Breaking Through Another Side

Bypassing Firmware Security Boundaries from Embedded Controller

Alex Matrosov
NVIDIA

Alexandre Gazet
AIRBUS

Disclaimer

All the details given about BIOS Guard technology is based on our own analysis and reverse-engineering[1]. Even with our best intents it may be inaccurate or contains errors.

[1]Actually ~5 months of passionate reverse-engineering nights in Portland and Toulouse

What are the Security Boundaries in HW world?

- ✓ Limitations of current Threat Model
- ✓ Security boundaries for firmware update process

Dissecting an Embedded Controller

- ✓ EC internals and previous attacks
- ✓ Why is EC not a security boundary?
- ✓ Breaking Lenovo EC update process

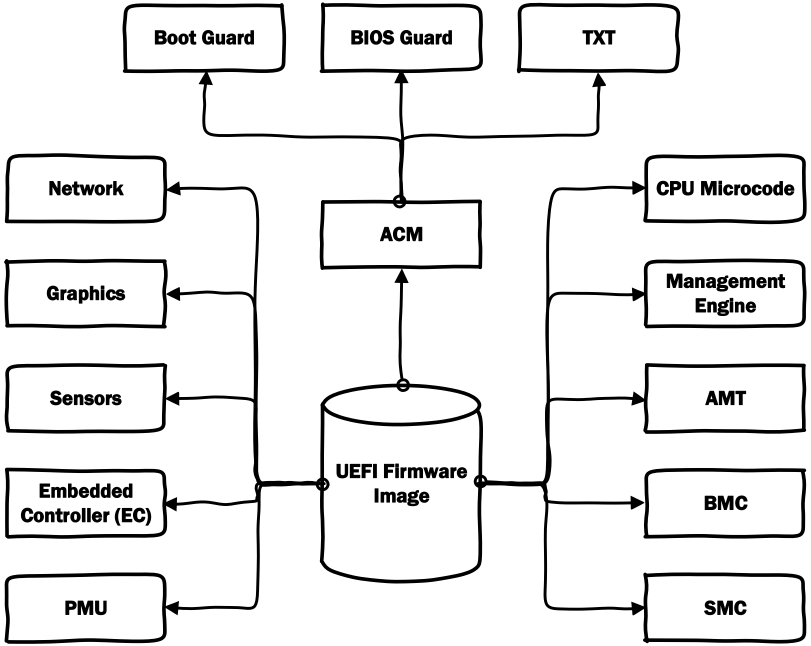
Deep dive into Bios Guard

- ✓ BIOS Guard internals (include BG script)
- ✓ EC and BIOS Guard relations
- ✓ Attack scenarios from BIOS and EC

What are Security Boundaries in HW world?

How many 3rd-party chips in your laptop?

- ❑ TPM module
- ❑ USB controller
- ❑ Embedded Controller (EC)
- ❑ Fingerprint Reader
- ❑ Touchpad
- ❑ and many others



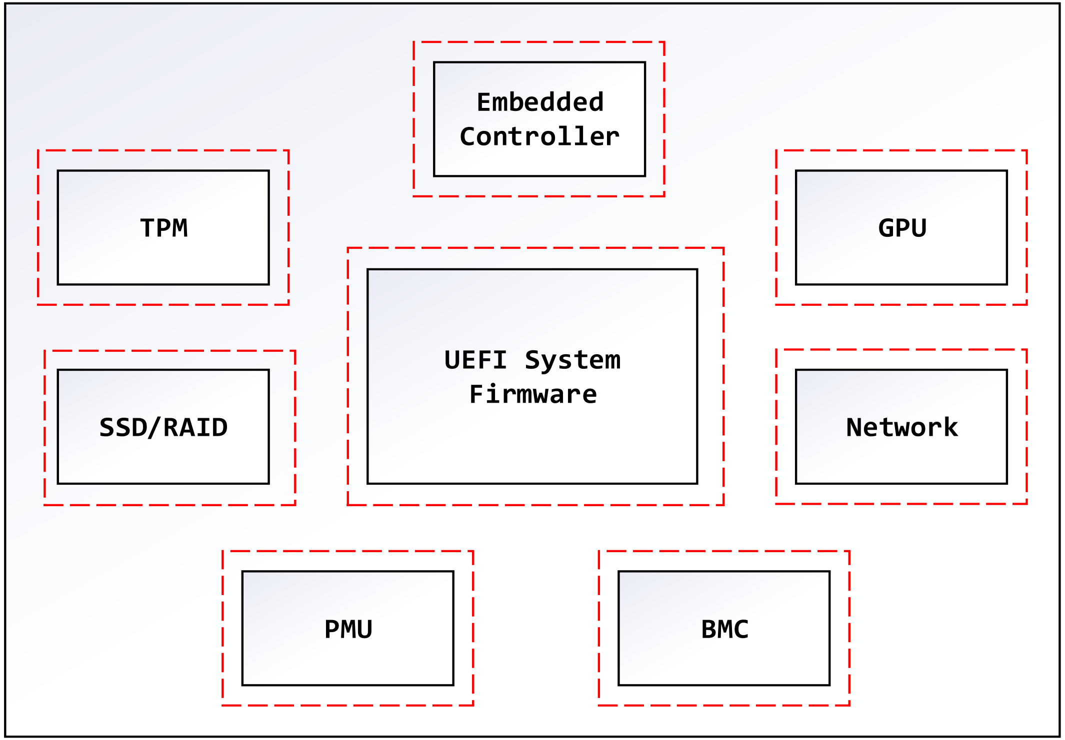
Hardware Security Boundaries

Most of those chips are:

- ❑ Not under direct control from laptop vendors
- ❑ Involved in security features implementation
- ❑ Connected to UEFI firmware (BIOS)
- ❑ Considered to generate trusted I/O
- ❑ Mostly out of the supervision scope of the main CPU

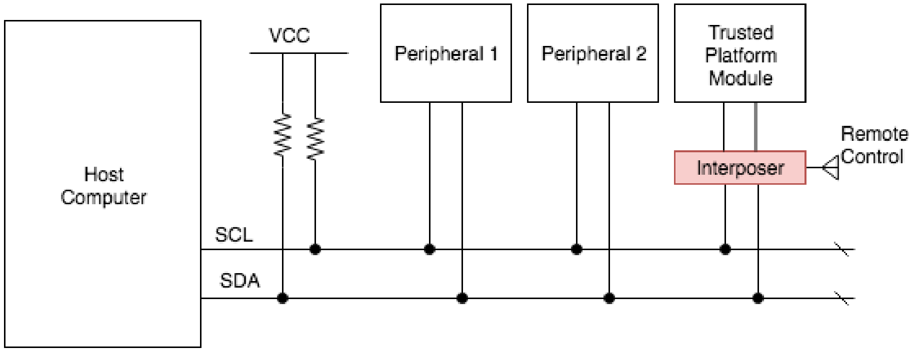
How can we trust anything that is not under our system control?

HW/FW Security != sum of all Boundaries



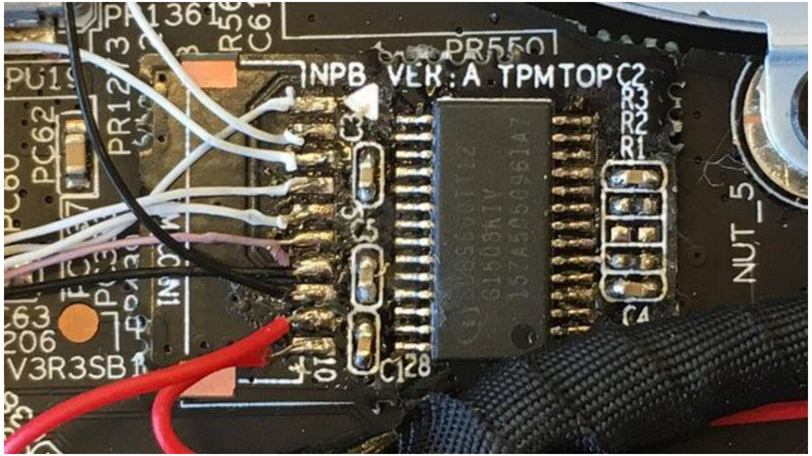
In current threat model HW is trusted 😈

@uffeux



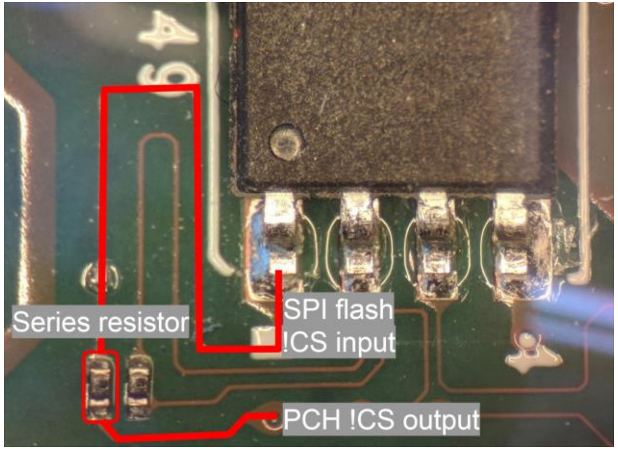
https://github.com/nccgroup/TPMGenie

@qrs



Intel Boot Guard TOCTOU from SPI flash

@qrs @peterbjornx



Authenticated once != trusted forever

https://edk2-docs.gitbooks.io/security-advisory/content/bootguard-toctou-vulnerability.html

