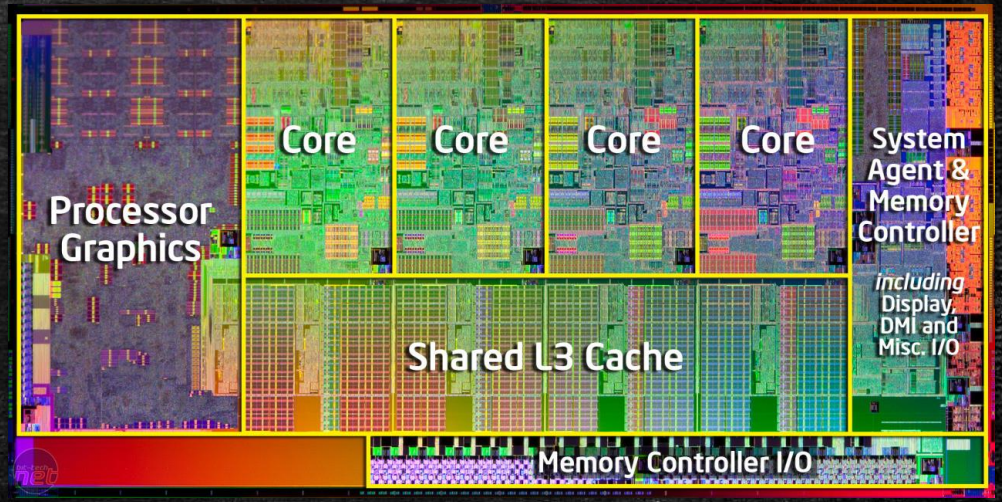
#Agenda

- Intel CPU Roots of Trust overview
  - Intel Microcode
  - FIT
  - Intel ACMs
- Intel BIOS Guard ACM
- Takeaways

Intel CPU Roots of Trust overview

Inside Intel CPU

- Processor cores
  - BSP (Bootstrap Processor)
  - APs (Application Processors)
- Graphics core
- IMC (Integrated Memory Controller)
- L3 cache
- I/O logic



Microcode

Control Unit has Microcode ROM that contains the CPU microcode - a program written in hardware-level instructions to implement higher-level instructions

For example, MOVS instruction implementation:

```
LLDF                    ; load direction flag to latch in functional unit
OR ecx, ecx             ; test if ECX is zero
JZ end                  ; terminate string move if ECX is zero

loop:
          MOVFM tmp0, [esi]    ; move the data to tmp data from source and inc/dec ESI
          MOVIM [edi], tmp0    ; move the data to destination and inc/dec EDI
          EDECXJNZ loop        ; dec ECX and repeat until zero
end:
          EXIT
```

Security Analysis of x86 Processor Microcode
https://www.dcddcc.com/docs/2014_paper_microcode.pdf

Microcode

Dumped example of CPUID instruction implementation:

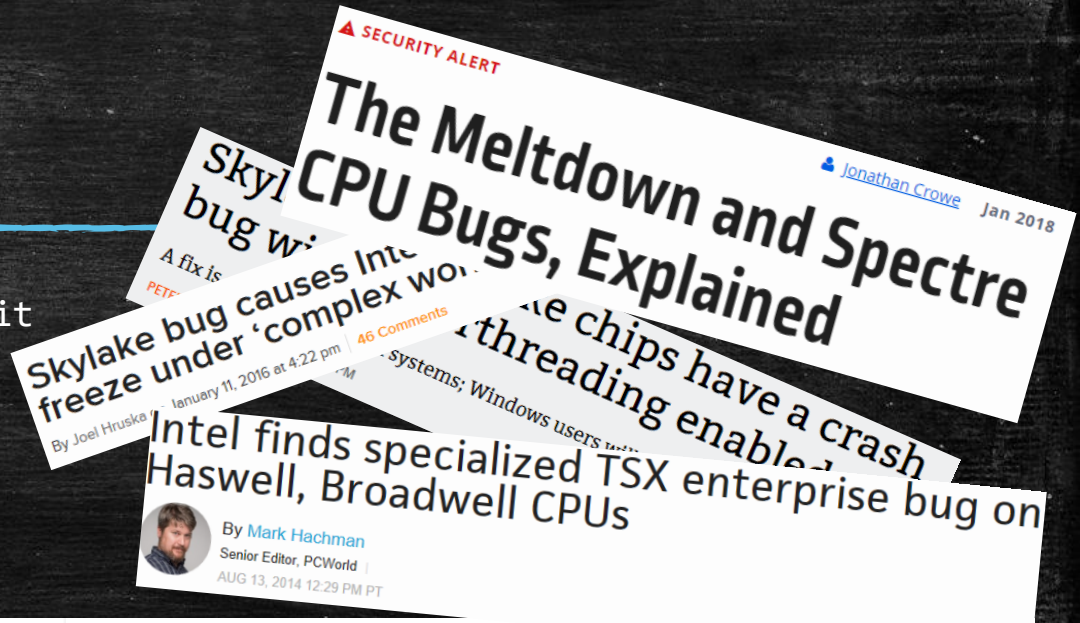
```
4760: 00050003b800      tmp11:= SUB_DSZ32(0x00000000, rax)
4761: 0fef02000000      GENARITHFLAGS(0x00000002)
4762: 100a80000200      MERGEFLGS(0x00000080)
------------------------------2-4e80-00-09------------------------------

4764: 100a40800200      MERGEFLGS(0x00000040)
------------------------------0-4776-00-06------------------------------
4765: 006311031200      tmp1:= READURAM(0x0011, 64)
4766: 0e6510072371      tmp2:= LDPPHYSTICKLE_DSZ64_ASZ64_SC1(tmp1, 0x00000510, mode=01)

4768: 006520031232      tmp1:= SHR_DSZ64(tmp2, 0x00000020)
4769: 00161f03323b      tmp3:= BTR_DSZ32(tmp11, 0x0000001f)
476a: 013e00032cf2      tmp2:= SUB1AND_DSZ32(tmp2, tmp3)
------------------------------3-0000-00-03------------------------------

476c: 003700032c72      tmp2:= CMOVCC_DSZ32_CONDNB(tmp2, tmp1)
476d: 00251f031232      tmp1:= SHR_DSZ32(tmp2, 0x0000001f)
476e: 002401034cc8      tmp4:= SHL_DSZ32(0x00000001, tmp3)
------------------------------3-0000-00-03------------------------------

4770: 000400034cb4      tmp4:= AND_DSZ32(tmp4, tmp2)
4771: 00071f032cc8      tmp2:= NOTAND_DSZ32(0x0000001f, tmp3)
4772: 013e00032cb1      tmp2:= SUB1AND_DSZ32(tmp1, tmp2)
------------------------------3-0000-00-03------------------------------

4774: 013500032d32      tmp2:= CMOVCC_DSZ32_CONDNZ(tmp2, tmp4)
4775: 2929903c0032      CMPUJNZ_DIRECT_NOTTAKEN(tmp2, 0x00000000, 0x1f90)
```

