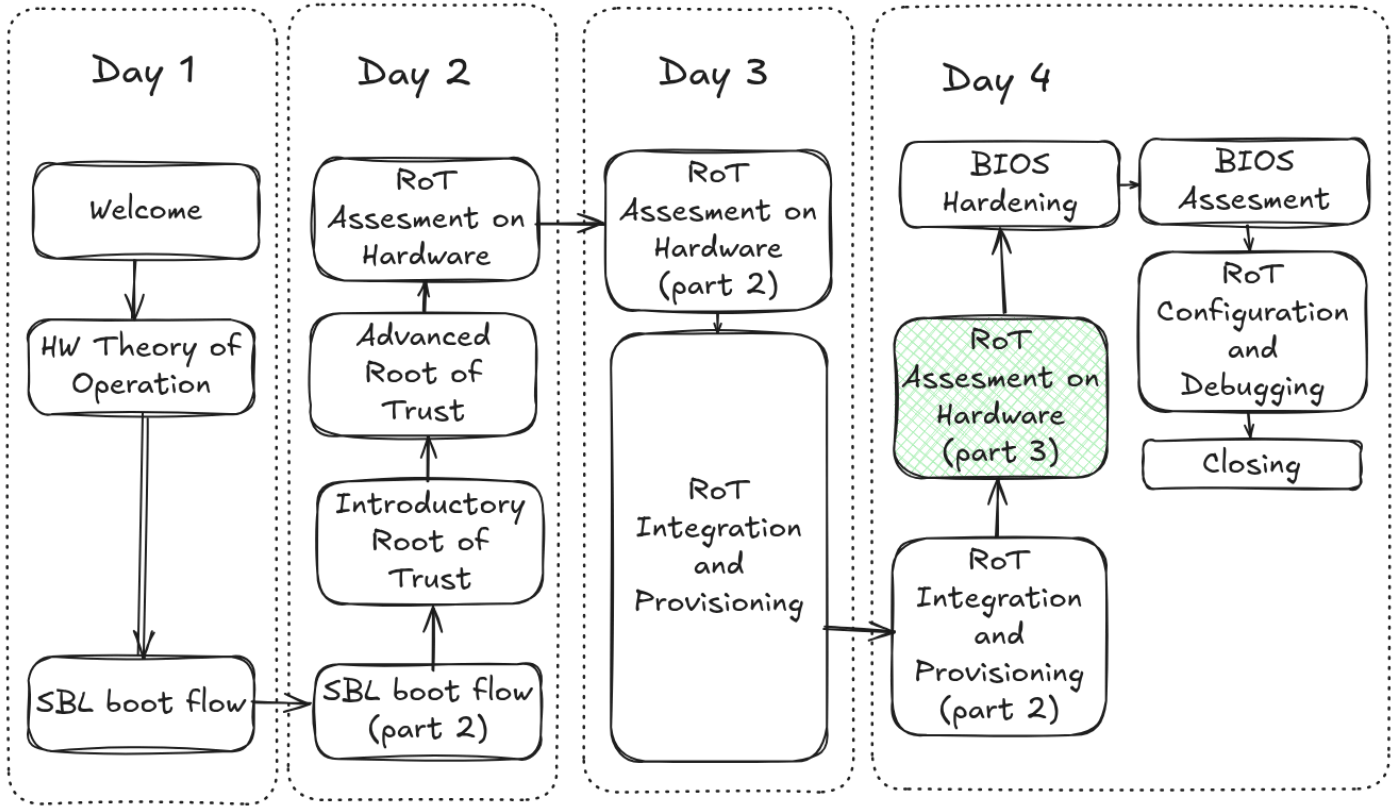
# DS09SBL: Dasharo TrustRoot Training

# Intel ME Configuration and FPF fuses

Root of Trust assessment on Hardware (part 2)

# Where we are in the course



# Goals of the Presentation

In this presentation, we aim to:

- Explore techniques for investigating ME configuration and fuses
- Describe and explain the most important fuses and configuration variables
- Find alternative ways for provisioning Root of Trust

# Disclaimer

- Information included in this presentation that comes from official Intel documentation will be marked on slides with an appropriate source.
  - Keep in mind that certain documents are old and may not be available anymore. That is why it is important to have own backups.
- Otherwise, it should be assumed that information was obtained in another way (leaks, experiments, binary behavioral and static analysis, researchers' findings). Sources of such knowledge will be provided if applicable.
- Even with all the sources of information, it is hard to get a big picture, which variables/fuses are responsible for what feature. Sometimes it is a guess, sometimes it is determined by cross-validating the outcome of modifying specific settings.

# Disclaimer

- The Management Engine architecture and security evolve, and the knowledge presented here may not fully apply in the future.
  - We try to show the evolution of fuses and ME configuration throughout the years, so that you may better understand how it works and how long given mechanism persisted in the history.
- Our expertise reach year ~2019 when our first professional encounter with Boot Guard occurred. This knowledge is a result of 6 years of work, gradually gaining experience and doing experiments. Despite that, not everything might be 100% accurate.
  - The Boot Guard project in 2019 involved Skylake/Kaby Lake system and the information presented here applies to these microarchitectures and newer.

# Location of ME configuration variables and fuses

- The emulated fuses and ME configuration have to live in the updatable flash memory. Of course, there is no other place than the ME region, where it could be stored.
- There are two locations where the emulated fuses and configuration variables are stored:
  - MFS (Management Engine File System) partition before ME 13.x [1]
  - UEP (Unified Emulation Partition) starting with ME 13.x [2]

[1] See: ptresearch/unME11, ptresearch/unME12 and ptresearch/parseMFS. By extracting the MFS partition from ME image and unpacking it with above utilities we can extract the ME configuration variables for ME 11.x and ME 12.x.

[2] Section "Programming NVARS" of System Tools User Guide in ME 13.x (ICL) toolkit and newer.

