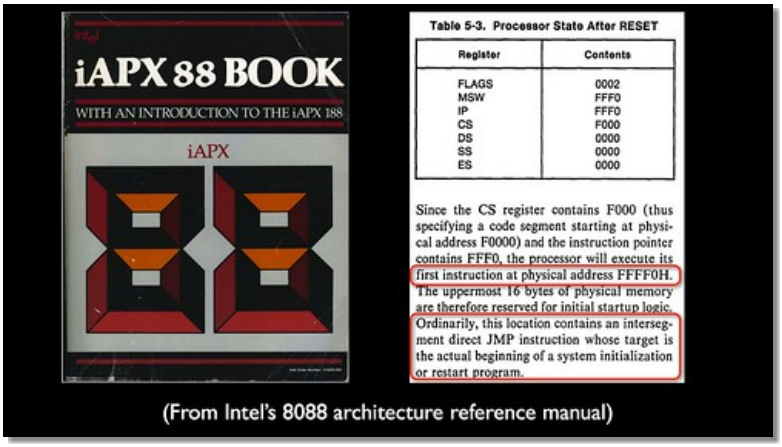
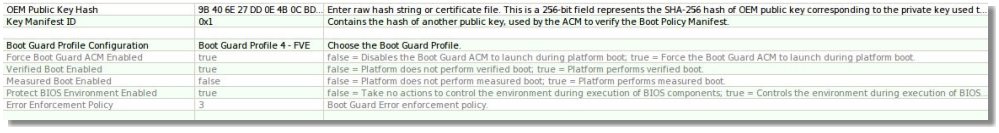


This page attempts to document what I've found about the modern era of CPU bootstrap and how we can use it to make slightly more secure systems. Most discussion of the boot process assume that the CPUs startup exactly the same as they were in the 1970s, which is only sort of true.

The legacy reset vector at 0xF:FFF0 is no longer the first instruction the x86 CPU executes. Instead the Intel Management Engine (ME) arranges for the CPU microcode to validate and execute an "Authenticated Code Module" (ACM) from the Firmware Interface Table (FIT). Once all of the ACMs in the FIT have run, the legacy reset vector is invoked and the system startup proceeds like its 1979.

Contents

Bootguard



OEM Public Key Hash	9B 40 6E 27 DD 0E 4B 0C BD..	Enter raw hash string or certificate file. This is a 256-bit field represents the SHA-256 hash of OEM public key corresponding to the private key used t...
Key Manifest ID	0x1	Contains the hash of another public key, used by the ACM to verify the Boot Policy Manifest.
Boot Guard Profile Configuration	Boot Guard Profile 4 - FVE	Choose the Boot Guard Profile.
Force Boot Guard ACM Enabled	true	false = Disables the Boot Guard ACM to launch during platform boot; true = Force the Boot Guard ACM to launch during platform boot.
Verified Boot Enabled	true	false = Platform does not perform verified boot; true = Platform performs verified boot.
Measured Boot Enabled	false	false = Platform does not perform measured boot; true = Platform performs measured boot.
Protect BIOS Environment Enabled	true	false = Take no actions to control the environment during execution of BIOS components; true = Controls the environment during execution of BIOS...
Error Enforcement Policy	3	Boot Guard Error enforcement policy.

To be written. FITC, Profiles, CPU fuses, PCH straps, ME interaction.

When the OEM receives the CPU, the ME is still in "Manufacturing Mode" and runs a special part of the firmware that will copy the "OEM Public Key Hash" and "Boot Guard Profile Configuration" policy values from its section of the flash ROM into the CPU field programable fuses (FPF) so that they are permanent and unchangable. It then sets a fuse to indicate that it has exited from manufacturing mode so that this portion of the firmware will not run again.

The "OEM Public Key Hash" is a SHA-256 hash of the public key that is used to verify the signature on the Bootguard Key Manifest, which has the public keys used to sign the "BIOS-SM" ACM and other flash files. The Boot Guard FPF has four policy settings:

- Force Boot Guard ACM Enabled: if set, there *must* be an OEM signed ACM in the FIT

- Verified Boot Enabled: if set, the platform will do a "verified boot", not sure yet exactly what that entails. Thinkpads set this bit.
- Measured Boot Enabled: if set, the platform will do a "measured boot", which I believe means *setting* PCR0 to the literal value "0x3", then extending it with the normal hash of the boot block, the ACMs and the next stages. Thinkpads do not set this bit.
- Protect BIOS Environment Enabled: if set, I believe this means that the ME will copy the IBB into the CPU cache so that it runs in a "cache-as-RAM" mode and has all DMA disabled to prevent devices from being able to modify it.

