Safeguarding rootkits: Intel BootGuard

Alexander Ermolov
Security researcher at a.ermolov@dsec.ru flothrone@gmail.com
1. No motherboards were harmed
2. The Intel Boot Guard implementation details given here is a result of a reverse engineering process, so it may contain some inaccuracy compared to the Intel Boot Guard specification (which is not public)
Intel x86 platform firmware
Execution environments:

- Intel CPU
- Intel chipset subsystems
- ACPI EC

Platform firmware is stored on common SPI flash memory
Common SPI flash memory

System firmware is divided into regions:

- Flash Descriptors
  - Descriptors of other regions
  - Access permissions
  - …
- GbE
- ME
- ACPI EC (since Skylake)
- BIOS
Main execution environment (BIOS\OS)

Privilege levels:

- Ring 3: User Mode
- ... (omitted)
- Ring 0: Kernel Mode
- Ring -1: Hypervisor Mode
- Ring -2: System Management Mode (SMM)
Root of Trust

- Microcode ROM (== Boot ROM?)
- AES key for decrypting microcode updates
- Hash of an RSA public key which verifies the microcode updates
- Hash of an RSA public key which verifies other Intel blobs (e.g. ACM...)
Chipset subsystem integrated into:

- Q-type chipsets since 960 series (2006)
  - Intel ME 2.x – 5.x
- All chipsets since 5 series (2010)
  - Intel ME 6.x – 11.x, TXE 1.x – 3.x, SPS 1.x – 4.x

Platforms affected:

- Desktop, LaptopIntel Management Engine (ME)
- MobileIntel Trusted Execution Engine (TXE)/Security Engine
- ServerIntel Server Platform Services (SPS)
Most privileged and hidden execution environment (Ring -3):

- Hidden from CPU runtime memory in DRAM
- Full access to DRAM
- Working even when CPU is in S5 (system shutdown)
- Out-of-Band (OOB) access to network interface
- Runs firmware (based on RTOS ThreadX) from common SPI flash
CPU architectures

- ME 2.x – 10.x, SPS 1.x – 3.x
  - ARC (ARC32/ARCompact)
- TXE 1.x – 2.x
  - SPARC
- ME 11.x, SPS 4.x, TXE 3.x
  - x86
Root of Trust

- ME ROM with the bootcode
- Hash of an RSA public key which verifies ME FW
- AES key to store sensitive data
- Field Programmable Fuses (FPFs)
Intel ME FW is divided into partitions of various type:

- Code
- Data
- File System
- ...

Code partitions verification flow ->
Integrated in Intel SoC since ? Bay Trail ?

Seems to be truncated version of Intel ME:
- ROM with bootcode and SRAM
- Has its own HECI
- Has a DMA engine ( ? shares some memory with ME ? )
- Runs firmware (ISHC partition of ME FW) from common SPI flash

Firmware can be developed and signed by Intel/OEM
MCU, present only on laptops to make power-management and ACPI-related features:

- Fn-buttons
- Touchpad/keyboard
- Battery supply
- ...

 Runs firmware (generally without any protection against modifications) from:
- internal flash (can be updated by BIOS, the update binary is included into BIOS)
- common SPI flash (since Skylake)
BIOS protection mechanisms

- Hardware Write Protect jumper
- Protected Range (PR) registers
- BLE (BIOS_WE)
- SMM_BWP
- Intel BIOS Guard (PFAT)
- Intel Boot Guard

Though some vendors using a few of these, but there are always many that don’t care...
Intel Boot Guard 1.x*

* - not official version number, this is how I order it’s versions
Hardware-based boot integrity protection available since Haswell

Operating modes:
• Measured Boot (MB)
• Verified Boot (VB)
• MB + VB
Measured Boot uses the Trusted Platform Module (TPM) Platform Configuration Registers (PCRs) to reflect boot components integrity.