BMC is inside trusted boundaries

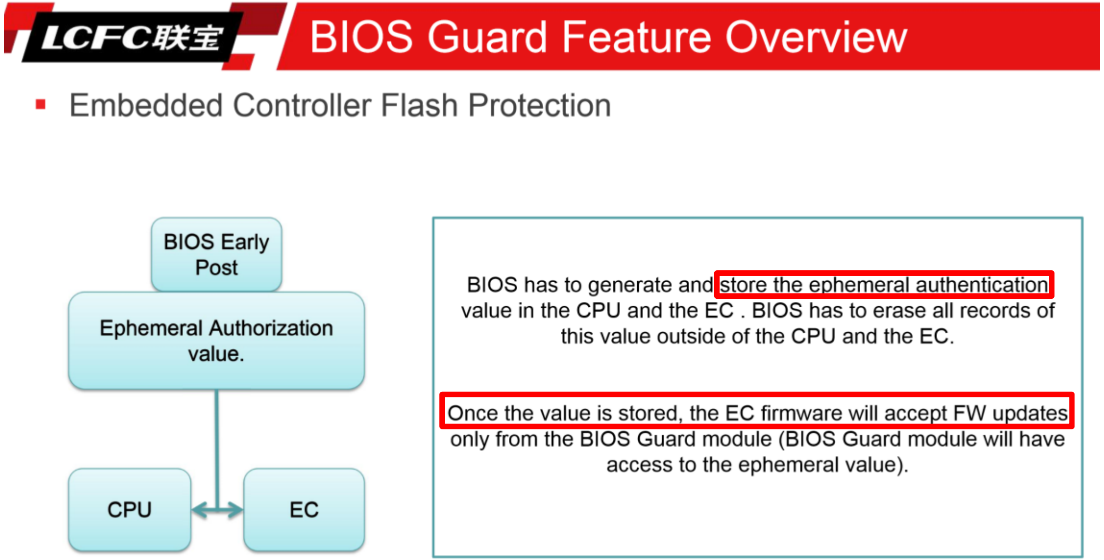
UEFI firmware blindly trust all hardware

But hardware can attack UEFI firmware 😈

https://airbus-seclab.github.io/ilo/ZERONIGHTS2018-Slides-EN-Turning_your_BMC_into_a_revolving_door-perigaud-gazet-czarny.pdf

Why EC got our attention?

We were researching BIOS Guard implementation on P50. Surprisingly to us, we found some relations between EC and BIOS Guard (will be discussed later in details).



Dissecting Embedded Controller

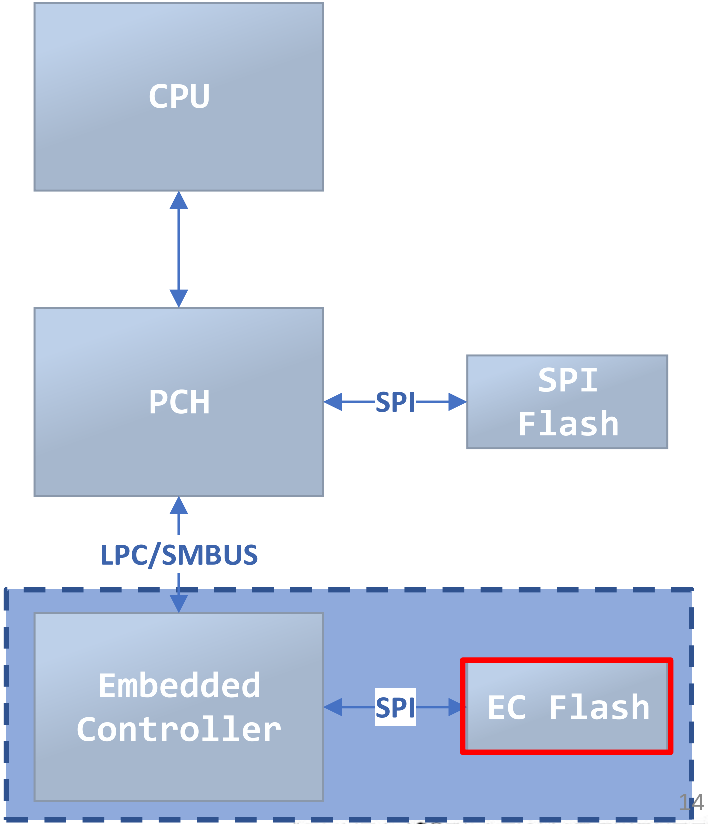
Our target platforms: Lenovo P50 and T540p

What is an Embedded Controller (EC)?

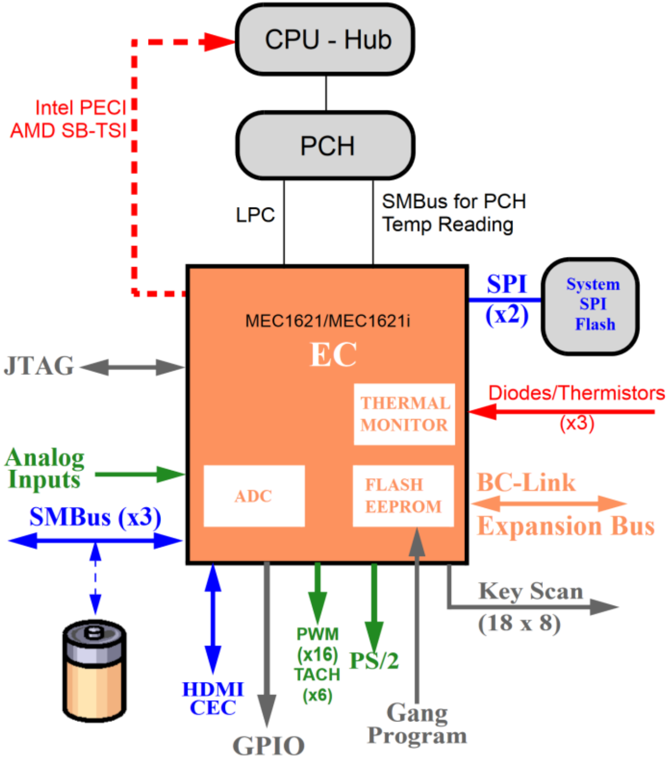
- Small 32-bit microcontroller, power every laptop
- Responsible for multiple things
  - Power management and battery life control
  - Thermal control sensors
  - Keyboard controller and dispatcher
- Also involved in security features implementation
- Manufacturing mode locks
- Keeping secrets outside of BIOS and NVRAM
- Intel BIOS Guard implementation

Lenovo ThinkPad EC

- Microchip MEC16xx family
- MEC1653 for Lenovo P50
- MEC1633 for Lenovo P540p
- ROM size 280k
- ARC-625D processor core
- Multi-device advanced I/O controller
- Collection of logical devices:
  - Keyboard Controller (8042)
  - ACPI EC Channels (4 of them)
  - Embedded Flash Interface
  - *etc.*



Modern EC SoC



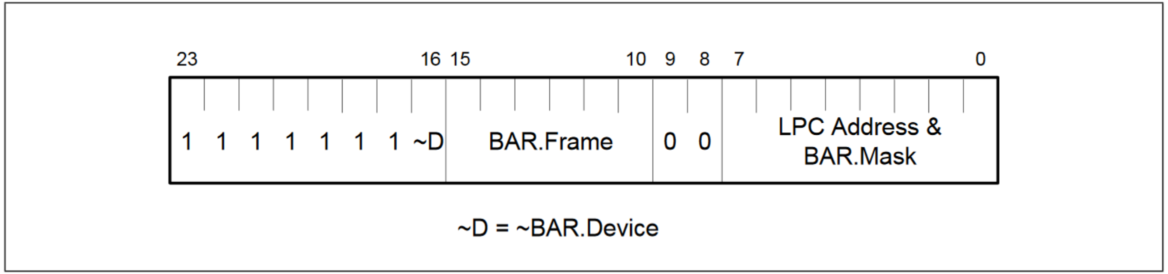
http://ww1.microchip.com/downloads/en/DeviceDoc/00002338A.pdf

Mapping Embedded Controller Endpoints

"*Logical devices [...] are peripherals that are located on the MEC16xx and are **accessible to the Host over the LPC bus**.*"

Low Pin Count (LPC) interface from EC point of view:

- ❑ Is itself a Logical Device (LD)
- ❑ Logical Device Number 0xC (LDN)
- ❑ Used to expose other LDs on the LPC bus
- ❑ Configuration registers (BAR) in the range FF_3360h - FF_3384h



Methodology

From EC:

- ❑ Identify LPC BAR configuration code
- ❑ Recover logical device ⬄ IO ports mapping
- ❑ EC’s endpoints exposed to host

From host:

- ❑ Find UEFI/BIOS ⬄ EC communications
- ❑ EDK2 EFI_CPU_IO2_PROTOCOL
- ❑ Lenovo’s EcIoDxe and EcIoSmm modules

```
# int __cdecl LDN30_enable()
LDN30_enable:
mov     r0, ID_LDN30      # ldn, 0x30
mov     r1, 0xAF          # 15E0 = 0xAF << 5
asl     r1, r1, 5         # lpc_host_addr
b.d     config_logical_device
mov     r2, 0x1F          # mask
# End of function LDN30_enable
```

Recovered mapping

❑ LDN00 (MAILBOX_INTERFACE)	0x1610
❑ LDN01 (KEYBOARD_CONTROLLER_8042)	0x0060-0x0064
❑ LDN02 (ACPI_EC_0)	0x0062-0x0066
❑ LDN03 (ACPI_EC_1)	0x1600-0x1604
❑ LDN04 (ACPI_EC_2)	0x1630-0x1634
❑ LDN05 (ACPI_EC_3)	0x1618
❑ LDN07 (UART)	0x03F8
❑ LDN0E (EMBEDDED_FLASH_INTERFACE)	0x1612-0x1616
❑ LDN11 (EM_INTERFACE_0)	0x1640
❑ LDN20 (BIOS_DEBUG_PORT_0)	0x1608
❑ LDN21 (BIOS_DEBUG_PORT_1)	0x160A
❑ LDN30 (unknown)	0x15E0