Undocumented behavour of CPUID instruction
https://twitter.com/_markel___/status/1324848756915920898?s=20

Microcode update

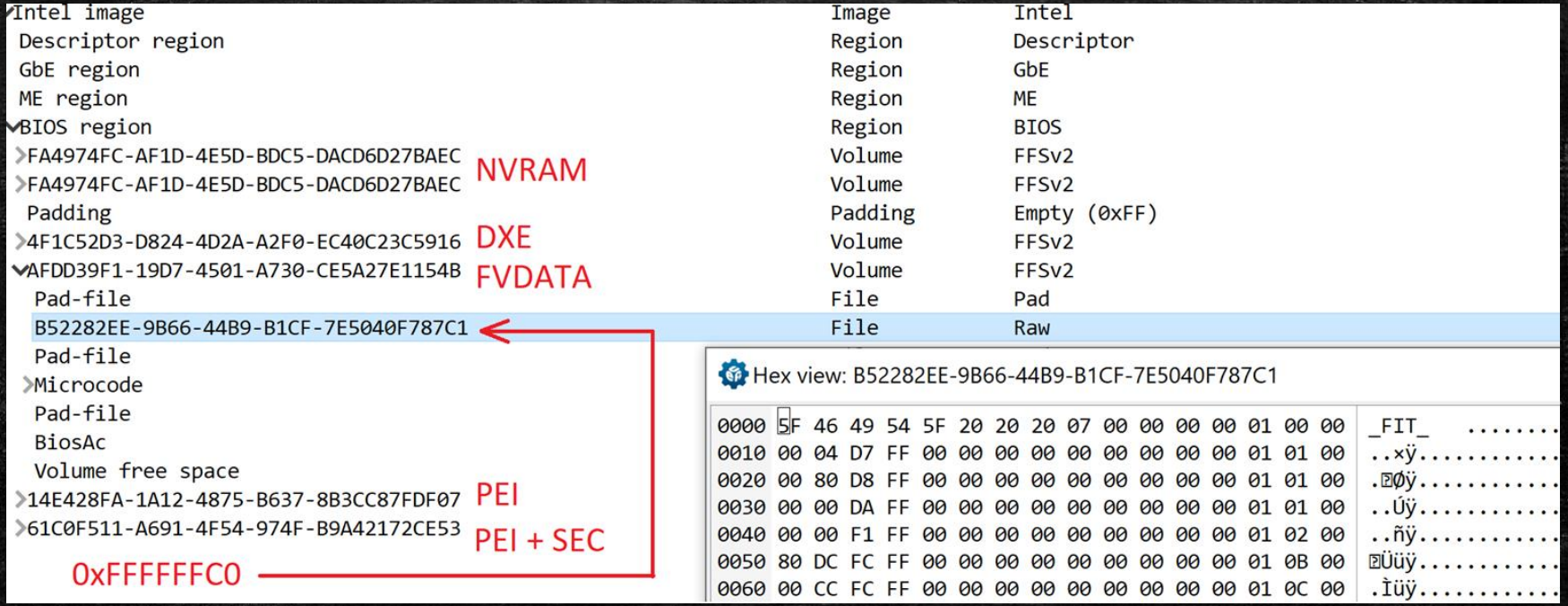
Microcode can have bugs, so it should be updatable

The updated microcode has to be loaded into Control Store upon each CPU power on



	Address	Size	Version	Checksum	Type	
1	_FIT_	00000070h	0100h	00h	FIT Header	
2	00000000FFD70400h	00017C00h	0100h	00h	Microcode	CPUID: 000906EAh, Revision: 00000096h, Date: 02.05.2018
3	00000000FFD88000h	00018000h	0100h	00h	Microcode	CPUID: 000906EBh, Revision: 0000008Eh, Date: 24.03.2018
4	00000000FFDA0000h	00017800h	0100h	00h	Microcode	CPUID: 000906ECh, Revision: 00000084h, Date: 19.02.2018
5	00000000FFF10000h	00008000h	0100h	00h	BIOS ACM	LocalOffset: 00000018h, EntryPoint: 00003BD1h, ACM SVN: 0000h, Date: 09.02.2017
6	00000000FFFCDC80h	00000000h	0100h	00h	BootGuard Key Manifest	
7	00000000FFFCCC00h	00000000h	0100h	00h	BootGuard Boot Policy	

Firmware Interface Table (FIT)



Firmware Interface Table (FIT)

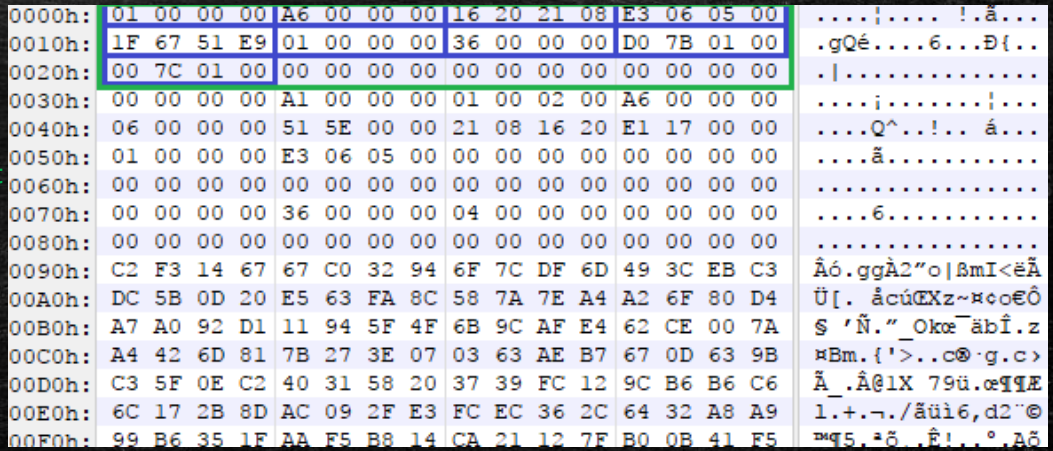
- Is a required element for Intel 64 architecture since introduction of Boot Guard technology
- Can point to microcode update (MCU) binaries
- CPU can load microcode updates from FIT prior to execution 0f BIOS and before starting Intel Boot Guard