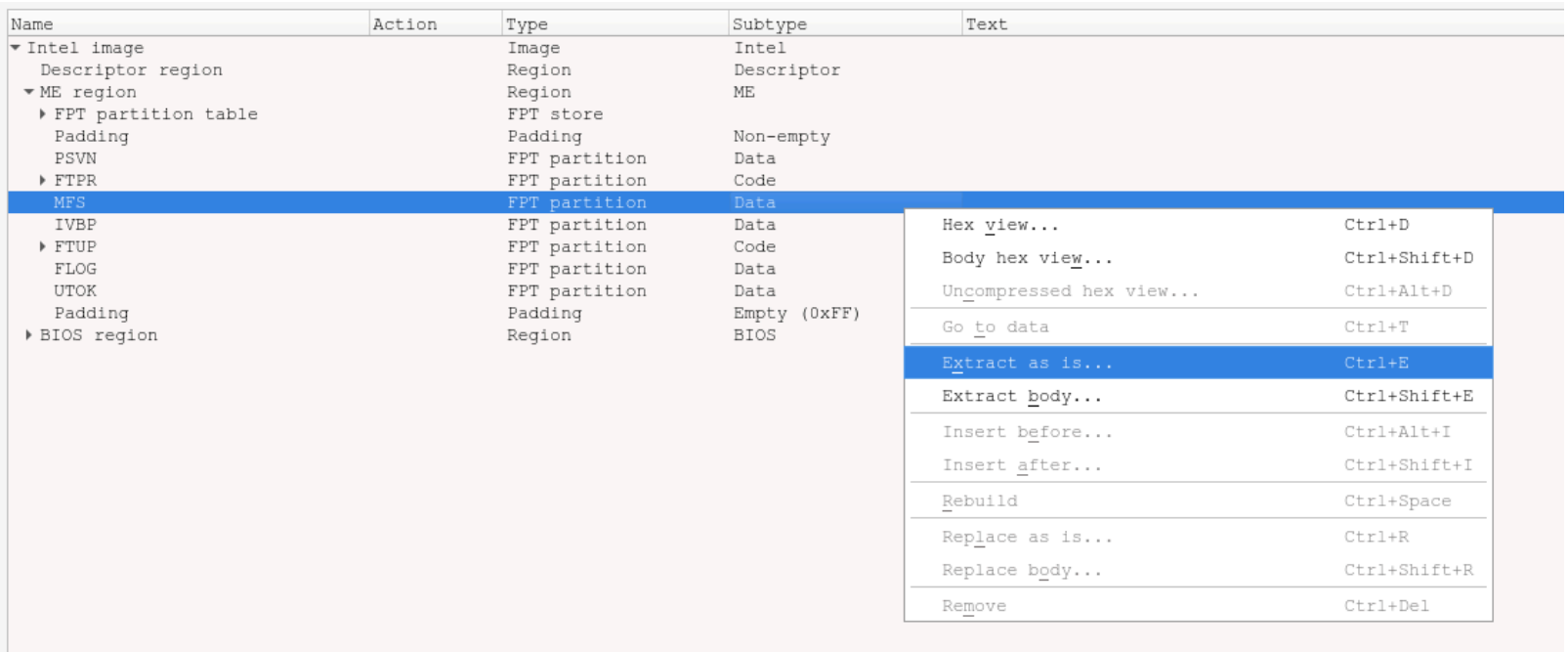
Practice #311 Exercise #1

Homework: Extract MFS from Skylake/Kaby Lake platform ME firmware and unpack it with `parseMFS`.

Feel free to take Protectli FW6 Dasharo release image.

1. Open the firmware image in UEFITool and expand `ME region` to find `MFS` in `FTPR`.
2. Extract MFS body and save to file.
3. Unpack the extracted MFS with PTResearch/parseMFS. You may need to patch it in order to work (see next slides).
4. Inspect the resulting ZIP file. The ZIP filename will be the extracted `MFS` filename appended with `.zip` suffix.

## Practice #311 Exercise #1




Use `Extract as is`!

```
diff --git a/parseMFS.py b/parseMFS.py
old mode 100644
new mode 100755
index 334fcd7..d1bd859
--- a/parseMFS.py
+++ b/parseMFS.py
@@ -65,17 +65,17 @@ class zipStorage(object):
 #******************************************************************************

 def sCfgMode(mode):
-  assert 0 == (mode & ~0x1FFF) # Only 13 lowest bits used
+  #assert 0 == (mode & ~0x1FFF) # Only 13 lowest bits used
   typ = " d"
   #      842184218421
-  sfl = "AEIrwxrwxrwx" # A for Anti-Replay, E for Encrypton, I for Integrity
+  sfl = "?AEIrwxrwxrwx" # A for Anti-Replay, E for Encrypton, I for Integrity
   r = []
   for i in xrange(len(sfl)):
     r.append(sfl[i] if mode & (1<<(len(sfl)-1-i)) else "-")
-  return typ[mode >> 12] + "".join(r)
+  return typ[(mode >> 12) & 1] + "".join(r)

 def sCfgOpt(mode):
-  assert 0 == mode & ~0x001F, "mode == 0x%X" % mode # Only 5 lowest bits used
+  #assert 0 == mode & ~0x001F, "mode == 0x%X" % mode # Only 5 lowest bits used
   sfl = "^?!MF" # M for afterManufacture, F for fromFIT
   r = []
   for i in xrange(len(sfl)):
```

```
@@ -137,7 +137,7 @@ class MFS_Blob_Security(object):
     if self.ar: self.rnd, self.ctr = struct.unpack_from("<LL", self.nonce)

     assert 0x12 == self.u7
-    assert self.enc << 1 == self.u12
+    #assert self.enc << 1 == self.u12

     if 0 == (self.flags & 7):
 #      if '\0'*16 != self.nonce: print self.nonce.encode("hex")
@@ -241,7 +241,7 @@ class MFS_Cfg_Record(object):
   fmtRec = struct.Struct("<12sHHHHHHL")
   def __init__(self, data, iRec):
     self.name, unused, self.mode, self.opt, self.cb, self.uid, self.gid, self.offs =
         self.fmtRec.unpack_from(data, 4+iRec*self.fmtRec.size)
-    assert 0 == unused
+    #assert 0 == unused
     self.name = self.name.rstrip('\0')

 class MFS_Cfg_Storage(object):
@@ -382,7 +382,7 @@ class MFS(object):

     self.sign, self.ver, self.cbTotal, self.nFiles =
         self.fmtVolHdr.unpack_from(self.dChunks[0]) # Volume header is in Chunk#0
     assert 0x724F6201 == self.sign # FileSystem signature
-    assert 1 == self.ver # FileSystem version
+    #assert 1 == self.ver # FileSystem version
     assert self.nSysChunks == (self.fmtVolHdr.size + 2*self.nDataChunks
        + 2*self.nFiles + self.CHUNK_SIZE - 1) / self.CHUNK_SIZE
     self.cbSys = self.nSysChunks * self.CHUNK_SIZE
     assert self.cbData + self.cbSys == self.cbTotal
```

Use the below commands to unpack the MFS:

```
python2.7 parseMFS.py FPT_partition_Data_MFS.bin
FPT_partition_Data_MFS.bin Size:400K, nPages:50, nSysPg:4, nDataPg:45, nDataChunks:5490,
   nSysChunks:188, nFiles:512/519, cbData:351360, cbSys:12032, cbTotal:363392
ls FPT_partition_Data_MFS.bin*
FPT_partition_Data_MFS.bin  FPT_partition_Data_MFS.bin.zip
unzip FPT_partition_Data_MFS.bin.zip
```

`FPT_partition_Data_MFS.bin.zip` is the result of unpacking.