Measure \(\text{(data)}:\)

\[
\text{PCR} = H(\text{PCR} | H(\text{data}))
\]

Some sensitive data can be sealed \(\text{(TPM\_Seal)}\) to the PCRs state.
Verified Boot cryptographically verifies the integrity of boot components

Options, in case of a verification fail:

• Do nothing
• Immediate shutdown
• Shutdown in timeout (e.g. 1 or 30 minutes)
Field Programmable Fuses (FPFs) are the hardware non-volatile storage inside Intel ME so only it can program and read them.

FPFs fits perfect to store the Intel BG configuration:

- Fuses can be one-time programmable
- Access only through Intel ME
## Intel Boot Guard

### Hash Key Configuration for Bootguard/ISH

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEM Public Key Hash</td>
<td>00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

This option is for entering the raw hash 256 bit string for OEM Public Key Hash.

### Boot Guard Configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Manifest ID</td>
<td>0x0</td>
</tr>
<tr>
<td>Boot Guard Profile Configuration</td>
<td>Boot Guard Profile 0 - No_FVME</td>
</tr>
<tr>
<td>CPU Debugging</td>
<td>Enabled</td>
</tr>
<tr>
<td>BSP Initialization</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

This option configures which Boot Guard Policy Profile is used.

When set to Enabled, the CPU debugging capability profile is used.

This setting determines BSP behavior when it receives a command to enable debug.
typedef struct BG_PROFILE
{
    unsigned long Force_Boot_Guard_ACM : 1;
    unsigned long Protect_BIOS_Environment : 1;
    unsigned long CPU_Debugging : 1;
    unsigned long BSP_Initialization : 1;
    unsigned long Measured_Boot : 1;
    unsigned long Verified_Boot : 1;
    unsigned long Key_Manifest_ID : 4;
    unsigned long Enforcement Policy : 2;    // 00b - do nothing
                                           // 01b - shutdown timeout
                                           // 11b - immediate shutdown
    unsigned long : 20;
};

Intel BG. Configuration
<table>
<thead>
<tr>
<th>BG profiles</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_FVME</td>
<td>Disabled</td>
</tr>
<tr>
<td>VE</td>
<td>VB, shutdown timeout</td>
</tr>
<tr>
<td>VME</td>
<td>VB + MB, shutdown timeout</td>
</tr>
<tr>
<td>VM</td>
<td>VB + MB, do nothing</td>
</tr>
<tr>
<td>FVE</td>
<td>VB, immediate shutdown</td>
</tr>
<tr>
<td>FVME</td>
<td>VB + MB, immediate shutdown</td>
</tr>
</tbody>
</table>
Intel BG configuration process

1) Prepare image with ME NVARs that should be committed to FPFs
   • Intel Flash Image Tool

2) Close the manufacturing mode and issue a global reset
   • Intel Flash Programming Tool
Let’s take a deeper look on BG implementation...

• Gigabyte GA-H170-D3H  BG support present
• Gigabyte GA-Q170-D3H  BG support present
• Gigabyte GA-B170-D3H  BG support present
• MSI H170A Gaming Pro  BG support not present
• Lenovo ThinkPad 460  BG support present
• Lenovo Yoga 2 Pro  BG support not present
• Lenovo U330p  BG support not present
No image of it for researching, but some docs mention that it does:

1) Find the Firmware Interface Table (FIT)
   • FIT base address is located at 0xFFFFFFF0

2) Find Intel BIOS Authenticated Code Module (ACM), verify, load and execute it
   • FIT contains the base address of Intel BIOS ACM
Intel CPU boot ROM