5	00000000FFDD6EB0h	00012C00h	0100h	00h	Microcode	CPUID: 000506E2h, Revision: 0000002Ch, Date: 01.07.2015
6	00000000FFDE9AB0h	00017800h	0100h	00h	Microcode	CPUID: 000806E9h, Revision: 00000042h, Date: 02.10.2016
7	00000000FFE012B0h	00017800h	0100h	00h	Microcode	CPUID: 000906E9h, Revision: 00000042h, Date: 02.10.2016
8	00000000FFE18AB0h	00017800h	0100h	00h	Microcode	CPUID: 000506E8h, Revision: 00000034h, Date: 10.07.2016
9	00000000FFFE0000h	00008000h	0100h	00h	BIOS ACM	LocalOffset: 00000018h, EntryPoint: 00003BB1h, ACM SVN: 0002h, Date: 07.02.2016
10	00000000FFFD4C80h	00000241h	0100h	00h	BootGuard Key Manifest	LocalOffset: 00000018h, KM Version: 10h, KM SVN: 00h, KM ID: 0Fh
11	00000000FFFD3C00h	000002DFh	0100h	00h	BootGuard Boot Policy	LocalOffset: 00000018h, BP SVN: 00h, ACM SVN: 02h

Microcode Update binary main header

Microcode Update binary starts with the main header followed by an extended header and update data

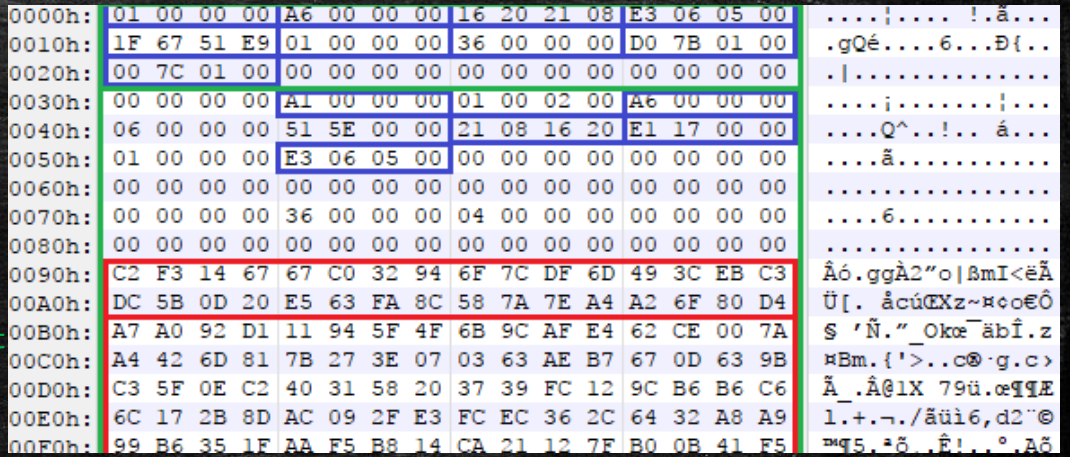
```
typedef struct MICROCODE_UPDATE_HEADER {
  unsigned long header_version;      // 1
  unsigned long update_revision;
  unsigned long date;                // BCD format
  unsigned long processor_signature; // CPUID
  unsigned long checksum;
  unsinged long loader_revision;
  unsinged long processor_flags;
  unsigned long data_size;           // in bytes
  unsigned long total_size;          // in bytes
  unsigned char reserved[0x0C];
};
```



0000h:	01	00	00	00	A6	00	00	00	16	20	21	08	E3	06	05	00
0010h:	1F	67	51	E9	01	00	00	00	36	00	00	00	D0	7B	01	00
0020h:	00	7C	01	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:	00	00	00	00	A1	00	00	00	01	00	02	00	A6	00	00	00
0040h:	06	00	00	00	51	5E	00	00	21	08	16	20	E1	17	00	00
0050h:	01	00	00	00	E3	06	05	00	00	00	00	00	00	00	00	00
0060h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070h:	00	00	00	00	36	00	00	00	04	00	00	00	00	00	00	00
0080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0090h:	C2	F3	14	67	67	C0	32	94	6F	7C	DF	6D	49	3C	EB	C3
00A0h:	DC	5B	0D	20	E5	63	FA	8C	58	7A	7E	A4	A2	6F	80	D4
00B0h:	A7	A0	92	D1	11	94	5F	4F	6B	9C	AF	E4	62	CE	00	7A
00C0h:	A4	42	6D	81	7B	27	3E	07	03	63	AE	B7	67	0D	63	9B
00D0h:	C3	5F	0E	C2	40	31	58	20	37	39	FC	12	9C	B6	B6	C6
00E0h:	6C	17	2B	8D	AC	09	2F	E3	FC	EC	36	2C	64	32	A8	A9
00F0h:	99	B6	35	1F	AA	F5	B8	14	CA	21	12	7F	B0	0B	41	F5

Microcode Update binary extended header

```
typedef struct MICROCODE_UPDATE_EXTENDED_HEADER {
  unsigned short module_type;          // 0
  unsigned short module_subtype;       // 0
  unsigned long  header_size;          // in dwords
  unsigned long  header_version;       // 0x20001
  unsigned long  update_revision;
  unsigned long  unknown[2];
  unsigned long  date;                 // BCD format
  unsigned long  update_size;          // in dwords
  unsigned long  svn;
  unsigned long  processor_signature;
  unsigned long  unknown2[0x0E];
  unsigned char  update_hash[0x20];    // SHA256 hash of the decrypted update data
  unsigned char  rsa_mod[0x100];       // RSA 2048 public key modulus
  unsigned long  rsa_exp;              // RSA 2048 public key exponent
  unsigned char  signature[0x100];     // RSA 2048 signature of the header
};
```