Attacking EC Update Process

Previous very cool works

Alexandre Gazet

«Sticky finger & KBC Custom Shop», Recon 2011

- http://esec-lab.sogeti.com/static/publications/11-recon-stickyfingers_slides.pdf

Matthew Chapman

Unlocking my Lenovo laptop

- http://zmatt.net/unlocking-my-lenovo-laptop-part-1/

Hamish Coleman

Infrastructure for examining and patching Thinkpad embedded controller firmware

- https://github.com/hamishcoleman/thinkpad-ec

EC firmware update process

On many platforms EC firmware not authenticated just flashed "as is"

- Typical EC programming is just read/write to HW port
- Verification is about integrity of flashed bytes
- Authentication mostly implemented outside of EC

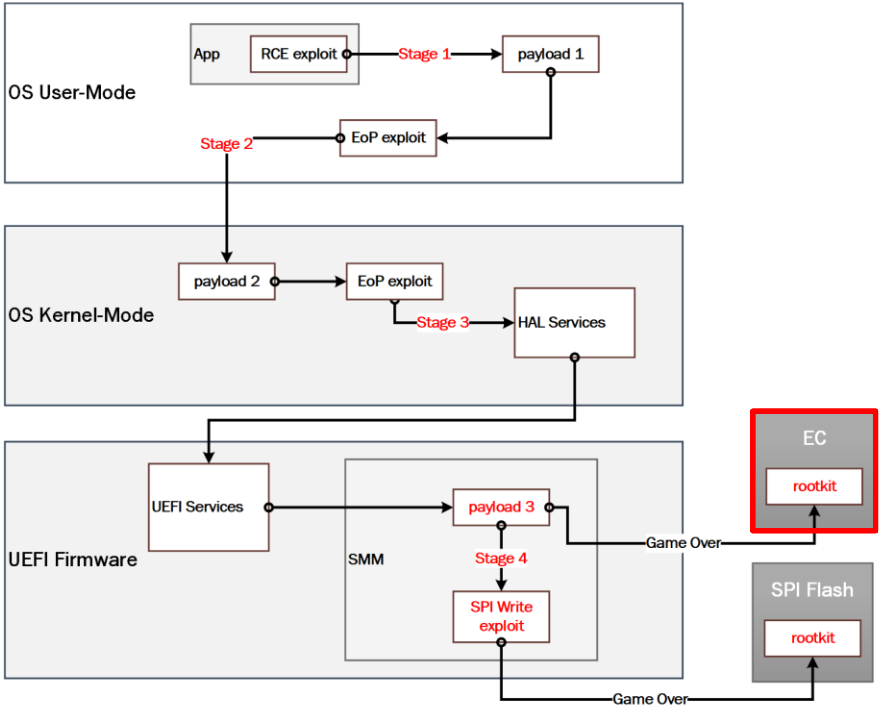
```
if ( update_image_buffer < 0x2000 && ec_cmd > 0xA || ec_cmd >= 0x20 || !upda
  break;
if ( cmd_arg )
{
  WriteHwPort(ec_cmd);
  printf("Send Erase Command...\n");
}
Sleep(100u);
printf("Erase Done\n");
if ( sub_401170() )
{
  printf("Return from Erase Checking: Done\n");
  if ( !cmd_arg )
  {
    printf("Send Erase Command Again\n");
    WriteHwPort(ec_cmd);
    Sleep(0x64u);
  }
  update_counter = 0;
  while ( !SendProgramCmd() )
  {
    printf("Programming the EC Firmware now.....\n");
    ++update_counter;
    ReadHwPort();
    ReadHwPort();
    WriteHwPort(ec_cmd);
    Sleep(0x64u);
  }
  printf("The EC Firmware Programmed Done & Verification Success.\n");
  ++ec_cmd;
}
else
{
  printf("Return from CheckDataFF: false\n");
  ++ec_cmd;
}
```

https://github.com/system76/ecflash
https://github.com/hughsie/fwupd/tree/master/plugins/superio

The ways to gain persistence on EC

- Physical access (most of the cases JTAG on EC chip not disabled)
- EC Update Tool from OS (usually the same tool as BIOS update)
- BIOS EC update DXE driver can be called from SMM or DXE shellcode
- All EC image authentication is happening in BIOS, architectural problem with TOCTOU by design hard to avoid

Impact of EC update auth bypass



persistent on EC

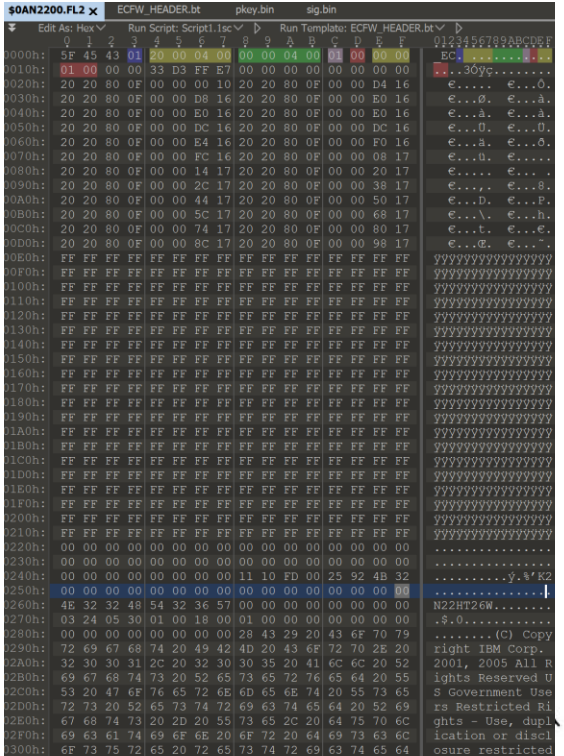
Lenovo Thinkpad EC update process

- Target system: Lenovo Thinkpad T540p and P50
- P50 EC chip: MEC1653
- Update tools from OS initiate EC update process
- BIOS responsible for flashing and authenticating the update image

EcFwUpdateDxe (0C396FCA-6BDA-4A15-B6A3-A6FA4544BDB7) 😈

Lenovo Thinkpad EC update header

```
typedef struct _ECFW_HEADER {
    UINT8           signature[3]; //_EC
    UINT8           version;
    UINT32          file_size;
    UINT32          image_size;
    UINT8           hash_algo; // 1 == SHA256
    SIGN_ALGORITHM  sign_algo; // 1 == RSA2048
    UINT16          hash_crc16; // CRC16
    UINT16          header_crc16; // CRC16
    UINT8           unknown;
} ECFW_HEADER;
```



Lenovo Thinkpad EC update process

OS

Lenovo TDK update tool

map EC update image to memory

set NVRAM var 'LenovoEcfwUpdate'

```
while ( v7 - &LenovoEcfwUpdate <= v5 );
memset_(buffer, 0, 1u);
buffer[0] = 1;
TdkBinCreateFromBuff(buffer, 1ui64, &tdk_bin);
result = TdkVariableSet(&a1, &a2, 7u, tdk_bin);
```

Lenovo EcFwUpdateDxe (not SMM)

```
res = LoadFirmware();
if ( res >= 0 )
{
  res = ValidateFirmwareHeader();
  if ( res >= 0 )
  {
    UpdateEcFw(ecfw_bin);
    res = 0i64;
  }
```

BIOS

Lenovo Thinkpad EC update process

OS

Len

map EC
image to

```
while ( v7 -
memset_(buff
buffer[0] =
TdkBinCreate
result = Tdk
```

ot SMM)

```
Header();
;
```

```
case 0x83u:
  v5 = "ECFW image file is invalid";
  break;
case 0x84u:
  v5 = "Failed to load ECFW image file";
  break;
case 0x85u:
  v5 = "This system BIOS supports signed ECFW image only.";
  break;
case 0x86u:
  v5 = "This system BIOS supports unsigned ECFW image only.";
  break;
```

T540p case

BIOS

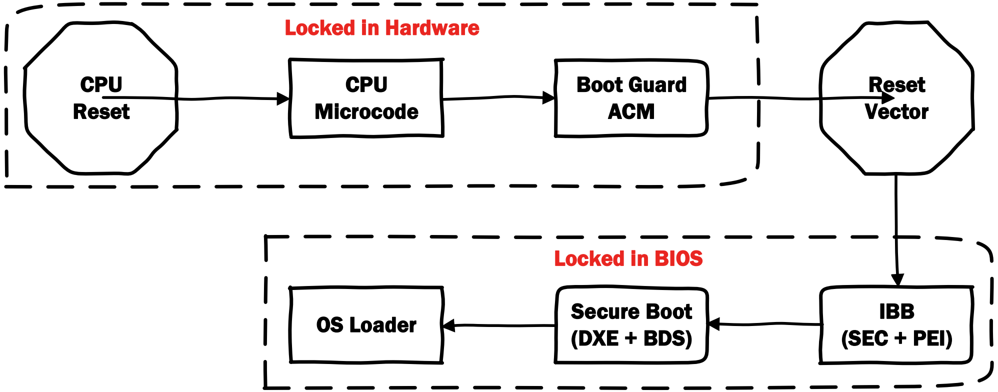
T540p EC can be exploited from OS by simple EC command sequence replay