**Diagram:**
- **Intel CPU**
  - **RESET**
  - Intel CPU boot ROM
- **SPI flash**
  - FIT
  - 0xFFFFFFCO
The FIT is a table of few entries and the first entry is a FIT header

typedef struct FIT_HEADER
{
    char        Tag[8];    // '_FIT_'
    unsigned long NumEntries; // including FIT header entry
    unsigned short Version;  // 1.0
    unsigned char  EntryType; // 0
    unsigned char  Checksum;
};
Other FIT entries have the same format
They describes Intel blobs that are to be parsed/executed before the BIOS, hence before the Legacy RESET-vector (0xFFFFFFFF0)

typedef struct FIT_ENTRY
{
    unsigned long     BaseAddress;
    unsigned long     : 32;
    unsigned long     Size;
    unsigned short    Version;           // 1.0
    unsigned char     EntryType;
    unsigned char     Checksum;
};
enum FIT_ENTRY_TYPES {
    FIT_HEADER = 0,
    MICROCODE_UPDATE,
    BIOS_ACM,
    BIOS_INIT = 7,
    TPM_POLICY,
    BIOS_POLICY,
    TXT_POLICY,
    BG_KEYM,
    BG_IBBM
};
typedef struct BIOS_ACM_HEADER
{
    unsigned short ModuleType; // 2
    unsigned short ModuleSubType; // 3
    unsigned long HeaderLength; // in dwords
    unsigned long : 32;
    unsigned long : 32;
    unsigned long ModuleVendor; // 8086h
    unsigned long Date; // in BCD format
    unsigned long TotalSize; // in dwords
    unsigned long unknown1[6];
    unsigned long EntryPoint;
    unsigned long unknown2[16];
    unsigned long RsaKeySize; // in dwords
    unsigned long ScratchSize; // in dwords
    unsigned char RsaPubMod[256];
    unsigned long RsaPubExp;
    unsigned char RsaSig[256];
};
Intel CPU boot ROM

RESET

- Intel CPU boot ROM
- Intel BIOS ACM

SPI flash

- FIT
- Intel BIOS ACM

0xFFFFF0C
Parse FIT:

1) Retrieve hash of OEM Root Pubkey and Boot Policies from Intel ME
2) Locate Key Manifest (KEYM) and verify it
3) Locate IBB Manifest (IBBM) and verify it
enum FIT_ENTRY_TYPES {
    FIT_HEADER = 0,
    MICROCODE_UPDATE,
    BIOS_ACM,
    BIOS_INIT = 7,
    TPM_POLICY,
    BIOS_POLICY,
    TXT_POLICY,
    BG_KEYM,
    BG_IBBM
};
Intel CPU boot ROM

**Intel CPU**

- **RESET**
- Intel CPU boot ROM
- Intel BIOS ACM

**Intel ME**

**SPI flash**

- FIT
- Intel BIOS ACM
- KEYM
- IBBM
- 0xFFFFFC0

**FPFs**

www.zeronights.org
typedef struct KEY_MANIFEST
{
    char Tag[8]; // __KEYM__’
    unsigned char : 8; // 10h
    unsigned char : 8; // 10h
    unsigned char : 8; // 0
    unsigned char : 8; // 1
    unsigned short : 16; // 0Bh
    unsigned short : 16; // 20h == hash size?
    unsigned char IbbmKeyHash[32]; // SHA256 of an IBBM public key
    BG_RSA_ENTRY OemRootKey;
};
typedef struct BG_RSA_ENTRY
{
    unsigned char       : 8;  // 10h
    unsigned short      : 16;  // 1
    unsigned char       : 8;  // 10h
    unsigned short      RsaPubKeySize;  // 800h
    unsigned long       RsaPubExp;
    unsigned char       RsaPubKey[256];
    unsigned short      : 16;  // 14
    unsigned char       : 8;  // 10h
    unsigned short      RsaSigSize;  // 800h
    unsigned short      : 16;  // 0Bh
    unsigned char       RsaSig[256];
};
typedef struct IBB_MANIFEST
{
    ACBP Acbp; // Boot policies
    IBBS Ibbs; // IBB description
    IBB_DESCRPTORS[];
    PMSG Pmsg; // IBBM signature
};
typedef struct ACBP {
    char Tag[8]; // '__ACBP__'
    unsigned char : 8; // 10h
    unsigned char : 8; // 1
    unsigned char : 8; // 10h
    unsigned char : 8; // 0
    unsigned short : 16; // x & F0h = 0
    unsigned short : 16; // 0 < x <= 400h
};
typedef struct IBBS
{
    char Tag[8];  // '__IBBS__'
    unsigned char : 8;  // 10h
    unsigned char : 8;  // 0
    unsigned char : 8;  // 0
    unsigned char : 8;  // x <= 0Fh
    unsigned long : 32;  // x & FFFFFFFF8h = 0
    unsigned long Unknown[20];
    unsigned short : 16;  // 0Bh
    unsigned short : 16;  // 20h == hash size ?
    unsigned char IbbHash[32];  // SHA256 of an IBB
    unsigned char NumIbbDescriptors;
};
Initial Boot Block (IBB) content is described in IBB_DESCRIPTORS