Microcode Update binary data

- The main part in MCU binary is Data (encrypted, the decryption key is hardcoded into CPU)
- Hash of RSA public key to authenticate the MCU is also hardcoded into CPU
- So no one knows exactly what Microcode is capable of
- Recently it was dumped decrypted, however format is yet to be researched
  - https://github.com/chip-red-pill
  - by Dmitry Sklyarov (@_Dmit), Mark Ermolov (@_markel___), Maxim Goryachy (@h0t)

Known facts about Microcode

- Implements instructions
- Configures the execution logic on the line (that’s how side-channels are fixed)
- Implements some startup behavior (like FIT parsing)
- Loads MCU from FIT
- Loads and executes Intel Authenticated Code Modules (ACMs) (from FIT or not)

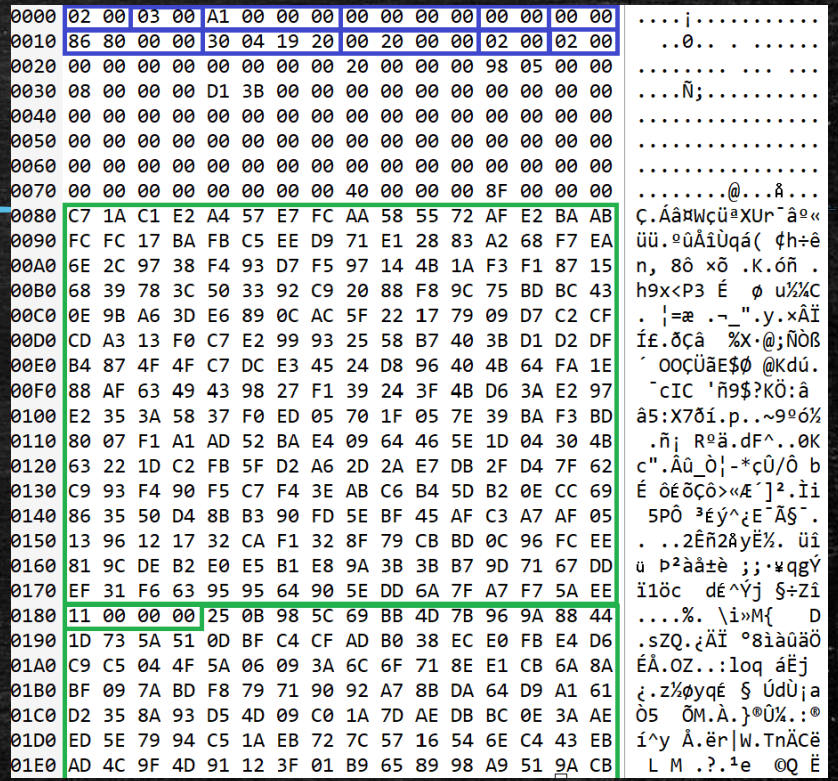
Authenticated Code Modules (ACMs)

- Signed and sometimes encrypted Intel code modules
- Loaded and executed from L3 cache (sometimes called AC RAM)
- Serve as a Root-of-Trusts and a core of implementation for technologies:
  - Intel Boot Guard
    type/subtype: 2.3, not encrypted, signed with KEY2
  - Intel Trusted Execution Technology (TXT)
    type/subtype: 2.0, not encrypted, signed with KEY3
  - Intel BIOS Guard (PFAT)
    type/subtype: 1.1, encrypted, signed with KEY1

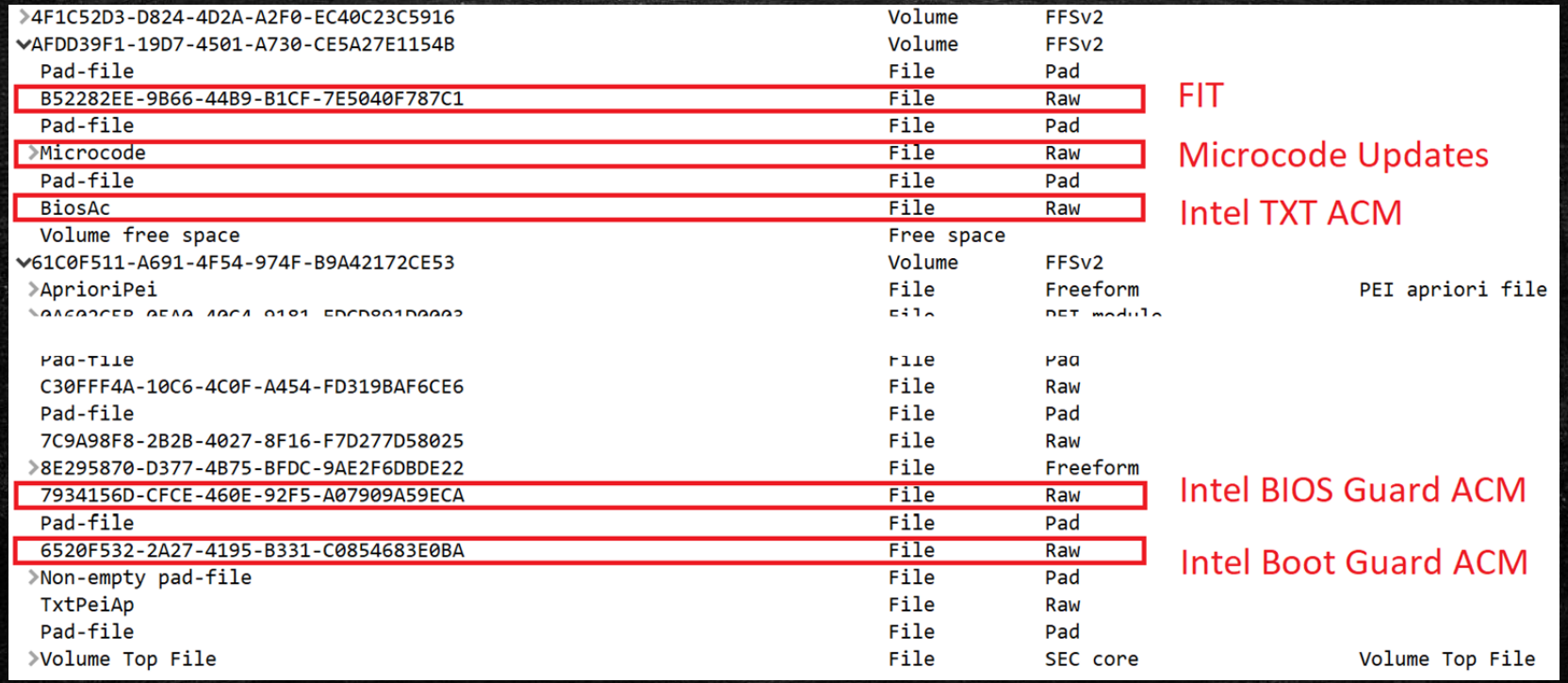
  FYI: microcode update binary
    type/subtype: 1.0, encrypted, signed with KEY1