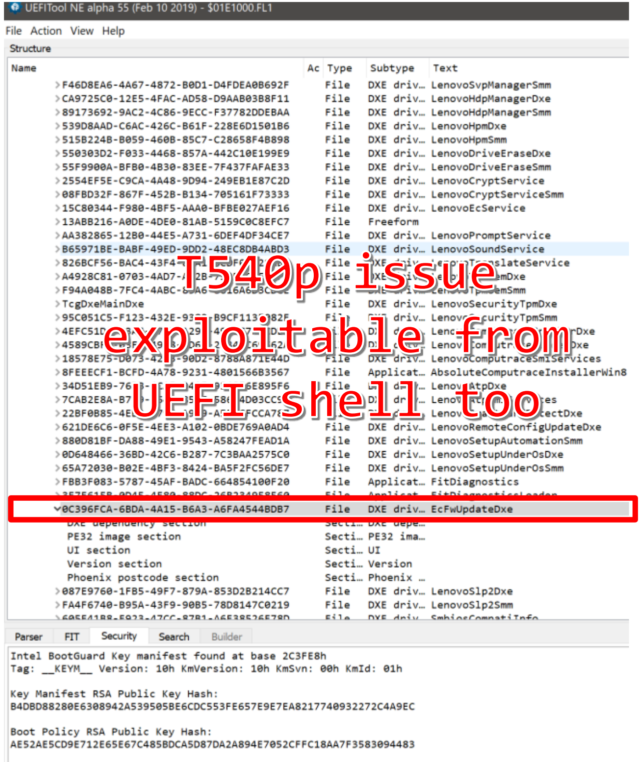
Host flash access not locked 😈

```
void write_flash_to_ec(unsigned int *flash_bufer)
{
    _outp(0x80, 0xC0);

    // writing EC flash block
    send_command_to_ec(0x06); // load flash block

    _outp(0x80, 0xC2);

    // point to buffer start.
    _outp(0x80, 0xC2);
    send_command_to_ec(0x07); // setup flash address

    unsigned int flash_block_start = 128 * 0x800; // flash_block size
    _outp(0x80, 0xC3)
    send_data_buffer_to_ec(flash_block_start & 0xFF);

    _outp(0x80, 0xC4);
    send_data_buffer_to_ec((flash_block_start >> 8 ) & 0xFF);

    _outp(0x80, 0xC5);
    send_data_buffer_to_ec((flash_block_start >> 16 ) & 0xFF);

    _outp(0x80, 0xC6);
    // writing EC flash block
    send_command_to_ec(0x08); // programm flash on EC
}
```

Boot Guard saves the day?

- ❑ 4th Intel Core generation
- ❑ Measure/verified boot
- ❑ “Hardware root of trust”
- ❑ Boot Guard coverage in the hand of OEMs





https://medium.com/@matrosov/bypass-intel-boot-guard-cc05edfca3a9

So can we just patch the EcFwUpdateModule again on P50?

Lenovo Thinkpad EC signature check

- EC update image mapped from OS update tool (TDK)
- Validate CRC16 checksum of EC image is correct
- Copy SecureFlash public key to EC related HOB
- Calculate RSA_verify(ECFW_signature, HOB_pulickey)
- IF signature correct: global sign_correct = TRUE;
- IF sign_correct == TRUE update EC firmware

Lenovo Thinkpad EC sig

- EC update image mapped from OS update tool (TDK)
- Validate CRC16 checksum of EC image is correct
- Copy SecureFlash public key to EC related HOB
- Calculate RSA_verify(ECFW_signature, HOB_pulickey)
- IF signature correct: global sign_correct = TRUE;
- IF sign_correct == TRUE update EC firmware

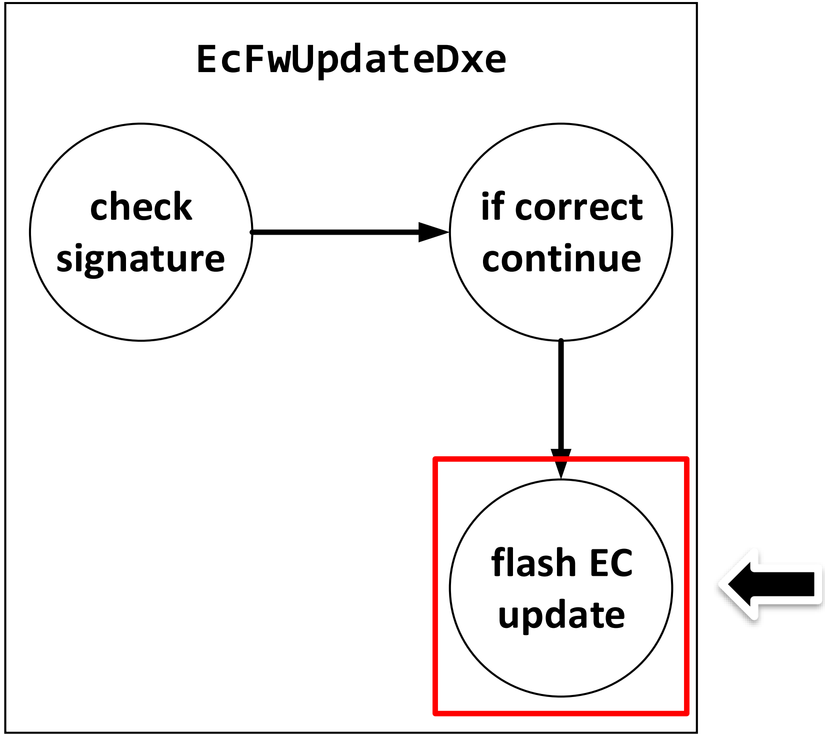
But what if separate verify and flash?

```
sig_hob = (&pHobList_->EfiMemoryBottom + 4);
tmp_buffer = 0i64;
qword_5EB0 = &pHobList_->EfiMemoryBottom + 4;
v11 = LODWORD(pHobList_->EfiMemoryBottom);
qword_5EA8 = LODWORD(pHobList_->EfiMemoryBottom);
buffer_fw = AllocatePool(HobType, sizeof(FW_BLOB));
if ( buffer_fw )
  buffer_fw = MemZero(buffer_fw, sizeof(FW_BLOB));
BUFFER_FW = buffer_fw;
if ( buffer_fw )
{
  if ( buffer_fw != FW_POOL2 )
  {
    MemCopy(buffer_fw, FW_POOL2, sizeof(FW_BLOB));
    buffer_fw = BUFFER_FW;
  }
  if ( (FW_SIZE - 0x10) > 0x45F10 )
  {
    do
    {
      pad_offset = pad_offset_++;
      buffer_fw->header.magic[pad_offset] = 0xFF;
    }
    while ( pad_offset_ < FW_SIZE - 0x10 );
  }
  sha2_context = AllocatePool_(sizeof(SHA_SVC_CTX));
  MemSet(sha2_context, sizeof(SHA_SVC_CTX), 0);
  if ( sha2_context != &CryptoSvcGuid )
    MemCopy(sha2_context, &CryptoSvcGuid, sizeof(GUID));
  [illegible]
    MemCopy(sha2_context->context, &SHA256_INIT, 0x20ui64);
  v15 = sha2_update(sha2_context, (BUFFER_FW + sizeof(FW_HEADER)), 0x46000ui64);
  res = 0i64;
  if ( v15 < 0 )
    res = v15;
  if ( res >= 0 )
  {
    sha2_final(sha2_context, &tmp_buffer);
    if ( !sig_hob
      [illegible]
      || (tmp_buffer = 0i64, LoadPubKeyHob(&tmp_buffer, v17, sig_hob))
      || (res = VerifySignWithPubKey(tmp_buffer, sha2_context, v18, 0x100i64)) != 0 )
    {
      res = EFI_SECURITY_VIOLATION;
    }
  }
```

get_ec_image_hash()

verify_ec_image_signature()

Lenovo P50 EC signature check flow



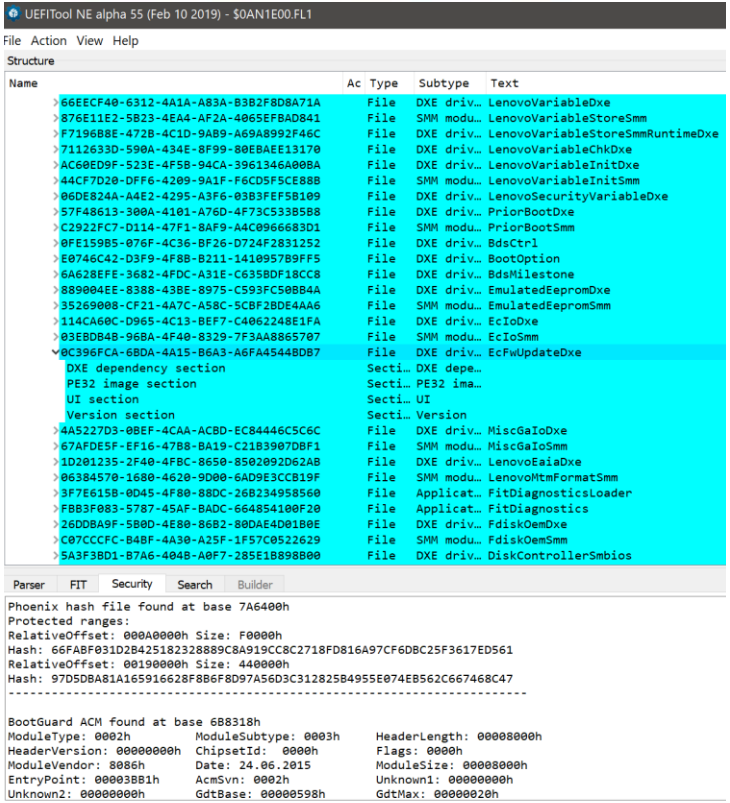
Obvious place for race condition (TOCTOU)?

Now, can we do the same attack with newer P50?