After unzipping the file we will get something like this.

```
.
├── chains
├── fitc.cfg
│   ├── fitc.cfg.log
│   ├── fitc.cfg.txt
│   └── home
├── intel.cfg
│   ├── home
│   ├── intel.cfg.log
│   └── intel.cfg.txt
└── varFS
    ├── home
    └── varFS.log
```

- `chains` is a directory with some intermediate products of extraction
- `intel.cfg` filesystem most likely stores Intel's default configuration (it's a guess).
- `fitc.cfg` filesystem stores configuration patched with FIT tool from metoolkit.
- `varFS` suggests that it is some variable filesystem (it's a guess)

## Practice #311 Exercise #1

```
cat fitc.cfg/ftc.cfg.txt
...
  50: d?--Irwxr-xr-x ———F uid=0046 gid=00EE                home/mca/
  51:  ---Irw———r-- -?!-F uid=0045 gid=0000 @001CC3[   1] home/mca/drtcc
  52:  ---Irw-r——— -?!-F uid=0000 gid=00EE @001CC4[   1] home/mca/eom
  53:  ---Irw-r——— -?--F uid=0046 gid=00EE @001CC5[   1] home/mca/fpf_commit
  54:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC6[   1] home/mca/ish_policy
  55:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC7[   1] home/mca/manuf_lock
...
  7A: d?--Irwxrwx--- ———F uid=0055 gid=0115                home/policy/skumgr/
  7B:  ———rw-rwx--- -?--F uid=0055 gid=00EE @0020A7[   1] home/policy/skumgr/ftpm_enable
...
  88: d?--Irwxrwxr-x ———F uid=0003 gid=0115                home/secureboot/
  89:  ———rw-rw——— -?-MF uid=0003 gid=00EE @0020A8[   2] home/secureboot/bootpolres
  8A:  ———rw-rw——— -?-MF uid=0003 gid=00EE @0020AA[   2] home/secureboot/bootpoltype
  8B:  ———rw-rw——— -?-MF uid=0003 gid=00EE @0020AC[   2] home/secureboot/enfpolicy
  8C:  ———rw-r——— -?-MF uid=0003 gid=00EE @0020AE[   2] home/secureboot/kmid
  8D:  ———rw-rw-r-- -?-MF uid=0003 gid=00EE @0020B0[  20] home/secureboot/pubkeyhash
  8E:  ———rw-rw——— -?-MF uid=0003 gid=00EE @0020D0[   1] home/secureboot/s3optdis
  8F:  ———rw-rw——— -?--F uid=0003 gid=0115 @0020D1[   4] home/secureboot/selftestres
```

## Practice #311 Exercise #1

```
cat fitc.cfg/ftc.cfg.txt
...
  50: d?--Irwxr-xr-x ————F uid=0046 gid=00EE                home/mca/
  51:  ---Irw————r-- -?!-F uid=0045 gid=0000 @001CC3[   1] home/mca/drtcc
  52:  ---Irw-r————— -?!-F uid=0000 gid=00EE @001CC4[   1] home/mca/eom
  53:  ---Irw-r————— -?--F uid=0046 gid=00EE @001CC5[   1] home/mca/fpf_commit
  54:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC6[   1] home/mca/ish_policy
  55:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC7[   1] home/mca/manuf_lock
...
  7A: d?--Irwxrwx--- ————F uid=0055 gid=0115                home/policy/skumgr/
  7B:  ————rw-rwx--- -?--F uid=0055 gid=00EE @0020A7[   1] home/policy/skumgr/ftpm_enable
...
  88: d?--Irwxrwxr-x ————F uid=0003 gid=0115                home/secureboot/
  89:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020A8[   2] home/secureboot/bootpolres
  8A:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020AA[   2] home/secureboot/bootpoltype
  8B:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020AC[   2] home/secureboot/enfpolicy
  8C:  ————rw-r————— -?-MF uid=0003 gid=00EE @0020AE[   2] home/secureboot/kmid
  8D:  ————rw-rw-r-- -?-MF uid=0003 gid=00EE @0020B0[  20] home/secureboot/pubkeyhash
  8E:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020D0[   1] home/secureboot/s3optdis
  8F:  ————rw-rw———— -?--F uid=0003 gid=0115 @0020D1[   4] home/secureboot/selftestres
```

## Practice #311 Exercise #1

```
cat fitc.cfg/ftc.cfg.txt
...
  50: d?--Irwxr-xr-x ----F uid=0046 gid=00EE                home/mca/
  51:  ---Irw----r-- -?!-F uid=0045 gid=0000 @001CC3[   1] home/mca/drtcc
  52:  ---Irw-r----- -?!-F uid=0000 gid=00EE @001CC4[   1] home/mca/eom
  53:  ---Irw-r----- -?--F uid=0046 gid=00EE @001CC5[   1] home/mca/fpf_commit
  54:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC6[   1] home/mca/ish_policy
  55:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC7[   1] home/mca/manuf_lock
...
  7A: d?--Irwxrwx--- ----F uid=0055 gid=0115                home/policy/skumgr/
  7B:  ----rw-rwx--- -?--F uid=0055 gid=00EE @0020A7[   1] home/policy/skumgr/ftpm_enable
...
  88: d?--Irwxrwxr-x ----F uid=0003 gid=0115                home/secureboot/
  89:  ----rw-rw----- -?-MF uid=0003 gid=00EE @0020A8[   2] home/secureboot/bootpolres
  8A:  ----rw-rw----- -?-MF uid=0003 gid=00EE @0020AA[   2] home/secureboot/bootpoltype
  8B:  ----rw-rw----- -?-MF uid=0003 gid=00EE @0020AC[   2] home/secureboot/enfpolicy
  8C:  ----rw-r------ -?-MF uid=0003 gid=00EE @0020AE[   2] home/secureboot/kmid
  8D:  ----rw-rw-r-- -?-MF uid=0003 gid=00EE @0020B0[  20] home/secureboot/pubkeyhash
  8E:  ----rw-rw----- -?-MF uid=0003 gid=00EE @0020D0[   1] home/secureboot/s3optdis
  8F:  ----rw-rw----- -?--F uid=0003 gid=0115 @0020D1[   4] home/secureboot/selftestres
```

## Practice #311 Exercise #1

```
cat fitc.cfg/ftc.cfg.txt
...
  50: d?--Irwxr-xr-x ————F uid=0046 gid=00EE                home/mca/
  51:  ---Irw————r-- -?!-F uid=0045 gid=0000 @001CC3[   1] home/mca/drtcc
  52:  ---Irw-r————— -?!-F uid=0000 gid=00EE @001CC4[   1] home/mca/eom
  53:  ---Irw-r————— -?--F uid=0046 gid=00EE @001CC5[   1] home/mca/fpf_commit
  54:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC6[   1] home/mca/ish_policy
  55:  ---Irw-r--r-- -?--F uid=0046 gid=00EE @001CC7[   1] home/mca/manuf_lock
...
  7A: d?--Irwxrwx--- ————F uid=0055 gid=0115                home/policy/skumgr/
  7B:  ————rw-rwx--- -?--F uid=0055 gid=00EE @0020A7[   1] home/policy/skumgr/ftpm_enable
...
  88: d?--Irwxrwxr-x ————F uid=0003 gid=0115                home/secureboot/
  89:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020A8[   2] home/secureboot/bootpolres
  8A:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020AA[   2] home/secureboot/bootpoltype
  8B:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020AC[   2] home/secureboot/enfpolicy
  8C:  ————rw-r————— -?-MF uid=0003 gid=00EE @0020AE[   2] home/secureboot/kmid
  8D:  ————rw-rw-r-- -?-MF uid=0003 gid=00EE @0020B0[  20] home/secureboot/pubkeyhash
  8E:  ————rw-rw———— -?-MF uid=0003 gid=00EE @0020D0[   1] home/secureboot/s3optdis
  8F:  ————rw-rw———— -?--F uid=0003 gid=0115 @0020D1[   4] home/secureboot/selftestres
```