Intel ACM header

```
typedef struct ACM_HEADER {
  unsigned short module_type;
  unsigned short module_subtype;
  unsigned long  header_size;          // in dwords
  unsigned long  header_version;
  unsigned short chipset_id;
  unsigned short flags;
  unsigned long  module_vendor;        // 0x8086
  unsigned long  date;                 // BCD format
  unsigned long  module_size;          // in dwords
  unsigned short acm_svn;
  unsigned short se_svn;
  …
  unsigned char  rsa_mod[0x100];       // RSA 2048 public key modulus
  unsigned long  rsa_exp;              // RSA 2048 public key exponent
  unsigned char  signature[0x100];     // RSA 2048 signature of the header + data
};
```



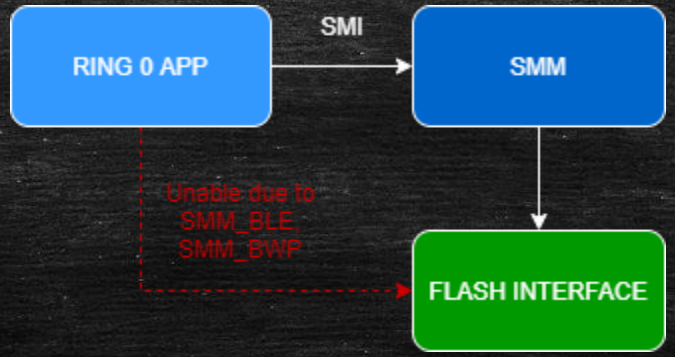
CPU Roots of Trust inside UEFI BIOS



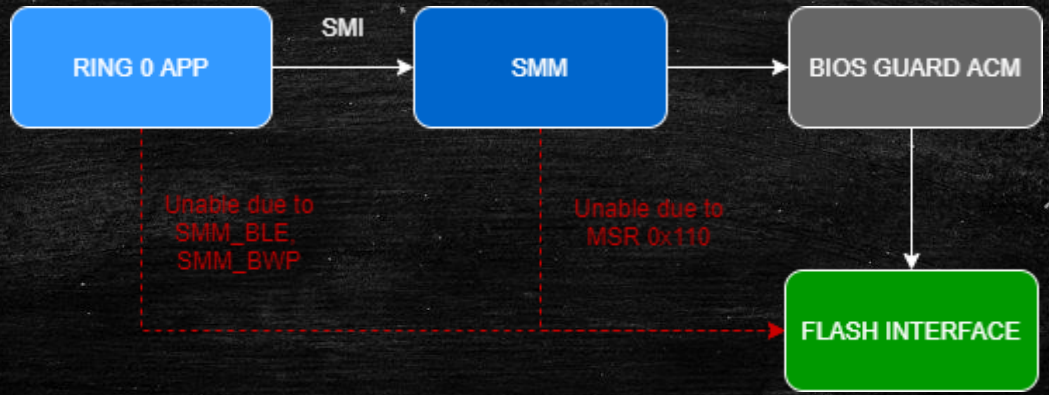
Intel BIOS Guard

Why Intel BIOS Guard (BiG)?

BIOS Guard disabled



BIOS Guard enabled



Intel BIOS Guard presence indications

- 0xCE PLATFORM_INFO_MSR:
  Bit 35: BIOS Guard supported bit

PLATFORM_INFO	0xCE	08080838F1012400

- 0x110 MSR_PLAT_FRMW_PLOT_CTRL (BIOS_GUARD_CTRL_MSR):
  Bit 0: Lock bit
  Bit 1: BIOS Guard enable/disable bit

BIOS_GUARD_CTRL	0x110	0000000000000001	BIOS_GUARD_CTRL	0x110	0000000000000003

This is the only BiG-related MSR accessible from kernel mode. The others are only for SMM, and any attempt to read them from OS will cause MCE (Machine Check Exception).

Intel BIOS Guard presence indications

- The most of BiG support logic implemented in BiosGuardServices {6D4BAA0B-F431-4370-AF19-99D6209239F6}

  shared between all IBVs (AMI, Phoenix, Insyde)

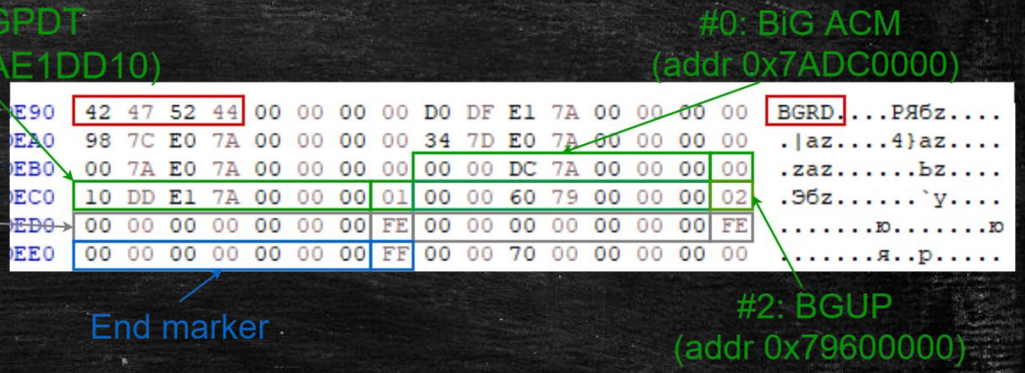
BiosGuardServices	File	SMM module	BiosGuardServices
MM dependency section	Section	MM dependency	
PE32 image section	Section	PE32 image	
UI section	Section	UI	
Version section	Section	Version	

- In some old firmwares this module is called PfatServices (can still be seen on Insyde)

Only BIOS Guard Control MSR indicates, if Intel BiG enabled or not!