P50 try-harder

On Thinkpad P50 and newer:

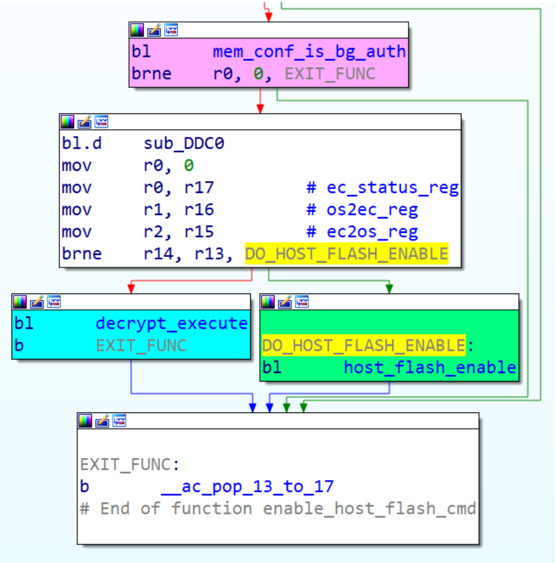
- ❑ Stronger coupling of security boundaries
- ❑ Boot Guard IBB hash coverage is better
- ❑ And…



P50 try-harder

Host flash access needs to be enabled by additional command to unlock 😈

- On the EC mem_conf_is_bg_auth check a status bit
- Set when the EC receives a magic value
- Shared secret between the BIOS and the EC



P50 try-harder

- ❑ Shared secret sent from the BIOS

```
op3 = 0x14;
op2 = 0xA;
op1 = 2;
*buffer = 0x6065845A;               // static unlock password
buffer[4] = 0x47;
buffer_size = 5;
LOBYTE(res) = EcIoDxeInterface->CpuIoCmdWriteBufferEC1(
                EcIoDxeInterface,
                *&op1,
                *&op2,
                *&op3,
                *&buffer_size,
                buffer);
```

Can we simply replay it? 😈

P50 try-harder

Nope, reduced window of opportunity with sanity check:

- EcFwUpdateModule sends a new command: 0xDF
- Lock the EC update in early BIOS
- Authentication no more possible on EC without reset

```
if ( HOB_TABLE->BootMode != BOOT_ON_FLASH_UPDATE )
{
  __outbyte(0x70u, 0x6Au);
  v6 = __inbyte(0x71u);
  __outbyte(0x70u, 0x6Au);
  __outbyte(0x71u, v6 & 0xBF);
  cmos_crc();
  LOBYTE(addr_read) = 0x3D;
  value_in = EcIoDxe->CpuIoCmdReadEC1(EcIoDxe, addr_read);
  LOBYTE(addr_write) = 0x3D;
  LOBYTE(value_out) = value_in | 0xDF;
  EcIoDxe->CpuIoCmdWriteEC1(EcIoDxe, addr_write, value_out);
}
```

Lenovo disclosure timeline

- 05/30 - Submit issue to Lenovo PSIRT
- 06/03 – Joint call with Lenovo PSIRT, answered questions and submit additional information
- 07/11 – CVE assigned for T540p report -> CVE-2019-6171
- 08/08 - Today is happy Disclosure day!

Lenovo Security Advisory:

https://support.lenovo.com/solutions/LEN-27764

Special thanks to Beverly Miller Alvarez from Lenovo PSIRT for her help in disclosure process!

EC take-aways

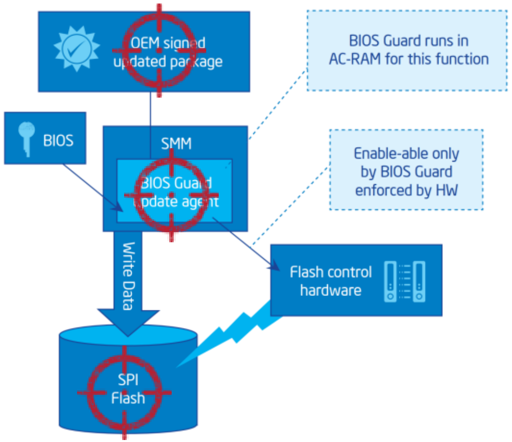
- ❑ Were looking for BIOS Guard ephemeral value auth
- ❑ Found static shared secret between BIOS and EC
- ❑ Can be abused in some scenario up to EC rootkit
- ❑ => No EC BIOS Guard ephemeral value support for these laptop lines (yet)

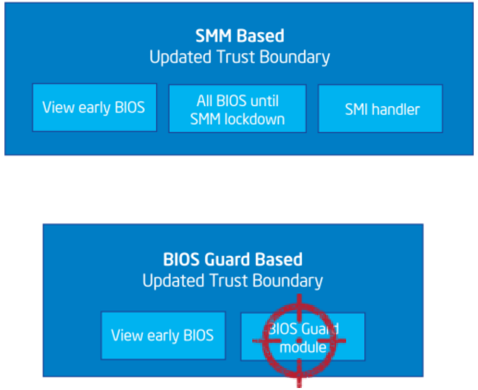
- ❑ Boot Guard does not fully protect from rogue update at runtime
- ❑ What does BIOS Guard would have change?

Deep dive into BIOS Guard

Intel BIOS Guard in a nutshell

- Rationale: BIOS security boundary is insufficient to protect critical code responsible for BIOS or EC firmware update
- Proposal: deport code to a safer environment: Authenticated Code Module RAM (ACM-RAM)





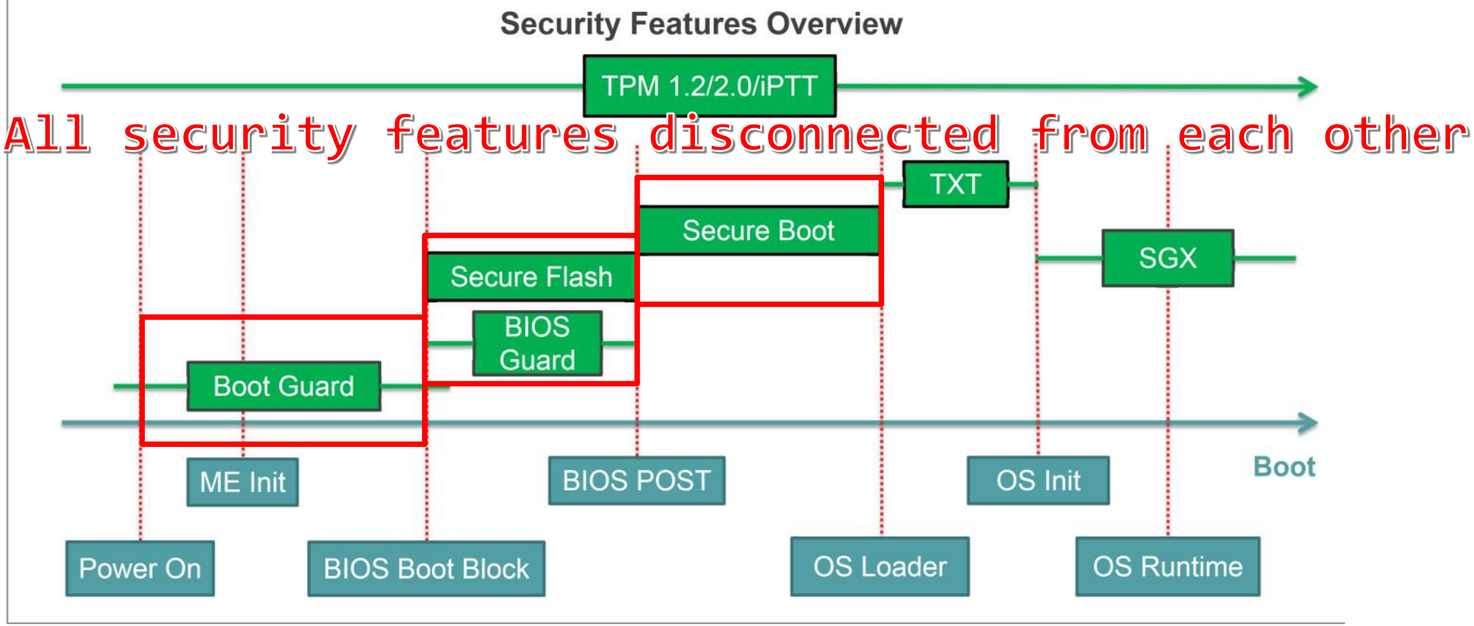
https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf

What is Intel BIOS Guard?