These values can be adjusted in the flash image with the Intel Flash Image Tool (FITC), but unless you have a way to force the ME into manufacturing mode then the values in the flash image are ignored. Since it is the public key hash, instead of the public key, corrupting it won't yield a factorizable key.

Based on what I've heard, but not seen documented, the verification of the OEM public key hash and the OEM signature on the FIT is done by the CPU microcode before it comes out of reset. I need to find this documented somewhere.

Firmware Interface Table

The FIT structure is defined in edk2's FirmwareInterfaceTable.h and the FIT pointer is usually at 0xFFFFFFC0 in the ROM. Verifying that it points to the table is easy since it starts with the header '_FIT_ '. It contains entries for the microcode updates, boot policys, TXT policy and other things. The one that we are most concerned with is the STARTUP_ACM.

```
00e1ce00: 5f 46 49 54 5f 20 20 20 0a 00 00 00 00 01 80 20  _FIT_   .......
00e1ce10: 00 22 df ff 00 00 00 00 00 00 00 00 01 01 00  ."..............
00e1ce20: 00 66 df ff 00 00 00 00 00 00 00 00 01 01 00  .f..............
00e1ce30: 00 aa df ff 00 00 00 00 00 00 00 00 01 01 00  ................
00e1ce40: 00 ea df ff 00 00 00 00 00 00 00 00 01 01 00  ................
00e1ce50: 00 42 e0 ff 00 00 00 00 00 00 00 00 01 01 00  .B..............
00e1ce60: 00 00 e2 ff 00 00 00 00 00 00 00 00 01 02 00  ................
00e1ce70: 00 00 ed ff 00 00 00 00 30 01 00 00 01 07 00  ........0......
00e1ce80: 00 d0 e1 ff 00 00 00 00 41 02 00 00 00 01 0b 00  ........A.......
00e1ce90: 00 e0 e1 ff 00 00 00 00 bb 02 00 00 00 01 0c 00  ................
00e1cea0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

This table decodes (note that the length fields are in counts of 16-bytes each):