Intel BIOS Guard Components

0 - Encrypted ACM
1 - BiG Platform Data Table (BGPDT)
2 - BiG update package (BGUP)
3 - BGUP certificate
4 - BIOS Guard log



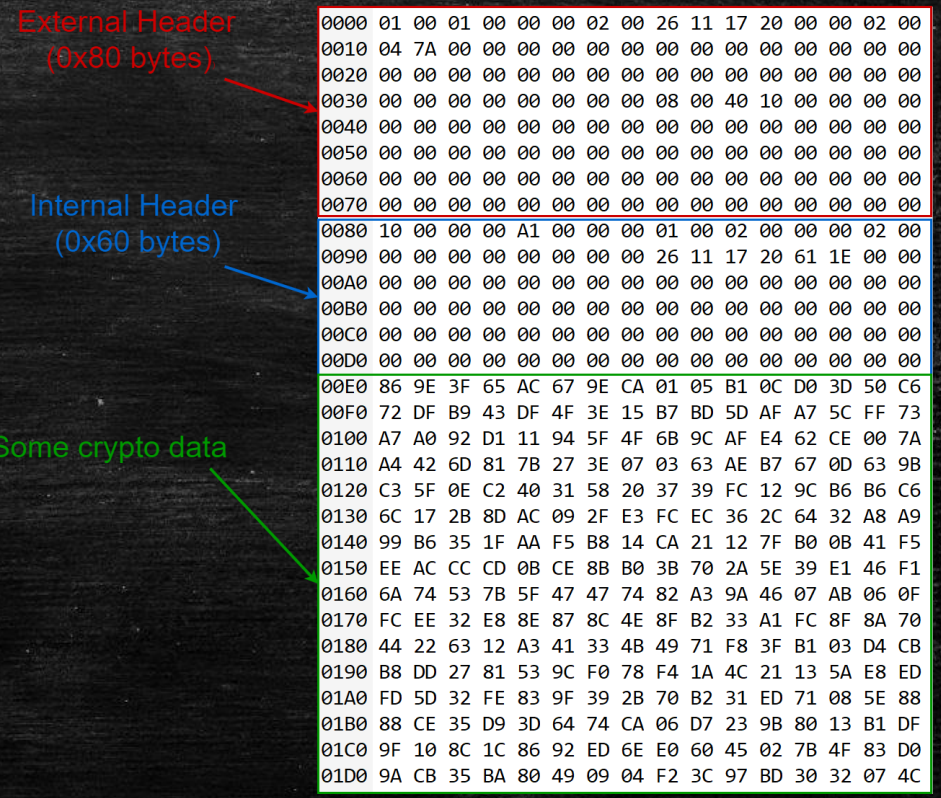
Betraying the BIOS: Where the Guardians of the BIOS are Failing by Alex Matrosov

Breaking Through Another Side by Alex Matrosov

Intel BIOS Guard ACM

Modifying the external header will not affect BiG ACM execution.

Internal header corrupting will result in a system crash.



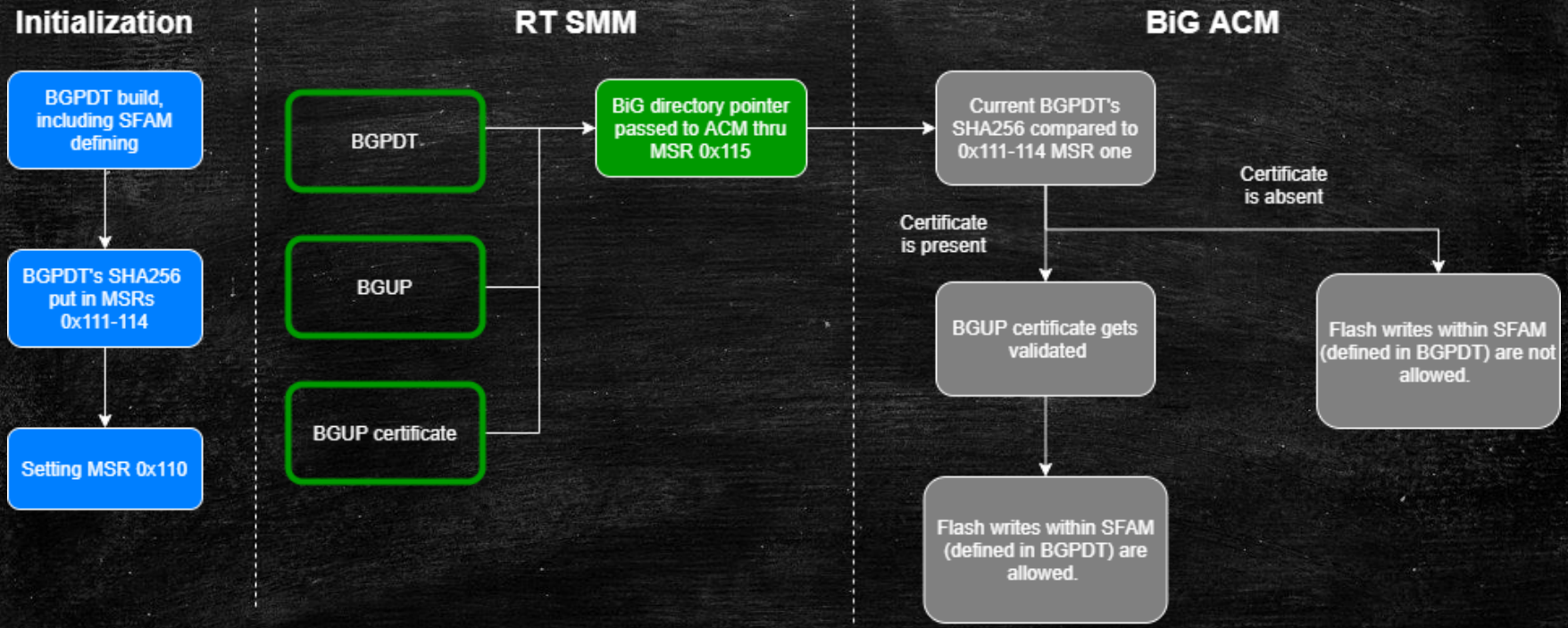
Intel BIOS Guard invoking

- 0x115 MSR - points to BiG components array
- 0x116 MSR - any write will trigger BiG ACM execution