- Platform Flash Armoring Technology (PFAT)
- Armoring SPI Flash access
  - ✓ Access controlled by BIOS Guard ACM
  - ✓ Partially implemented in Microcode, PCH, BIOS and EC
  - ✓ PCH locked SPI flash access without PFAT
- BIOS update authentication
  - ✓ Authenticated by BIOS Guard ACM
- Game over for malicious updates?
  - ✓ Physical access + direct programming SPI flash still possible
  - ✓ POST update verification only relies on Intel Boot Guard integrity

LCFC联宝 Summary

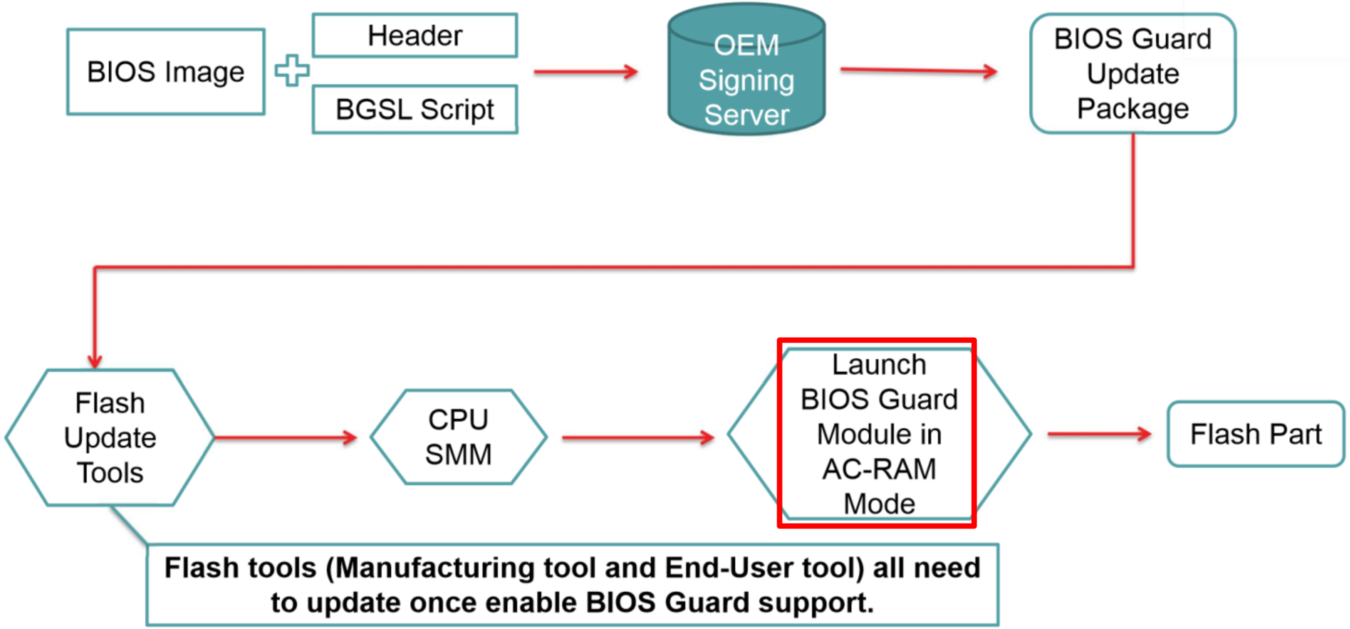
- Intel Security Features Diagram



https://wenku.baidu.com/view/f1d955c46bd97f192379e9aa

LCFC联宝 BIOS Guard Feature Overview

- Typical BIOS Update Process with BIOS Guard



https://wenku.baidu.com/view/f1d955c46bd97f192379e9aa

Lenovo Thinkpad PFAT update process

- ❑ Lenovo TDK update framework maps new BIOS image into memory
- ❑ Triggers BIOS Guard tool SMI over ACPI
- ❑ Sends BGUP memory address, BGUP size, IO Trap address
- ❑ BIOS Guard SMI sets BG directory, trigger MSR to load ACM
- ❑ ACM triggers Microcode flow to verify and apply BIOS Guard update and reboot machine

```
logout("Initialize Flash module.\n");
v0 = map_bios_update_to_memory(tdk_bin);
if ( v0 )
{
  v56 = 200;
  goto LABEL_364;
}
if ( v57 == 5 )
{
  v17 = UpdatePUPThroughPFAT(0x22u, flash_bios_image_from_memory, 0i64, 0);
  v0 = v17;
  if ( v17 )
  {
    v56 = v17;
  }
  else
  {
    logout("Going to update with PUP, this might take a while, please wait.\n");
    v0 = UpdatePUPThroughPFAT(0xCu, reboot_and_flash, &v58, 4u);
    if ( v0 )
    {
      v56 = 241;
    }
    else
    {
      logout("\nThe PUP is flashed through PFAT successfully.\n");
      v56 = 0;
    }
```

Resources

- Platform Firmware Armoring Technology (PFAT) patents
  US 2013/0219191 A1 & US 2012/0072734 A1

- Dell Firmware Security, 2018, Justin Johnson

https://www.platformsecuritysummit.com/2018/speaker/johnson/PSEC2018-Dell-Firmware-Security-Justin-Johnson.pdf

- Betraying the BIOS: Going Deeper into BIOS Guard Implementations, 2018, Alex Matrosov

https://github.com/REhints/Publications/blob/master/Conferences/Betraying%20the%20BIOS/Offensivecon_18%5Bv2.0%5D.pdf

- Cross-analysis of BIOS implementations:
  - Phoenix-based: Lenovo Thinkpad P50, T540
  - AMI-base: Gigabyte C246, Lenovo IdeaPad, Dell Inspiron

BIOS Guard at hardware (Intel) level

From now on, we focus on Lenovo P50 BIOS implementation:

- ❑ Phoenix-based
- ❑ Intel Skylake 6th generation processor

BIOS Guard hardware support

Interactions through a set of MSRs

- PLATFORM_INFO MSR (0CEh)

```
PLATFORM_INFO_MSR = __readmsr(0xCEu);
if ( PLATFORM_INFO_MSR & 0x800000000i64 )     // bit 35: BiosGuard feature available
{
```

- PLATFORM_FIRMWARE_PROTECTION_CONTROL (110h)

```
PLAT_FRMW_PROT_CTRL_MSR = __readmsr(0x110u);
if ( PLAT_FRMW_PROT_CTRL_MSR & 1 )          // bit0: BiosGuard Lock
{
  v17 = (PLAT_FRMW_PROT_CTRL_MSR & 2) == 0;// bit1: BiosGuard Enable
```

BIOS Guard hardware support

- PLATFORM_FIRMWARE_PROTECTION_EPHEMERAL (117h)
  - Early provisioning (PEI phase)
    - Module SiInit (Silicon Init)
    - Generate ephemeral value (RDRAND)
    - Send it to the EC but never used
    - Buried in hardware (MSR 117h)
    - Most probably Write-Only register
    - Discard value
  - Run-time: only BIOS Guard can unlock controllers (PCH/EC) using the ephemeral value

```
ephemeral_value = rdrand_safe();
shift = 0;
size = 4;
do
{
  EC0_cmd(ppi_F8D5438E_, 2, 0, ephemeral_value >> shift, 0);
  shift += 8;
  --size;
}
while ( size );
EC0_cmd(ppi_F8D5438E_, 3, 0, 0, &ec_status_out);
v2 = ec_status_out != 0;
writemsr_0x117(ephemeral_value);
```

BIOS Guard hardware support

- BIOS Guard Platform Data Table (BGPDT)
  - Platform specific, static, BIOS Guard configuration
- PLATFORM_FIRMWARE_PROTECTION_HASHx MSRs (111h-114h)
  - Early provisioning (PEI phase)
  - Set up BGPDT, compute its digest
  - Possibly write-once MSRs or locked depending on BG status
  - Immutable BGPDT then

```
__writemsr(0x111u, *bgpdt->sha2_digest);
__writemsr(0x112u, *&bgpdt->sha2_digest[8]);
__writemsr(0x113u, *&bgpdt->sha2_digest[0x10]);
__writemsr(0x114u, *&bgpdt->sha2_digest[0x18]);
LODWORD(bios_guard_status_) = bios_guard_status | 3;
__writemsr(0x110u, bios_guard_status_);
```

At this point (PEI phase, early boot) BIOS Guard configuration is set up and locked-down

BIOS Guard ACM execution flow

- PLATFORM_FIRMWARE_PROTECTION_TRIGGER_PARAM (115h)
  - Set up with a pointer on BIOS Guard Directory
  - Parameters for operations
  - Placeholder for the return value as well

- PLATFORM_FIRMWARE_PROTECTION_TRIGGER (116h)
  - BG "syscall" or trigger

```
__writemsr(0x115u, BiosGuardContext->BiosGuardDirectory);// set params
__writemsr(0x116u, 0i64);                               // trigger BG ACM module
BiosGuardContext->res = __readmsr(0x115u);             // read return value
```

BIOS Guard ACM

- File format close to Intel Boot Guard ACM
- Size 29-32k
- Signed and encrypted (most likely AES-CBC)
- Black box, expected to implement:
  - BGPTD hash verification
  - Update package signature check (optional)
  - Script interpreter
  - Flash SPI access and communications with the EC
- Provided by Intel to OEM as binary blob

BIOS Guard at software (OEM) level

BIOS Guard Directory

- Top-level structure
- Array of pointers (6)
- Address passed in MSR 115h
- ACM module and BGPDT, first exposed by PlaformInit HOB

```
struct BIOSGUARD_DIRECTORY {
    EFI_PHYSICAL_ADDRESS AcmModule;
    EFI_PHYSICAL_ADDRESS Bgpdt;
    EFI_PHYSICAL_ADDRESS UpdatePackage;
    EFI_PHYSICAL_ADDRESS Unknown0;
    EFI_PHYSICAL_ADDRESS Unknown1;
    EFI_PHYSICAL_ADDRESS Unknown2;
} bg_dir;
```

- Ored entries:
  - With 0xFE << 56 if not set
  - With index << 56 otherwise

```
BiosGuardContext->bg_dir.UpdatePackage = UpdatePackage;
BiosGuardContext->bg_dir.BgAcmModule = BgAcmModule;
BiosGuardContext->bg_dir.Bgpdt = Bgpdt | 0x100000000000000i64;
BiosGuardContext->bg_dir.UpdatePackage |= 0x200000000000000ui64;
BiosGuardContext->bg_dir.Unknown0 = 0xFE00000000000000ui64;
BiosGuardContext->bg_dir.Unknown1 = 0xFE00000000000000ui64;
BiosGuardContext->bg_dir.Unknown2 = 0xFF00000000000000ui64;
```