# ME file system interface

- MFS and configuration variables are usually modified with FIT/mFIT tool
- It is possible to modify MFS using commands sent to ME via MEI interface in OS:
  - The Intel ME BIOS interface specification describes the Read File and Set File APIs. These APIs are used by FPT tool.
  - These APIs consume the file path we have seen in `parseMFS` output.

**Table 6-46. Definition for READ FILE Message**

| Field | Size (Byte) | Default | Description |
|---|---|---|---|
| MEI Header | 4 | 0x804D0007 | Refer to Section 6.2.1 for more details |
| MKHI Header | 4 | 0x0000020A | Refer to Section 6.3.1 for more details |
| File Path | 64 | NA | File name path. |
| Offset | 4 | NA | Offset of file |
| Data Size | 4 | NA | Size to read |
| Flags | 1 | 0 | Reserved |

*ME 12.x BIOS Interface Specification (doc 572579)*

# ME file system interface

- There are also Intel documents called ME Firmware Variable Structures that describe (in pseudo code form) how to read and write ME configuration. But also show some of the file names.

```
////////////////////////////////////////////////////////////
// global list of valid CVARs and their configurations
Var
  // defines the size of the rule minus a UINT8 and plus a UINT32
  CFG_RULE_SIZE   : (SizeOf(RULE) - 1 + 4);
  gCvarConfigList   : array [0..2] of CvarConfig = (
    (name   : 'OEMSkuRule';
     length   : 4;
     fileName : '/home/policy/cfgmgr/cfg_rules';
     offset   : CFG_RULE_SIZE(MEFWCAPS_PCV_OEM_CAP_CFG_RULE);
     detail   : IS_NVAR + IS_FOV + GRESET_REQ + IS_HIDDEN),

  (name   : 'MEManufacturingModeDone';
   length   : 1;
   fileName : '/home/mca/eom';   <-------------------------
   offset   : 0;
   detail   : 0),
```

Source: Firmware variable Structures for ME 11.0 Firmware Programming Specification (doc 605318)

# ME file system vs fuses

- One can also find a sample application in the wild that uses Linux MEI driver to read the ME configuration variables and fuses.
  - Fuses can be read using the same file API discussed before
- GitHub axxia/axxia_support is a mine of knowledge and sample code, especially at revision 6e1ca2c69a12266da4fcec74104e0b270ea2aa9b (already in the link)
- The repo contains a lot of PEM keys used for RoT committed by `@intel.com` email...

`heci_example/heci_example.c` :

```
FUSE_INFO_t  FuseList[] = {
  {"FtpmEnabled", 1},
  {"SocCfgLock",  1},
  {"SBValid", 1},
  {"VendorDefId", 16},
  {"OemCred", 256},
  {"SbPolicies",  16},
  {"Enf0",  1},
  {"Enf1",  1},
...
```

# ME file system vs fuses

Mapping of fuses to MFS files:

- `SocCfgLock` -> `fpf_commit` (guessing?)
- `SBPolicies` -> concatenated `bootpolres`, `bootpoltype` and `kmid`
- `Enf0/1` -> `enfpolicy`
- `OemCred` -> `pubkeyhash`
- `FtpmEnabled` -> `ftpm_enable`

`SBValid` is probably a fuse that indicates Boot Guard valid configuration.

Source: Own experiments on ME11.x firmware

Homework: Try to run heci_example application from GitHub axxia/axxia_support on Skylake or Kaby Lake system and see if you can read some fuses `/fpf/*` or configuration variables `/home/*`.

Extend the code and fuse list with the configuration variables files if you need.

# Quiz #1

# Quiz #1

Where were emulated fuses stored before ME 13.x?

# Quiz #1

Where were emulated fuses stored before ME 13.x?

- ME region MFS partition,

# Quiz #1

Where were emulated fuses stored before ME 13.x?

- ME region MFS partition,

Where are emulated fuses currently stored (after ME 13.x)?

# Quiz #1

**Where were emulated fuses stored before ME 13.x?**

- ME region MFS partition,

**Where are emulated fuses currently stored (after ME 13.x)?**

- ME region UEP partition,

# Quiz #1

**Where were emulated fuses stored before ME 13.x?**

- ME region MFS partition,

**Where are emulated fuses currently stored (after ME 13.x)?**

- ME region UEP partition,

**What are the ways to read/modify emulated fuses and ME variables?**

# Quiz #1

**Where were emulated fuses stored before ME 13.x?**

- ME region MFS partition,

**Where are emulated fuses currently stored (after ME 13.x)?**

- ME region UEP partition,

**What are the ways to read/modify emulated fuses and ME variables?**

- FIT/mFIT,
- ME File API with FPT,

# ME variables and fuses vs documentation

| Number | FACB | V | M | ENF | Features/Use Case/Rationale |
|---|---|---|---|---|---|
| 5 | 1 | 1 | 1 | 3 | 1. Boot Guard performs all "secure boot" functions and does not allow for recovery.<br>2. Strictest Standard Boot Policy / Profile. |

*Boot Guard BIOS Specification 3.0.0 (doc 557867)*

- Boot Guard Profile 5 - ENF=3: `Enf0` = 1 and `Enf1` = 1
  - `Enf0` - bit0 of ENF
  - `Enf1` - bit1 of ENF
- `FACB`, `V`, and `M` are part of `SBPolicies`

# ME variables and fuses vs documentation

**Fragment of BTG_SACM_INFO MSR (Sheet 2 of 2)**

| Bit | Label | Description |
|---|---|---|
| 6 | Boot Policies | BP.TYPE.V - Boot Guard verifies IBB. Copied from BP.TYPE register |
| 5 | | BP.TYPE.M - Boot Guard measures IBB into the TPM. Copied from BP.TYPE register |
| 4 | | BP.RSTR.FACB - launch of Boot Guard ACM is enforced. Copied from BP.RSTR register |