The ACM will run only on BSP, all APs will be put to Sleep mode via invoking this ACM on them the same way.

```
for ( i = 0; i < gSmst->NumberOfCpus; ++i ) {
    if ( i != gSmst->CurrentlyExecutingCpu )
    {
        gSmst->SmmStartupThisAp(
            TriggerBiosGuardAcm,
            i,
            mBiosGuardDirectoryPtr);
    }
}

TriggerBiosGuardAcm(mBiosGuardDirectoryPtr);

…

void TriggerBiosGuardAcm(void *BiosGuardDirectoryPtr)
{
    …
    AsmWriteMsr64(MSR_PLAT_FRMW_PROT_TRIG_PARAM,
BiosGuardDirectoryPtr);
    AsmWriteMsr64(MSR_PLAT_FRMW_PROT_TRIGGER, 0);
    …
}
```

Intel BIOS Guard Verification Flow



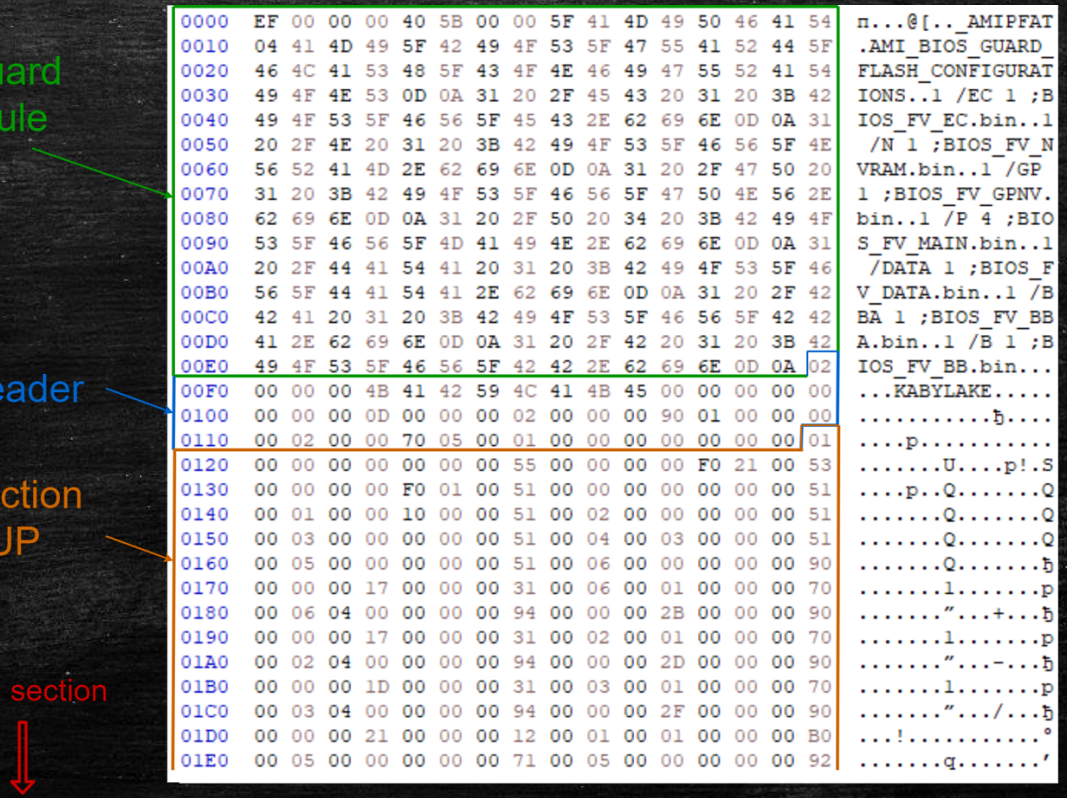
AMI BIOS Guard update capsule

AMI BIOS Guard update capsule header

First BGUP header

BiG script section of the BGUP

Data section



Intel BIOS Guard Script

```
0x0000  start
0x0008  set     flash0, 34F000h
0x0010  set     buffer0, FF000h
0x0018  set     r0, 0
0x0020  set     r1, 1000h
0x0028  set     r2, 0
0x0030  set     r3, 0
0x0038  set     r4, 3
0x0040  set     r5, 0
0x0048  set     r6, 0
0x0050  jmp     lb_B8

0x0058  lb_58:
0x0058  add     r6, 1
0x0060  cmp     r6, r4
0x0068  jge     lb_158
0x0070  jmp     lb_B8

0x0078  lb_78:
0x0078  add     r2, 1
0x0080  cmp     r2, r4
0x0088  jge     lb_168
0x0090  jmp     lb_E8
```

```
0x0098  lb_98:
0x0098  add     r3, 1
0x00A0  cmp     r3, r4
0x00A8  jge     lb_178
0x00B0  jmp     lb_108

        ; check if the data is already flashed
0x00B8  lb_B8:
0x00B8  load    buffer1, flash0, r1
0x00C0  rdsts   r5
0x00C8  cmp     r5, 0
0x00D0  jne     lb_58
0x00D8  cmp     buffer0, buffer1, r1
0x00E0  je      lb_128

        ; erase necessary flash areas
0x00E8  lb_E8:
0x00E8  eraseblk flash_0
0x00F0  rdsts   r5
0x00F8  cmp     r5, 0
0x0100  jne     lb_78

        ; store the update to the flash
0x0108  lb_108:
0x0108  store   flash0, buffer0, r1
0x0110  rdsts   r5
0x0118  cmp     r5, 0
0x0120  jne     lb_98
```

```
0x0128  lb_128:
0x0128  add     r0, r1
0x0130  cmp     r0, 100000h
0x0138  jge     lb_180
0x0140  sub     flash_0, r1
0x0148  sub     buff0, r1
0x0150  jmp     lb_B8

0x0158  lb_158:
0x0158  set     r15, 1
0x0160  jmp     lb_180

0x0168  lb_168:
0x0168  set     r15, 2
0x0170  jmp     lb_180
```

BUT!

How did we manage to disassemble the Script?