```
1: address ffdf2200 @ 00000000: ver 0100 type Microcode (0x01)
2: address ffdf6600 @ 00000000: ver 0100 type Microcode (0x01)
3: address ffdfaa00 @ 00000000: ver 0100 type Microcode (0x01)
4: address ffdfea00 @ 00000000: ver 0100 type Microcode (0x01)
5: address ffe04200 @ 00000000: ver 0100 type Microcode (0x01)
6: address ffe20000 @ 00000000: ver 0100 type Startup ACM (0x02)
7: address ffed0000 @ 00130000: ver 0100 type BIOS Startup Module (0x07)
8: address ffe1d000 @ 00002410: ver 0100 type Key Manifest (0x0b)
9: address ffe1e000 @ 00002bb0: ver 0100 type Boot Policy Manifest (0x0c)
```

More information on the Intel Microcode update format is available, although we're treating it as a black box. As long as it is measured, we don't care too much for now.

Startup ACM

The ACM format is defined in the Intel TXT Software Development Guide (section A.1). The Startup ACM doesn't have a length in the FIT table,be encoded in the entry itself at offset 0x18. The entry point offset into the ACM is at offset 0x34. and there appears to be a 512-byte public key at offset 0x80 with an exponent of 17 (0x11) and a 512-byte signature. This does not appear to match the hash in the boot guard profile, but the same 512-byte key appears in the Boot Policy in the FIT (at offset 0x2080).

0x00: 0001 module type
0x02: 0002 module subtype
0x04: 000000a1 header len in quad-words = 0x284 bytes
0x08: 00000000 header version (0 for pre-2017 ACM, 3 for newer)
0x0C: 0000 chipset id
0x0D: b002 module flags
0x10: 00008086 vendor
0x14: 20160818 date code
0x18: 00008000 module size in quad-words == 128 KB
0x1C: 0000 TXT security version number
0x1D: 0000 SGX security verson number
0x20: 00000000 CodeControl ("Authenticated code control flags")
0x24: 00000000 Error Entry Point
0x28: 00000020 GDT limit ("defines last byte of GDT")
0x2C: 00001274 GDT base pointer offset
0x30: 00000008 Segment selector initializer
0x34: 0000ab39 Entry point offset from header
0x38: 00000000 Reserved2 (0x40 bytes)
0x78: 00000040 Keysize n quad words == 256 bytes
0x80: .... RSA Public Key (256 bytes)
0x180: 00000011 RSA exponent (not present in version 3?)
0x184: .... RSA signature (256 byte)

This code appears to run in 32-bit mode with references to a stack. The ME might enable "cache-as-RAM" mode, which allows the CPU to use the cache in non-writeback mode as if it were RAM, long before the memory controllers have been brought online. It also appears to use very low-value memory addresses for its data segment, possibly indicating that it is relocated to 0x0 in RAM, although the GDT referenced in the header might allow this to be relocated elsewhere. The specification says that "since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX."

In the one that I've analyzed there are SHA1 and SHA256 implementations (easily spotted due to their constants 0x67452301 and 0x06a09e667), and it appears to talk directly to the TPM at address0xfed40000 and interacts with the TXT private configuration registers at 0xfed20000.

One of the routines reads from0xfed20328, which I believe is the ACM STATUS REGISTER mentioned (but not documented) in of the Intel TXT Software Dev Guide(footnote on page 80 mentions a (non-public?) document):

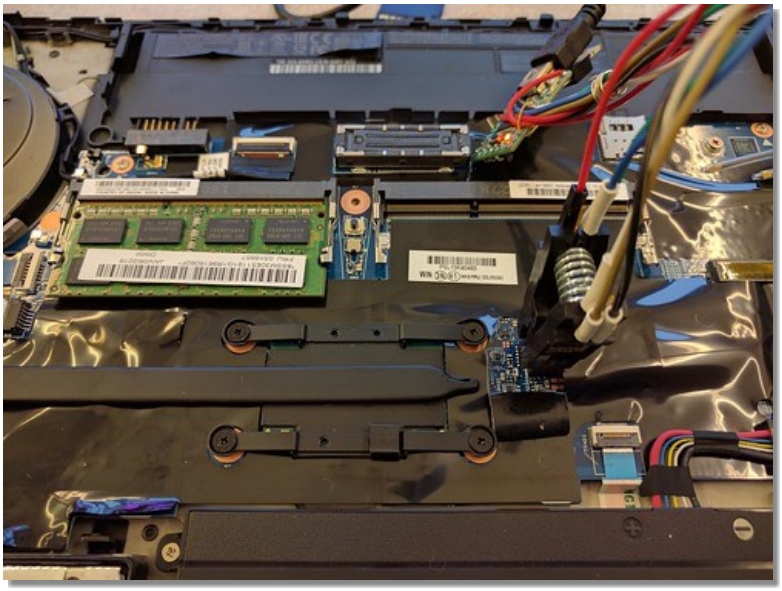
> Three new registers have been defined in TXT device space: [...] ACM_STATUS_REGISTER at offset 0x328 carries pre-boot error code information instead or ERRORCODE register. ERRORCODE register is left to exclusive use of SINIT and CPU uCode; BOOTSTATUS register at offset 0xA0 caries status flags of Startup function execution. Detailed information of layout of these registers can be found in “Converged Boot Guard and Intel Trusted Execution Technologies. BIOS Specification for Client CNL and Server ICX platforms”

When it is done I believe it callsGETSEC[EXITAC] (%eax=3), which will shutdown the ACM and jump to a near pointer to resume normal startup. There is a lengthy function that finds this address

Early device memory mappings (copied fromIntel's "BIOS Memory Map"):

PCI Express: 0xE0000000 - 0x10000000
IO APIC: 0xFEC00000 - 0x1000
HPET: 0xFED00000 - 0x400
MCH BAR: 0xFED10000 - 0x8000
DMI BAR: 0xFED18000 - 0x1000
EGP BAR: 0xFED19000 - 0x1000
RCBA: 0xFED1C000 - 0x4000
Intel TXT priv: 0xFED20000 - 0x10000 (private config registers)
Intel TXT pub: 0xFED30000 - 0x10000 (public status registers)
TPM: 0xFED40000 - 0x5000
TPM locality x: 0xFED4x000 - 0x1000 (one page for localities 0-4)
Intel PTT: 0xFED70000 - 0x1000
Intel VTd: 0xFED90000 - 0x4000
Local APIC: 0xFEE00000 - 0x100000
Flash: 0xFF000000 - 0x1000000

BIOS Startup Module



The "BIOS Startup Module" appears to be a normal UEFI FVH structure. There is probably a signature on the end, I haven't yet checked but the FITC tool says that this is supposed to be signed. This module is around 1.2MB and includes the usual list of PEI, S3 and SMM modules. It also includes the TPM driver, as well as graphics, USB, etc.

The one that I've examined (t550) starts at `0xFFED0000` and covers all the way to the top of 4GB, which includes the legacy reset vector at `0xF:FFF0`. It is not clear to me where it is signed nor where the signature is checked, although it is definitely hashed and measured by the Startup ACM prior to being invoked.