typedef struct IBB_DESCRIPTOR
{
    unsigned long : 32;
    unsigned long BaseAddress;
    unsigned long Size;
};

So the concatenation of blocks (usually all SEC/PEI modules in UEFI image) that are pointed by IBB descriptors forms the IBB
typedef struct PMSG {
    char Tag[8]; // '__PMSG__'
    unsigned char : 8; // 10h
    BG_RSA_ENTRY IbbKey;
}
Hence, the SEC/PEI code is verified before the CPU starts executing from the RESET vector (FFFFFFF0h)

Then the BootGuard supporting code in PEI must verify the DXE volumes

Such PEI module is developed by OEM, e.g.:

- Lenovo
  LenovoVerifiedBootPei {B9F2AC77-54C7-4075-B42E-C36325A9468D}

- Gigabyte
  BootGuardPei {B41956E1-7CA2-42DB-9562-168389F0F066}
This BootGuard PEI module does:

- Find the hash table by the GUID
- Verify the DXE code pointed by this hash table
if (EFI_PEI_SERVICES->GetBootMode() != BOOT_ON_S3_RESUME)
{
    if (!FindHashTable())
        return EFI_NOT_FOUND;

    if (!VerifyDxe())
        return EFI_SECURITY_VIOLATION;
}
Hash table PEI module \{389CC6F2-1EA8-467B-AB8A-78E769AE2A15\}

typedef struct HASH_TABLE {
    char        Tag[8];           // \$HASHTBL
    unsigned long NumDxeDescriptors;

    DXE_DESCRIPTORS[];
};

typedef struct DXE_DESCRIPTOR {
    unsigned char BlockHash[32];   // SHA256
    unsigned long Offset;
    unsigned long Size;
};
int bootMode = EFI_PEI_SERVICES->GetBootMode();

if (bootMode != BOOT_ON_S3_RESUME &&
    bootMode != BOOT_ON_FLASH_UPDATE &&
    bootMode != BOOT_IN_RECOVERY_MODE)
{
    if (!FindHashTable())
        return EFI_NOT_FOUND;

    if (!VerifyDxe())
        return EFI_SECURITY_VIOLATION;
}

Hash table PEI module {389CC6F2-1EA8-467B-AB8A-78E769AE2A15}

typedef HASH_TABLE DXE_DESCRIPTORS[];

typedef struct DXE_DESCRIPTOR
{
    unsigned char BlockHash[32]; // SHA256
    unsigned long BaseAddress;
    unsigned long Size;
};
Safeguarding rootkits
One day I found out that some systems have the SPI flash regions unlocked and the BootGuard configuration not set (nor enabled, nor disabled):

• All Gigabyte systems
• All MSI systems
• 21 Lenovo branded notebook machine types and 4 ThinkServer machine types
• other few vendors I cannot mention at the moment

That’s because of the close manufacturing fuse was not set at the end of the manufacturing line.
«Lenovo has released fixes for the affected products, which can be found at https://support.lenovo.com/solutions/LEN_9903 or via our security advisory website, https://support.lenovo.com/product_security, and we have adjusted manufacturing processes, where necessary, to prevent reoccurrence of this issue in the future. We sincerely appreciate Mr. Ermolov's responsible disclosure and partnership in this matter.»
«Intel’s guidance to our business partners is to close manufacturing mode at the end of production in order to maximize the security of the platform.»
So any user could configure the Intel BG instead of OEM:

• Load into OS
• Modify BIOS
• Write proper BG configuration and verification entities (KEYM, IBBM) using Intel Flash Image Tool
• Set the closemnf fuse using the Intel Flash Programming Tool

This will permanently enable Intel BG on the system and will protect modified BIOS
DEMO
The rootkit can be an SMM driver with the following capabilities:

1) Executed during OS
   • Registers a SMI ISR and configure a timer to generate SMI events

2) Full (except ME UMA) access to CPU physical address space and complete isolation from OS
   • SMRAM

3) An encrypted blob which self-decrypts itself during upon each execution
Hence, the issue allows:

- to create hidden, black box and irremovable (even with SPI flash programmer) rootkit on a platform

- to modify the ISH firmware on the platform which opens a new attack surface
Safeguarding rootkits
Conclusion

* - not official version number, this is how I order it’s versions

62
Conclusion

- Description of Intel BootGuard implementation
- There are so many proprietary Intel blobs executing before RESET-vector
- The number of execution environments is increasing (CPU x86_64, ME x86, ISH x86, ...)
- A scenario to make any past BIOS modification permanent and updatable only from BG Root Key owner
Mitigation

• Vendors that intentionally left the closemnf fuse unset in servicing purposes should find another way

• Vendors that left the closemnf fuse by mistake should roll out a fix (Lenovo have already done this)

• Users can disable the Intel BG technology manually:
  Just run the MEinfo to make sure the Intel BG in not configured on the platform and run the FPT with –closemnf argument
## Mitigation

<table>
<thead>
<tr>
<th>Feature</th>
<th>FPF</th>
<th>ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEM Public Key Hash FPF</td>
<td>Not set</td>
<td></td>
</tr>
<tr>
<td>OEM Public Key Hash ME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>KM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>BSMM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>GuC Encryption Key FPF</td>
<td>Not set</td>
<td></td>
</tr>
<tr>
<td>GuC Encryption Key ME</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>FPF</th>
<th>ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force Boot Guard ACM</td>
<td>Not set</td>
<td>Disabled</td>
</tr>
<tr>
<td>Protect BIOS Environment</td>
<td>Not set</td>
<td>Disabled</td>
</tr>
<tr>
<td>CPU Debugging</td>
<td>Not set</td>
<td>Enabled</td>
</tr>
<tr>
<td>BSP Initialization</td>
<td>Not set</td>
<td>Enabled</td>
</tr>
<tr>
<td>Measured Boot</td>
<td>Not set</td>
<td>Disabled</td>
</tr>
<tr>
<td>Verified Boot</td>
<td>Not set</td>
<td>Disabled</td>
</tr>
<tr>
<td>Key Manifest ID</td>
<td>Not set</td>
<td>0x0</td>
</tr>
<tr>
<td>Enforcement Policy</td>
<td>Not set</td>
<td>0x0</td>
</tr>
<tr>
<td>PTT</td>
<td>Not set</td>
<td>Enabled</td>
</tr>
<tr>
<td>EK Revoke State</td>
<td>Not Revoked</td>
<td></td>
</tr>
<tr>
<td>PTT RTC Clear Detection FPF</td>
<td>Not set</td>
<td></td>
</tr>
</tbody>
</table>
## Mitigation

<table>
<thead>
<tr>
<th>Feature</th>
<th>FPF</th>
<th>ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEM Public Key Hash FPF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OEM Public Key Hash ME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>KM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>BSMM SVN FPF</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>GuC Encryption Key FPF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GuC Encryption Key ME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Force Boot Guard ACM</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Protect BIOS Environment</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>CPU Debugging</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>BSP Initialization</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>Measured Boot</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Verified Boot</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Key Manifest ID</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>Enforcement Policy</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>PTT</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>PTT Lockout Override Counter</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>FK Revoke State</td>
<td>Not Revoked</td>
<td></td>
</tr>
</tbody>
</table>
Thank you