We got a decrypted ACM :)

Intel BIOS Guard ACM decryption attempts

We used to try:

- Relocation of MMIO areas (for example, overlap SPIBAR to different BiG components)
- Dump decrypted ACM from cache and while debugging
- Find and exploit Race Conditions

Finally, when we started to intersect BiG directory items with ACM in physical memory...

Intel BIOS Guard ACM decryption attempts

It failed too! :)

It was really hard to dump the decrypted ACM. Intel did a good job on protecting ACM.

Intel BIOS Guard ACM decryption attempts

The decrypted version of ACM was inside Alder Lake package for Simics from the recent leak from Intel!

BIOSGuardModule_10.o	23.04.2020 9:20
empty.brdid	23.04.2020 9:20
huffman_cnl.dic1	23.04.2020 9:20

Intel BIOS Guard ACM decryption attempts

- chip-red-pill debugging setup (Apollo Lake architecture)

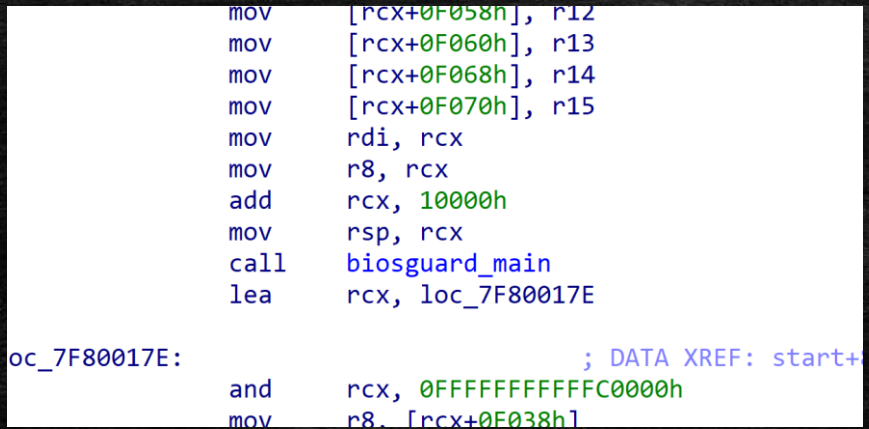
		(rst)	reserved
37:37	RW	0x0 (rst)	**TIMED_MWAIT_ENABLE (Timed MWAIT Enable)** If set, this field indicates that the core Timed WAIT feature is supported.
36:36	RO	0x0 (rst)	**RESERVED_3 (RESERVED_3)** reserved
35:35	RW	0x0 (rst)	**PFAT_ENABLE (PFAT_ENABLE)** If set, this field indicates that PFAT is enabled

- Find PFAT ACM from the system
- Enable Intel BIOS Guard via debugging with Intel DCI
- Dump & patch ‘wrmsr’ instruction (exit from ACM) with the code to dump the cache into physical memory

Intel BIOS Guard ACM analysis

ACM execution consists of two stages.

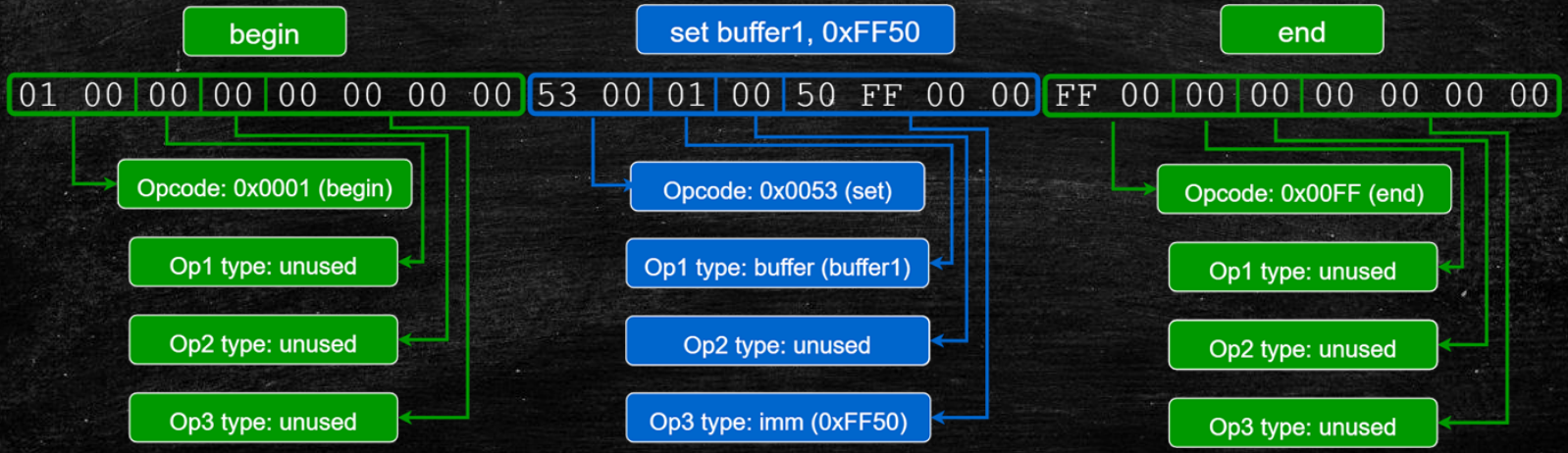
First - hardware initialization. Includes some initial I/O, MSR operations and BiG directory parsing + it’s security checks.



BIOS Guard script interpreter

Second stage - BIOS Guard Script interpreter, the heart of BiG ACM.

Instructions opcode analysis:



BIOS Guard script interpreter

A few types of operands:

- data buffer
- flash pointer
- register
- imm value

```
1 set r1, 0x1000
2 set flash1, 0xA00000
3 load buffer1, flash1, r1
4 cmp buffer1, buffer0, r1
5 jne _somewhere
```

buffer0 points to the BGUP's data section, not writeable.

See it for yourself!

We have created Intel BIOS Guard Script assembler/disassembler tool.

https://github.com/allowitsme/big-tool

#Takeaways

- Supply chain problems
  - disadvantage of security through obscurity model
- The problem in a basic component compromises all technologies it serves as a Root-of-Trust
- PC Firmware vendors aimed to protect code integrity, still forgetting about other attacks vectors
  - e.g. data-only attacks

Thank you