BIOS Guard Platform Data Table

```
struct BGPDT {
  unsigned int   TableSize;
  unsigned int   Unknown;
  unsigned char  Platform[16]; // Skylake
  unsigned char  PubKeyDigest0[32];
  unsigned char  PubKeyDigest1[32];
  unsigned char  PubKeyDigest2[32];
  unsigned int   Unknown;
  unsigned int   Unknown;
  unsigned int   Unknown;
  unsigned int   EcFlags;
  unsigned int   EcPortCmd;    // 0x66
  unsigned int   EcPortData;   // 0x62
  unsigned int   EcCmdExtra0;  // 0xB3
  unsigned int   EcCmdExtra1;  // 0xB4
  unsigned int   EcCmdExtra2;  // 0xB5
  unsigned int   EcCmdExtra3;  // 0xB6
  unsigned int   Unknown;
  unsigned int   NbRanges;

  struct SFAM_RANGE {
    unsigned int Start;
    unsigned int End;
  } ranges[ bgpdt.NbRanges ]
} bgpdt;
```

- Static configuration of the protection
  - EC IO ports, commands
  - Public keys digests
  - SFAM array: protected flash memory ranges
- Sealed at PEI phase

BIOS Guard Platform Data Table

- SFAM ranges
- Protected range of flash regions => only accept signed operations
- Regions can be found in the _FLASH_MAP structure

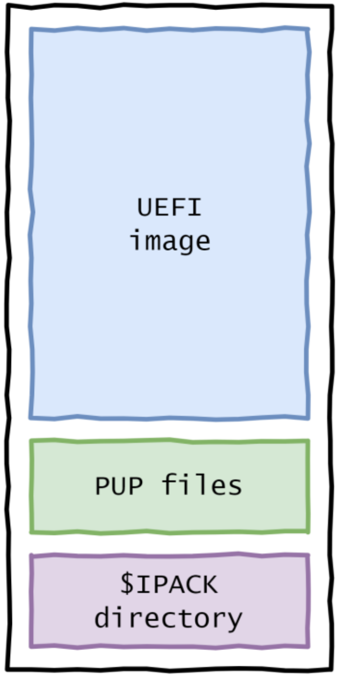
```
bg_hob->bgpdt.field_7C = 0x53000;
bg_hob->bgpdt.SfamRanges[4].End = 0xFFFFFFFF;
bg_hob->bgpdt.SfamRanges[0].Start = 0xFF8A0000;
bg_hob->bgpdt.SfamRanges[0].End = 0xFF98FFFF;
bg_hob->bgpdt.SfamRanges[1].Start = 0xFF990000;
bg_hob->bgpdt.SfamRanges[1].End = 0xFFDCFFFF;
bg_hob->bgpdt.SfamRanges[2].Start = 0xFFDD0000;
bg_hob->bgpdt.SfamRanges[2].End = 0xFFDFFFFF;
bg_hob->bgpdt.SfamRanges[3].Start = 0xFFEC0000;
bg_hob->bgpdt.SfamRanges[3].End = 0xFFFDFFFF;
bg_hob->bgpdt.SfamRanges[4].Start = 0xFFFE0000;
bg_hob->bgpdt.SfamRanges[5].Start = 0xFF89D000;
bg_hob->bgpdt.SfamRanges[5].End = 0xFF89DFFF;
bg_hob->bgpdt.SfamRanges[6].Start = 0xFFEB0000;
bg_hob->bgpdt.SfamRanges[6].End = 0xFFEBFFFF;
bg_hob->bgpdt.NbRanges = 6;
bg_hob->bgpdt.size = 0xE0;
```

BIOS Guard Update Package

- Operation parameters for the BIOS Guard ACM
  - Header (platform, versions, signature requirement, *etc.*)
  - Script: dynamic or templated
  - Buffer to be written in flash
  - Cryptographic material (signature)

- Templated scripts for signed/protected operations
  - $IPACK structure in Lenovo's image
- Dynamically generated scripts
  - BiosGuardService API (wrapped into BIOS_GUARD_PROTOCOL)

$IPACK structure




```
struct IPACK_VOLUME {

    struct IPACK_HEADER {
        unsigned char Magic[6]      <bgcolor=cBlue>; // $IPACK
        unsigned char Reserved[2];
        unsigned int  VolumeSize    <bgcolor=cWhite>;
        unsigned int  FilesCount    <bgcolor=cPurple>;
        unsigned char Reserved2[0x200];
    } header;

    struct IPACK_FILE {
        unsigned char Name[0x100] <bgcolor=cGreen>;
        unsigned int RawOffset    <bgcolor=cRed>;
        unsigned int RawSize      <bgcolor=cAqua>;
        unsigned char Flags       <bgcolor=cYellow>;
        unsigned char Reserved[3];
        unsigned int  Unknown;
    } files[ volume.header.FilesCount ];

} volume;
```

$IPACK files

- _IMG_.ORG: main UEFI image (0x88E350 bytes)
- PUPHEAD.BIN: update header (0x30 bytes)
- PUPDUMMYHEAD.BIN
- PUPSCRP.BIN: update script (0xD0 bytes)
- PUPDUMMYSCRP.BIN
- PUPCERT.BIN: certificate (0x20C bytes)
- PUPDUMMYSIGN.BIN
- PUPSIGN.BIN: signatures collection (0x6C000 bytes)

```
res = BgFindPupHead(&bPupHeadPresent);
if ( res )
  return res;
if ( bPupHeadPresent )
{
  res = IPackFileRead("PUPHEAD.BIN", &buffer_PUPHEAD, &pup_sizes.puphead_size);
  if ( res )
    return res;
  res = IPackFileRead("PUPSCRP.BIN", &buffer_PUPSCRP, &pup_sizes.pupscrp_size);
  if ( res )
    return res;
  res = IPackFileRead("PUPCERT.BIN", &buffer_PUPCERT, &pup_sizes.pupcert_size);
  if ( res )
    return res;
  res = IPackFileRead("PUPSIGN.BIN", &buffer_PUPSIGN, &pup_sizes);
  if ( res )
    return res;
}
```

PUPCERT.bin

```
Edit As: Hex   Run Script   Run Template: pupcert.bt
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000h: 01 00 00 00 01 00 00 00 3B F4 3E 74 37 BB 5D 96
0010h: 77 1B 3E F6 4A 3B 45 82 9A 86 9F CA D5 88 6E DC
0020h: A7 67 35 5E 0F BA 58 A0 54 29 5F 37 45 27 B9 8B
0030h: 57 84 C7 3C D5 9C 02 C2 AE 54 47 32 39 55 22 CB
0040h: 19 CE 35 14 49 7B BE 26 B2 48 D8 EF 1B 1E F8 C5
0050h: 8D 62 AE F6 F6 9F 49 E3 33 62 9E 48 77 6B C2 B6
0060h: A3 A4 8B 7D ED 85 7E 6B 07 EF 7C 18 91 58 EB 50
0070h: EE 67 00 68 17 32 4F 8C FE 14 CE 82 6B BC B0 E9
0080h: 1E FF 18 60 45 FB 06 8A 79 58 7B 4E 48 1A ED 46
0090h: 92 CF D2 5A 97 34 60 0B 76 D4 7D EB 34 3B D6 3A
00A0h: 7B FA 4C 98 FE 6E 46 89 A5 52 E0 57 B8 8E 63 34
00B0h: 36 08 F5 D0 62 A0 49 06 C3 F3 4F BF 3B F6 1C E8
00C0h: 66 2E 65 9A 93 99 A6 11 60 41 5F 59 A8 43 33 63
00D0h: E0 13 8B 78 60 BB 7B 6C B9 D1 2B 5A 75 BC 3E EA
00E0h: EC 59 E0 46 F8 32 50 7F 9F 51 5B C9 C1 4E 65 E2
00F0h: D7 7F A9 0D 22 E5 98 47 50 E1 2C 5A 0A 60 C7 1A
0100h: 48 A0 4C 42 34 46 8E D0 01 00 01 00 00 00 00 00
0110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0120h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0140h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0180h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0190h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01C0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01D0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01E0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0200h: 00 00 00 00 00 00 00 00 00 00 00 00
```

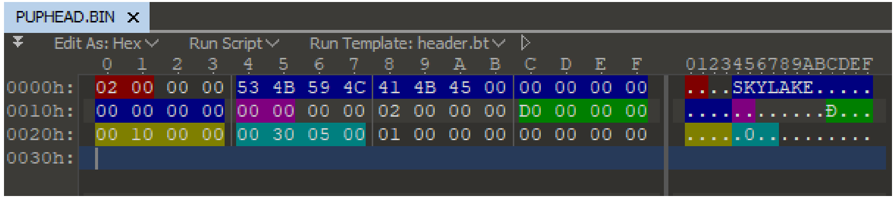
- ❑ Cryptographic material
- ❑ Template file
- ❑ RSASSA-PKCS1-v1_5, SHA2
- ❑ For each signed operation, chunk signature is written over the placeholder

```
struct PUBCERT_BIN {
    unsigned int  PubKeyType <bgcolor=cGreen>; // guess, 1 => 2048bits
    unsigned int  SigType    <bgcolor=cBlue>; // guess, 1 => 2048bits
    unsigned char PubKey[0x100]         <bgcolor=cYellow>;
    unsigned int  Exponent              <bgcolor=cRed>;    // 0x10001
    unsigned char SigPlaceholder[0x100] <bgcolor=cWhite>;
} pupcert;
```