**ACM_POLICY_STATUS Register (Sheet 1 of 2)**

| Bit | Label | Comment |
|---|---|---|
| 3:0 | KMID | Key Manifest ID from BP.KEYTYPE FPF register |
| 4 | BootPolicies | BP.TYPE.M - Boot Guard measures IBB into the TPM |
| 5 | | BP.TYPE.V - Boot Guard verifies IBB |
| 6 | | BP.TYPE.HAP – Indicates HAP platform |
| 7 | | BP.TYPE.T – Indicates Intel® TXT supported |
| 8 | | Reserved |
| 9 | | BP.RSTR.DCD – Disable CPU debug |
| 10 | | BP.RSTR.DBI – Disable BSP init |
| 11 | | BP.RSTR.PBE Protect BIOS environment |

*CBnT BIOS Writers Guide 1.2.5 (doc 575623)*

# ME variables and fuses vs documentation

1. `/fpf/SbPolicies` or `/fpf/intel/SbPolicies` - Boot Guard policy FPF:
   - bit0 - Force Anchor Cove Boot (FACB)
   - bit1 - Disable CPU Debug (DCD)
   - bit2 - Disable BSP Initialization (DBI)
   - bit3 - Protect BIOS Environment (PBE)
   - bit4 - Verified Boot (V)
   - bit5 - Measured Boot (M)
   - bit6 to bit 9 - Key Manifest ID (may be shifted further on newer ME versions, to bits 8-11)

Source: Own experiments on ME11.x firmware

# ME variables and fuses vs documentation

2. `/home/secureboot/bootpolres` - refers to `BP.RSTR`:
   - bit0 - Force Anchor Cove Boot (FACB)
   - bit1 - Disable CPU Debug (DCD)
   - bit2 - Disable BSP Initialization (DBI)
   - bit3 - Protect BIOS Environment (PBE)
3. `/home/secureboot/bootpoltype` - refers to `BP.TYPE`:
   - bit0 - Verified Boot
   - bit1 - Measured Boot
   - bit2 - High Assurance Platform (HAP)
   - bit3 - TXT Supported (ME12.x and later)
4. `/home/secureboot/kmid` - Key Manifest ID has weird 3-bit offset in ME11.x:
   - bit2-bit5 - Key Manifest ID

Source: Own experiments on ME11.x firmware

# ME variables and fuses vs documentation

**Table 3-11. Host Firmware Status Register #6 (HFSTS6) Definition (Offset 6Ch)**

| Bits | Description |
|---|---|
| 0 | **Force Boot Guard ACM Boot Policy (F):** If the Boot Guard ACM is not found or is not authenticated, the CPU will take action based on the value of this field.<br>If this field is 0, then the BIOS will jump to the legacy BIOS boot vector.<br>If this field is 1, then the BIOS will shut down. The delay before shutdown is determined by the value of the Error Enforcement Policy (ENF) field in Bits [7:6] of this register. |
| 1 | **CPU Debug Disabled:** If this bit is true, the platform shall disable CPU debug mode.<br>0 – False<br>1 – True |

*ME 12.x BIOS Interface Specification (doc 572579)*

- ME mirrors the Boot Guard policies in the PCI space (00:16.0) for older ME versions:
  - `Force Boot Guard ACM Boot Policy (F)` -> `Force Anchor Cove Boot (FACB)`
  - `CPU Debug Disabled` -> `Disbale CPU Debug (DCD)`

# ME variables and fuses vs documentation

| Bits | Description |
|---|---|
| 2 | **BSP Initialization Disabled:** If this bit is true, BSP will shut down when it receives a CPU INIT signal.<br>0 – False<br>1 – True |
| 3 | **Protect BIOS Environment Policy (PBE):** If PBE policy is enabled, Boot Guard ACM is going to set up the protected environment for BIOS Initial Boot Block (IBB) execution.<br>0 – Take no action to control the environment during the execution of the BIOS component.<br>1 – Take action to control the environment during the execution of the BIOS component. |

*ME 12.x BIOS Interface Specification (doc 572579)*

- `BSP Initialization Disable` -> `Disable BSP Initialization (DBI)`
- `PBE` is `PBE` without changes

# ME variables and fuses vs documentation

| | |
|---|---|
| 7:6 | **Error Enforcement Policy (ENF):** The system will follow the ENF policy when the Boot Guard ACM encounters a fatal error.<br>0 – The system will not shut down due to a Boot Guard ACM failure<br>1 – The system will shut down due to a Boot Guard ACM failure<br>2 – The system will shut down immediately due to a Boot Guard ACM failure<br>3 – The system will shut down in 30 minutes due to a Boot Guard ACM failure |
| 8 | **Measured Boot Policy (M):** If M policy is enabled, the platform performs the measured boot.<br>0 – Disable<br>1 – Enable |
| 9 | **Verified Boot Policy (V):** If V policy is enabled, the platform performs the verified boot.<br>0 – Disable<br>1 – Enable |

*ME 12.x BIOS Interface Specification (doc 572579)*

- Measured Boot, Verified Boot, and Enforcement Policy also without changes

# ME variables and fuses vs documentation

| | |
|---|---|
| 13:10 | **Boot Guard ACM Security Version Number (ACMSVN):** This field contains the Boot Guard ACM revocation value. The value is Field Programmable Fuses (FPF) data.<br>Once the platform has been configured with the updated Boot Guard ACM, then the platform will not accept a Boot Guard ACM with a lower Security Version Number (SVN). |
| 17:14 | **Key Manifest Security Version Number (KMSVN):** This field contains the key manifest revocation value. The value is Field Programmable Fuses (FPF) data.<br>Once the platform has been configured with the updated KM, then the platform will not accept a KM with a lower Security Version Number (SVN). |
| 21:18 | **Boot Policy Manifest Security Version Number (BPMSVN):** This field contains the BIOS policy manifest revocation value. The value is Field Programmable Fuses (FPF) data.<br>Once the platform has been configured with the updated BPM, then the platform will not accept a BPM with a lower Security Version Number (SVN). |
| 25:22 | **Key Manifest ID (KMID):** This field contains the hash of another public key, used by the Boot Guard ACM to verify the Boot Policy Manifest. The value is Field Programmable Fuses (FPF) data. |

*ME 12.x BIOS Interface Specification (doc 572579)*

- Key Manifest ID without changes
- ACMSVN -> `/fpf/SbAcmSvn` fuse
- KMSVN -> `/fpf/SbKmSvn` fuse
- BPMSVN -> `/fpf/SbBsmmSvn` fuse

# ME variables and fuses vs documentation

| Bits | Description |
|---|---|
| 26 | **BSP Boot Policy Manifest Execution Status:** The field indicates the Boot Policy Manifest execution complete on BSP.<br>1 – **Boot Policy Manifest** completed execution on BSP<br>0 – **Boot Policy Manifest** did not complete execution on BSP |
| 27 | **Error:** CPU encountered an unexpected error and is asking Intel® ME firmware to start the enforcement logic. Intel® ME firmware will check this ERROR bit at the time of receiving the Startup Module (SM) completion on BSMES bit [26]. |
| 28 | **Boot Guard Disable:** This field indicates the Boot Guard fuse state in the PCH.<br>0 – Boot Guard Enabled<br>1 – Boot Guard Disabled |
| 29 | **Field Programmable Fuses (FPF) Disable:** This field indicates the FPF fuse state in the PCH.<br>0 – FPF Enabled<br>1 – FPF Disabled |
| 30 | **Field Programmable Fuses (FPF) SoC Configuration Lock:** This field indicates the status of chipset FPF Configuration Lock.<br>0 – FPF is not committed<br>1 – FPF is committed |
| 31 | **TXT Support:** This field indicates the status of TXT support.<br>0 – TXT is not supported<br>1 – TXT is supported |