The BIOS ACM has a public key that matches the startup ACM and what appears to be a signature, but it does not cover the entire module. It appears to only cover the `0x18c80` bytes starting at `0xFFFE0000`, which does not include the legacy reset vector.

However, modifying the legacy reset vector prevents the system from starting, so it is included in a signature or hash somewhere that has yet to be determined.

Bootguard status

It is possible to see if Bootguard is enabled by reading the MSR:

sudo modprobe msr
sudo rdmsr 0x13a

A value of 0 indicates that it is not provisioned. The other values come from `intelmetool` in coreboot:

#define BOOTGUARD_DISABLED 0x400000000
#define BOOTGUARD_ENABLED_VERIFIED_MODE 0x100000000
#define BOOTGUARD_ENABLED_MEASUREMENT_MODE 0x200000000
#define BOOTGUARD_ENABLED_COMBI_MODE 0x300000000

Chain of Trust

- Intel's ME public key hash on die
- Intel's ME public key stored in flash
- Intel's ucode symmetric key (burned into the CPU?)
- Intel's ucode public key hash (burned into the CPU?)
- Intel's ucode public key stored in ucode file in the flash
- Intel's ACM public key hash (in CPU and updatable in microcode?)
- Intel's ACM public key in the ACM header
- OEM Key Manifest Security Version Number (KMSVN, stored in fuses in PCH)
- OEM root public key hash (OTP fuses in the PCH, delivered from ME to x86)
- Bootguard Key Manifest (with KMSVN to protect against rollback) signed by OEM public key
- OEM root public key in the Bootguard Key Manifest in the flash
- OEM per-model public key stored in Bootguard Key Manifest
- Bootguard Boot Policy, signed by OEM per-model public key
- Bootguard region hashes, stored in Bootguard Boot Policy
- UEFI Platform Key (PK) / Key Exchange Key (KEK) / Database keys (DB) / Revocation Database (DBX)

As soon as power is available, the ME boots up from its on-die boot ROM, checks some straps and fuses to determine its configuration, and typically then copies the flash partition table from the ME region of the flash to its on-die SRAM (although ROMB fused chips have limited security). The boot ROM locates the FTPR partition and copies it from the SPI flash into the on-die SRAM. It then checks that the SHA-1 hash of the key stored in the parition manifest matches the one in its on-die ROM and validates the RSA signature on the rest of the partition manifest. The partition table contains hashes of each of the modules in the partition, allowing the modules to be validated after they are copied into the on-die SRAM for execution (Intel ME Flash File System Explained - Sklyarov et al, BH-EU 2017).

When the ME boots the x86 CPU, the on-die microcode fetches the FIT pointer at 0xFFFFFFC0, which points to the FIT table somewhere in the SPI flash image. This table is then searched for microcode updates that match the CPU ID. The current microcode copies them from flash in a linear fashion (into L3 cache?) and decrypts with an on-die symmetric AES key (Chen & Ahn 2014), then validates with (on-die?) RSA key. It's likely that these ucode updates also contain the key hashes for the ACM.

Next the x86 ucode goes back to the FIT to find the Startup ACM and does an odd copy of it into L3 (looks like multiple hyperthreads are copying 4KB chunks?). The ACM contains an RSA public key; the ucode compares it against either an on-die key (hash?) or one stored in the ucode update and halts the CPU if it does not match. The ucode then checks the signature on the ACM and again halts if it does not match.

The Startup ACM is supposed to run entirely out of L3, although there have been bugs ([[TOCTOU|Bosch & Hudson, 2019]) that allowed a proximate attacker to use devices like the Spispy to detect accidental cache misses and TOCTOU the firmware. The ACM receives the OEM public key hash and Bootguard Profile from the ME via MSR. (Can't remember the order of operations here),

The ACM reads the BootGuard Key Manifest from the SPI flash (pointed to by the FIT and identified by__KEYM__) into L3 and hashes the RSA public key stored in it. If it doesn't match the OEM public key hash or if the OEM public key signature on the Key Manifest or if the stored KmSvn isn't right, the ACM takes action based on the Bootguard Profile bits. If it does match, it locates the Bootguard Policy in the FIT (and identified by __ACBP__) and copies it into L3. The ACM then computes the hash of the RSA public key in the Policy and compares it to the SHA256 hash stored in the Key Manifest. If that fails to match, or if the RSA signature on the Policy doesn't match, then the ACM again takes action based on the Profile settings.

The ACM uses the now validated Bootguard Policy structure to read the Initial Boot Block (IBB) segments into L3, hashing them as they are copied (and hopefully not causing any cache misses). If this computed hash doesn't match the "IBB Digest" in the Policy, the ACM takes action based on the Profile settings.

Resources

- Who watches the BIOS watchers? (Alex Matrosov, 2017)
- Intel TXT lab handout
- mjg59: Intel Boot Guard, Coreboot and user freedom (2015)

Categories: 2017 Security