PUPHEAD.bin

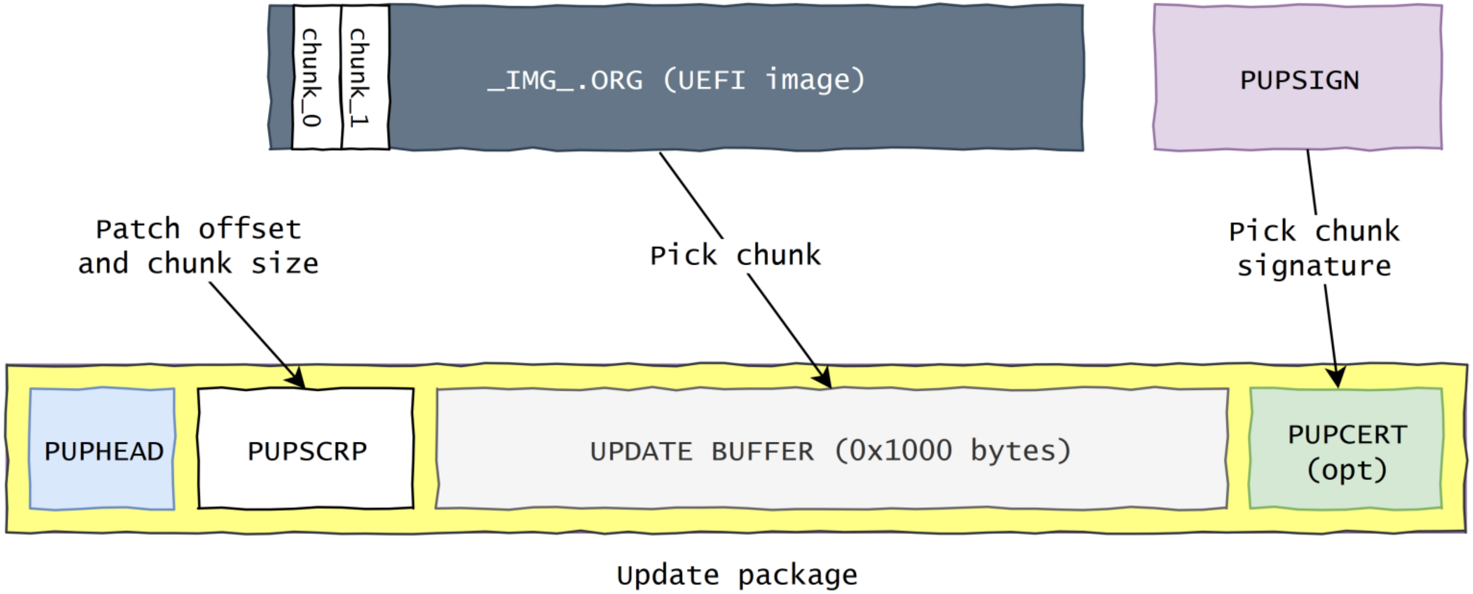
Operation header:

- **Flags**: a bit is set to require a signed operation
- **Platform**: should match the one from BGPDT



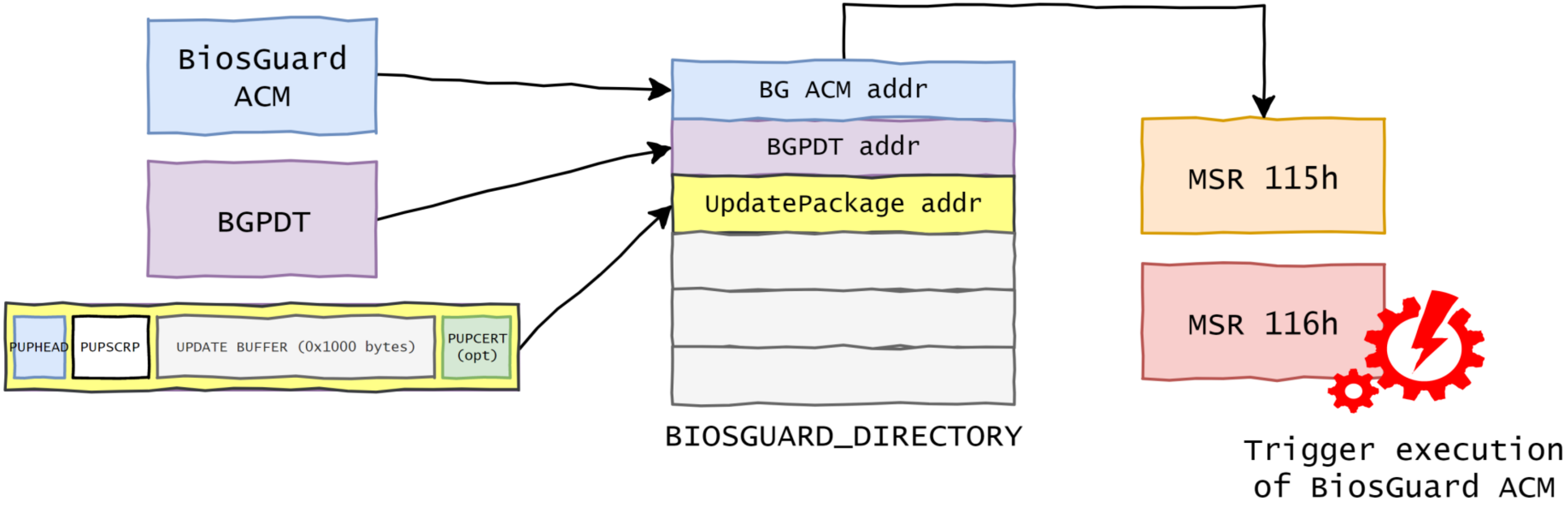

```
struct PUPHEAD_BIN {
  unsigned short Version;
  unsigned char  Unknown[2];
  unsigned char  Plaform[16];
  unsigned short Flags;
  unsigned char  Unknown2[2];
  unsigned int   Unknown3;
  unsigned int   ScriptSize;
  unsigned int   Chunksize;
  unsigned int   FwSvn;
  unsigned int   EcSvn;
  unsigned int   Unknown4;
} pupheader;
```

BIOS Guard update package



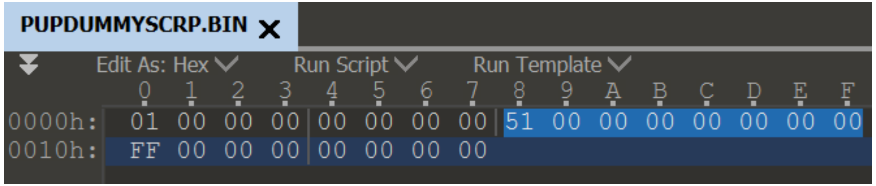
SystemFlashUpdateDriverDxe debug string: "../../Lib/Common/PfatPupRomWrite.c"

BIOS Guard operation



BIOS Guard scripting

- Fixed size instruction set (8 bytes)



- Few instructions guessed:
  - OP_START = 01 00 00 00 00 00 00 00
  - OP_END = FF 00 00 00 00 00 00 00
  - OP_SET_FLASH_ADDR = 55 00 00 00 XX XX XX XX
  - OP_FLASH_ERASE = 14 00 00 00 00 00 00 00
  - OP_FLASH_WRITE = 11 00 00 00 00 00 00 00

- Interpreter expected to be in the ACM module or Microcode

BIOS Guard scripting

- Generated dynamically (unsigned operations)
  - Very basic scripts (4 instructions)
  - Ex: OP_START | OP_SET_FLASH_ADDR | OP_FLASH_WRITE | OP_END
- PUPSCRP.bin used as a template (signed operations)
  - 26 instructions program
  - Patch flash address in 2nd instruction operands
  - Patch chunk size in 3rd instruction operands
- Only signed operations can write/erase SFAM ranges (ERR_SFAM_VIOLATION otherwise)

Open questions

- SHA2 of public key is expected in BGPDT
  - Same digest values for P50 and T540
  - Could not recompute the value
- Chunks signature:
  - RSASSA-PKCS1-v1_5 signature, SHA2 digest
  - Unsure about the scope of the signature
  - Whole update package?
- Unsigned operations
  - Interpreter in ACM exposes a rather large attack surface
  - Fuzzing?

Notes for future research

- Interesting error codes:

  "ERR_UNSUPPORTED_CPU", "ERR_BAD_DIRECTORY",
  "ERR_BAD_BGPDT", "ERR_BAD_BGUP",
  "ERR_SCRIPT_SYNTAX", "ERR_UNDEFINED_FLASH_OBJECT",
  "ERR_UNEXPECTED_OPCODE", "ERR_BAD_BGUPC",
  "ERR_UNSIGNED_B0_STORE", "ERR_RANGE_VIOLATION",
  "ERR_SFAM_VIOLATION", "ERR_EXEC_LIMIT", *etc.*

Experiments

ACM FUN

- Tried debug over Intel DCI to access ACM memory and dump decrypted BIOS Guard ACM => no success ☹
- Replace BIOS Guard ACM module with older one from another platform => temporarily bricked a laptop (need reflash)
- Remove ACM from update image before flash over OS updater => start loop of weird reboots on S3, after few recover to previous version

Conclusions

Conclusions

- Complex feature:
  - Hardware support, but…
  - Many software components (PEI, SMM, DXE)
  - Specific format for BIOS image
- Strong dependency of OEM vendors to Intel (BIOS Guard ACM)
- Lenovo’s EC support still limited?
- Could possibly support other firmware's as well?
- Many implementation details in the hands of OEM
  => room for misconfiguration

BIOS Guard implementation checklist

- ❑ SFAM regions coverage don’t have obvious mistakes
- ❑ Signed vs unsigned operations with BIOS Guard script
- ❑ Communications between BIOS and EC implemented correctly (not static session password)
- ❑ Recovery process implemented without supply chain backdoors

Shout-out

- All friends who shared enlightening thoughts with us, you know who you are ☺
- Igor and Ilfak for outstanding IDA’s support
- @AirbusSecLab for the review and feedback
- Darrell Hut from NVIDIA for disclosure process support and help
- Rodrigo Branco (bsdaemon) from Intel for feedback

Thank you

Q&A