*ME 12.x BIOS Interface Specification (doc 572579)*

- `Boot Guard Disable` -> inverted `/fpf/SbValid` fuse perhaps?
- `FPF SoC Configuration Lock` -> `/fpf/SocCfgLock` fuse
- `TXT Support` -> `/fpf/TxtEnabled` fuse

# ME configuration variables in ME13.x and later

- With the introduction of ME 13 on Ice Lake platforms, all releases of ME changed how the configuration is stored and the API to interface with it.
- The configuration is now stored in Unified Emulation Partition (UEP), which contains the emulated fuses to be used during development.
- The file access API changed to use 32bit `File ID` and `Offset` instead of file paths on client systems. Servers still use file paths:

**Table 6-107. READ FILE EX Message Definition**

| Field | Size (Byte) | Default | Description |
|---|---|---|---|
| MEI Header | 4 | 0x80xx0009 | Refer to Section 6.2.1 for more details |
| MCHI Header | 4 | 0x00000A0A | Refer to Section 6.5.1 for more details |
| File ID | 4 | N/A | File ID |
| Offset | 4 | N/A | Offset of file |
| Data Size | 4 | N/A | Size to read |
| Flags | 1 | 0 | Reserved |

*ME 13.0 BIOS Interface Specification (doc 574509)*

# ME configuration variables and fuses on servers

- Apparently, SPS on servers still use filepaths in the file API:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| */fpf/SBValid* | *0* | *1* | */home/SCA/CpuDbg* | *0* | *1* | Cpu Debug Policy Enabled | Cpu Debug Policy Enabled |
| */fpf/SBValid* | *0* | *1* | *N/A; When BtG FPFs burned correctly, set to 1* | *N/A* | | No FITm parameter | Boot Guard FPFs Enabled |
| */fpf/PttEnabled* | *0* | *1* | */home/policy/skumgr/ftpm_enable* | *0* | *1* | Intel(R) PTT Supported <wbr>[FPF] | Intel(R) PTT Supported <wbr>[FPF] |
| */fpf/Fd0vEn* | *0* | *1* | */home/SCA/Fd0vDRA* | *0* | *1* | Node: Flash Descriptor Verification | Flash Descriptor Verification Enabled |
| */fpf/Enf0* | *0* | *1* | */home/secureboot/enfpolicy* | *0* | *1* | Part of Boot Guard Profile Configuration | Error Enforcement Policy 0 |
| */fpf/Enf1* | *0* | *1* | | *1* | *1* | | Error Enforcement Policy 1 |
| */fpf/SbPolicies* | *0* | *4* | */home/secureboot/bootpolres* | *0* | *4* | Part of Boot Guard Profile Configuration | *Boot Guard Policy Restrictions -> <wbr>[0] - Force Boot Guard ACM Boot Policy* |
| | | | | | | CPU debugging | *Boot Guard Policy Restrictions -> <wbr>[1] - CPU debugging capability probe mode* |
| | | | | | | BSP Initialization | *Boot Guard Policy Restrictions -> <wbr>[2] - Determines BSP behavior* |
| | | | | | | Part of Boot Guard Profile Configuration | *Boot Guard Policy Restrictions -> <wbr>[3] - Protect BIOS Environment* |
| | *4* | *2* | */home/secureboot/bootpoltype* | *0* | *2* | Part of Boot Guard Profile Configuration | Boot Guard Policy Type |
| | *6* | *4* | */home/secureboot/kmid* | *2* | *4* | Key Manifest ID | Key Manifest ID |
| */fpf/OemCred* | *0* | *256* | */home/secureboot/pubkeyhash* | *0* | *256* | OEM Public Key Hash | OEM Public Key Hash |

*SPS Tools User Guide from SPS_E5_06.01.04.089.0 toolkit (kit number 845274)*

Practice #311 Exercise #3

Inspecting UEP in ODROID-H4 Slim Bootloader binary provisioned for Boot Guard.

1. Check the SHA384 hash of the OEM Root Key:

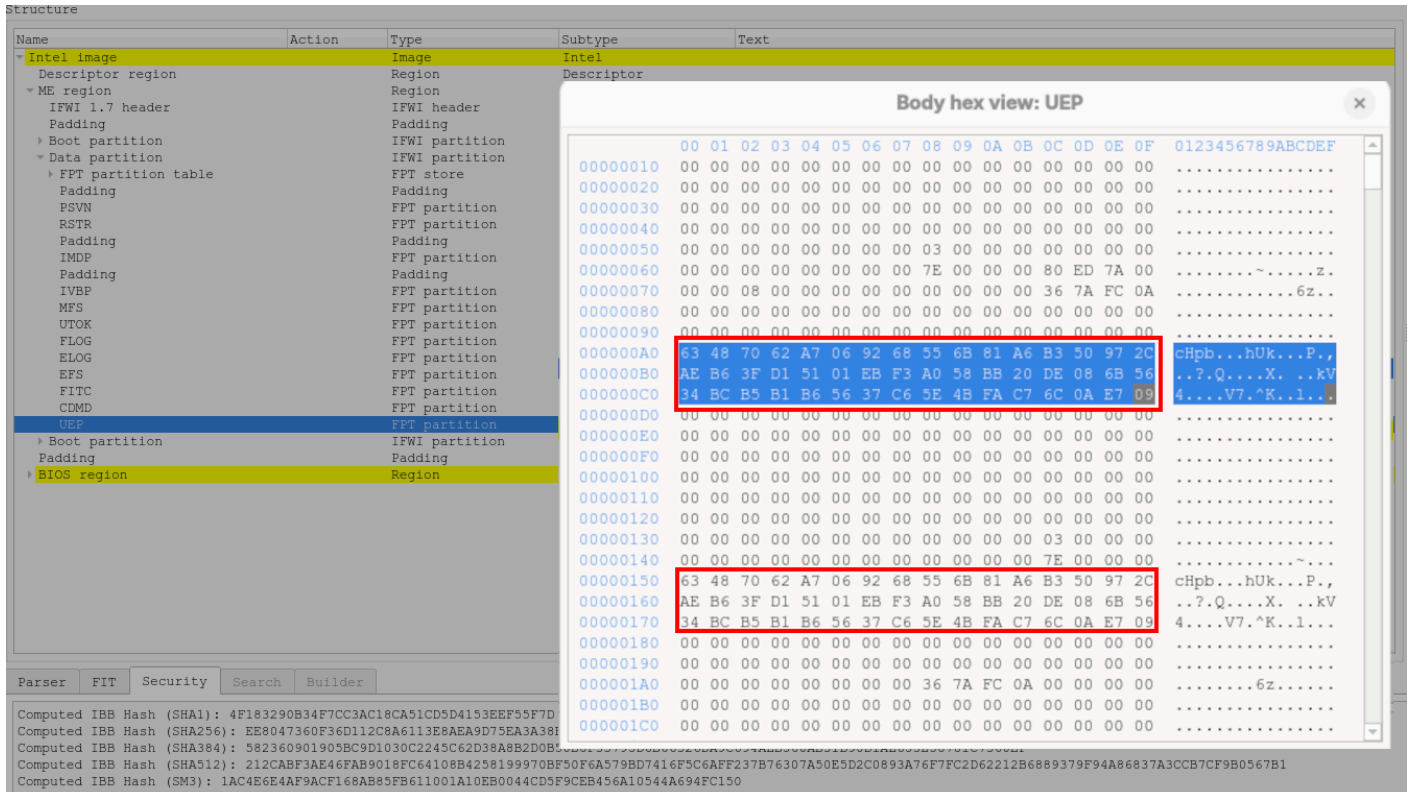
```
sha384sum ~/training_materials/src/slimbootloader/Temp/kmsigpubkey.bin
```

2. Open provisioned image (build it first if you don't have it anymore) in UEFITool:

```
user@OST2-VM:~$ ~/training_materials/uefi/uefitool \
 ~/training_materials/src/slimbootloader/sbl_ifwi_adln.bin
```

3. Open the body hex view of `Intel Image/ME region/Data Partition/UEP`.

```
user@OST2-VM:~$ sha384sum ~/training_materials/src/slimbootloader/Temp/kmsigpubkey.bin
63487062a7069268556b81a6b350972caeb63fd15101ebf3a058bb20de086b5634bcb5b1b65637c65e4bfac76c0ae709
```



# ME configuration variables in ME13.x and later

- The `File ID` and `Offset` for the file API are even less documented than the file paths for ME 12 and earlier. It is yet unexplored terrain for us.
- But do we have any way to find out the File IDs and offsets?
  - Of course we have!
- Time for some behavioral and static analysis of the Intel tools.

Searching for strings inside `MEInfo`.

1. Transfer the ME tools we used in RoT Assessment and Provisioning to VM.
2. Run `strings -N 7 MEInfo > meinfo_strings` on the Linux version of he utility.
3. Inspect the `meinfo_strings` file.

## Practice #311 Exercise #4

```
user@OST2-VM:~$ cat meinfo_strings |grep home
...
/home/secureboot/btgdbgdata
...
/home/mca/eom
/home/mca/manuf_lock
...
/home/ptt/ptt_smx_disable
...
/home/secureboot/brm_cfg
...
/home/mca/eom_config
```

## Practice #311 Exercise #4

```
user@OST2-VM:~$ cat meinfo_strings |grep fpf
/fpf/intel/SBValid
...
/fpf/intel/Enf0
/fpf/intel/Enf1
...
/fpf/intel/FtpmEnabled
...
/fpf/intel/OemCred
...
/fpf/intel/SbPolicies
...
/fpf/intel/SocCfgLock
```

Practice #311 Exercise #5

Inspecting MEInfo behavior on the Linux system.

We will be analysing the `MEInfo` behavior running under DTS on ODROID-H4. We will use `strace` utility to track what the `MEInfo` application is writing to or reading from the MEI interface.

1. Transfer `MEInfo` to DTS USB stick.
2. Plug the DTS USB stick to ODROID-H4.
3. Power on the ODROID-H4 and boot to DTS shell.
4. Execute `strace -x MEInfo > meinfo_strace 2>&1`.

meinfo_strace:

```
openat(AT_FDCWD, "/dev/mei0", O_RDWR)   = 3
ioctl(3, IOCTL_MEI_CONNECT_CLIENT, 0×7ffcaae6cd60) = 0
write(3, "\x0a\x07\x00\x00\x00\x5b\x00\x40\x00", 9) = 9
pselect6(4, [3], NULL, NULL, {tv_sec=30, tv_nsec=0}, NULL) = 1 (in [3], left {tv_sec=29, tv_nsec=999244924})
read(3, "\x0a\x87\x00\x00\x30\x00\x00\x00\x00", 4096) = 9
write(3, "\x0a\x07\x00\x00\x00\x5b\x00\x40\x00", 9) = 9
pselect6(4, [3], NULL, NULL, {tv_sec=30, tv_nsec=0}, NULL) = 1 (in [3], left {tv_sec=29, tv_nsec=999341162})
read(3, "\x0a\x87\x00\x00\x30\x00\x00\x00\x00", 4096) = 9
write(3, "\x0a\x0a\x00\x00\x00\x5b\x00\x40\x00\x00\x00\x00\x30\x00\x00\x00\x04", 17) = 17
pselect6(4, [3], NULL, NULL, {tv_sec=30, tv_nsec=0}, NULL) = 1 (in [3], left {tv_sec=29, tv_nsec=998728306})
read(3, "\x0a\x8a\x00\x00\x30\x00\x00\x00\xc6\xa3\x82\x6c\xf0\x4c\x1e\x59
 \x35\x11\x43\xc2\xb5\x39\xdc\x55\xf0\x18\xa0\x8a\x35\xb3\x55\x27"... , 4096) = 56
```

Here comes our OEM Root Key hash... MEInfo output:

```
1st OEM Public Key Hash UEP
C6A3826CF04C1E59351143C2B539DC55F018A08A35B35527
A53E2A16B3DFE4D953BBEAB590F21306FC69652EDF4AC5BF
```

**Table 6-107. READ FILE EX Message Definition**

| Field | Size (Byte) | Default | Description |
|---|---|---|---|
| MEI Header | 4 | 0x80xx0009 | Refer to Section 6.2.1 for more details |
| MCHI Header | 4 | 0x00000A0A | Refer to Section 6.5.1 for more details |
| File ID | 4 | N/A | File ID |
| Offset | 4 | N/A | Offset of file |
| Data Size | 4 | N/A | Size to read |
| Flags | 1 | 0 | Reserved |

```
write(3, "\x0a\x0a\x00\x00\x00\x5b\x00\x40\x00\x00\x00\x00\x30\x00\x00\x00\x04", 17) = 17
```

- `MCHI header` - `\x0a\x0a\x00\x00` ( `0×00000A0A` in Little Endian)
- `File ID` - `\x00\x5b\x00\x40` ( `0×40005b00` in Little Endian)
- `Offset` - `\x00\x00\x00\x00` ( `0×00000000` in Little Endian)
- `Data Size` - `\x30\x00\x00\x00` ( `0×00000030` in Little Endian)
- `Flags` - `\x04` ( `0×04` in Little Endian)

**Table 6-108. READ FILE EX RESPONSE Message Definition**

| Field | Size (Byte) | Default | Description |
|---|---|---|---|
| MEI Header | 4 | 0x80xx0009 | Refer to Section 6.2.1 for more details |
| MCHI Header | 4 | 0x00008A0A | Refer to Section 6.5.1 for more details |
| Data size | 4 | N/A | Actual size was read. |
| Data | xx | N/A | Data buffer for Data to read |

```
read(3, "\x0a\x8a\x00\x00\x30\x00\x00\x00\xc6\xa3\x82\x6c\xf0\x4c\x1e\x59
 \x35\x11\x43\xc2\xb5\x39\xdc\x55\xf0\x18\xa0\x8a\x35\xb3\x55\x27"..., 4096) = 56
```

- `MCHI header` - `\x0a\x8a\x00\x00` ( `0×00008A0A` in Little Endian)
- `Data Size` - `\x30\x00\x00\x00` ( `0×00000030` in Little Endian)
- `Flags` - `\x04` ( `0×04` in Little Endian)
- `Data` - `\xc6\xa3\x82\x6c\xf0\x4c\x1e\x59 ...` <- our actual OEM Root Key hash

# Mapping of MEInfo output to fuses

Let's recall the output of MEInfo from the RoT Assessment presentation:

```
FW Supported FPFs                              FPF         UEP
 *In Use
 ---          ---
...
Error Enforcement Policy 0                     Not set     Disabled
Error Enforcement Policy 1                     Not set     Disabled
...
Intel(R) PTT                                   Not set     Enabled
...
OEM Secure Boot Policy                         Not set     0x42
CPU Debugging                                  Not set     Disabled
BSP Initialization                             Not set     Enabled
Protect BIOS Environment                       Not set     Disabled
Measured Boot                                  Not set     Disabled
Verified Boot                                  Not set     Disabled
Key Manifest ID                                Not set     0x01
Force Boot Guard ACM                           Not set     Disabled
TXT Supported                                  Not set     Disabled
SOC Config Lock State                          Not set     Disabled
...
1st OEM Public Key Hash UEP
 0638EFCBB322E655C912D984070450BD22151A6BA96E53E8
 6DDBAD48273EEE2BAB15DBE819696D989ABDC5EB9EB68A11
```

# Mapping of MEInfo output to fuses

Let's recall the output of MEINfo from the RoT Assessment presentation:

```
Error Enforcement Policy 0                      Not set     Disabled
Error Enforcement Policy 1                      Not set     Disabled
```

- MFS: `/home/secureboot/enfpolicy`
- FPF: `/fpf/Enf0` and `/fpf/Enf1`, or `/fpf/intel/Enf0` and `/fpf/intel/Enf1`

```
Intel(R) PTT                                    Not set     Enabled
```

- MFS: `/home/policy/skumgr/ftpm_enable`
- FPF: `/fpf/FtpmEnabled` or `/fpf/intel/FtpmEnabled`

```
OEM Secure Boot Policy                          Not set     0x42
```

- MFS: concatenated `/home/secureboot/bootpolres` and `/home/secureboot/bootpoltype`
- FPF: `/fpf/SBPolicies` or `/fpf/intel/SBPolicies`

# Mapping of MEInfo output to fuses

```
CPU Debugging                                   Not set     Disabled
BSP Initialization                              Not set     Enabled
Protect BIOS Environment                        Not set     Disabled
Force Boot Guard ACM                            Not set     Disabled
```

- MFS: part of `/home/secureboot/bootpolres`
- FPF: part of `/fpf/SbPolicies` or `/fpf/intel/SbPolicies`

```
Measured Boot                                   Not set     Disabled
Verified Boot                                   Not set     Disabled
```

- MFS: part of `/home/secureboot/bootpoltype`
- FPF: part of `/fpf/SbPolicies` or `/fpf/intel/SbPolicies`

```
TXT Supported                                   Not set     Disabled
```

- MFS: part of `/home/secureboot/bootpoltype`
- FPF: part of `/fpf/TxtEnabled` or `/fpf/intel/TxtEnabled`

# Quiz #2

# Quiz #2

Where to look for information about fuses and ME configuration variables?

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

ME/SPS tools User Guides in ME/SPS toolkits from Intel.

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

ME/SPS tools User Guides in ME/SPS toolkits from Intel.

- SPS versions are especially valuable.
- Modern SPS toolkits contain mFIT with Python source and XMLs!

  Can be leveraged to know the structure of UEP or how to patch ME/SPS image.

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

ME/SPS tools User Guides in ME/SPS toolkits from Intel.

- SPS versions are especially valuable.
- Modern SPS toolkits contain mFIT with Python source and XMLs!

  Can be leveraged to know the structure of UEP or how to patch ME/SPS image.

Researchers' findings like PTResearch/parseMFS and their talks on the conferences,

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

ME/SPS tools User Guides in ME/SPS toolkits from Intel.

- SPS versions are especially valuable.
- Modern SPS toolkits contain mFIT with Python source and XMLs!

  Can be leveraged to know the structure of UEP or how to patch ME/SPS image.

Researchers' findings like PTResearch/parseMFS and their talks on the conferences,

Intentional or unintentional leaks? Like GitHub axxia/axxia_support,

# Quiz #2

**Where to look for information about fuses and ME configuration variables?**

ME/SPS BIOS interface specifications from Intel,

ME/SPS tools User Guides in ME/SPS toolkits from Intel.

- SPS versions are especially valuable.
- Modern SPS toolkits contain mFIT with Python source and XMLs!

  Can be leveraged to know the structure of UEP or how to patch ME/SPS image.

Researchers' findings like PTResearch/parseMFS and their talks on the conferences,

Intentional or unintentional leaks? Like GitHub axxia/axxia_support,

Firmware variable Structures for ME x.y Firmware Programming Specification from Intel.

# Dream and nightmare - deguard

- deguard is an utility leveraging CVE-2017-5705 to bypass Boot Guard fuses with ROPs
- Require a specific older versions of ME 11.x (newer MEs are not affected nor supported by the utility)
- Overwrites the copy of field programmable fuses stored in SRAM, resulting in the fused Boot Guard configuration being replaced with an arbitrary one

## Dream for open-source but nightmare for businesses relying on Boot Guard for security

# Conclusions

- From the previous presentations, we know that the usage of Intel tools is far from production readiness
  - Replacing those tools is possible, although there are still couple problems unresovled. E.g. how to inject OEM Key Manifest into ME region?
  - We have some ideas, but never enough time to actually do it.
- We learned about the fuses and their purpose and how they evolved over time.
- We learned how we can read the fuses ourselves in Linux.
- Understanding the ME fuses, configuration and security is a tremendous effort given the scarce documentation, especially in client segment.
- We saw how vulnerability in ME firmware may lead to a complete compromise of the Root of Trust of whole system.

# Q&A