Rootkit in your laptop:
Hidden code in your chipset and how to discover what exactly it does

Igor Skochinsky
Hex-Rays

Breakpoint 2012
Melbourne
Outline

- High-level overview of the ME
- Low-level details
- ME security and potential attacks
- Results
- Future work
ME: High-level overview

- Management Engine (or Manageability Engine) is a dedicated microcontroller on recent Intel platforms
- In first versions it was included in the network card, later moved into the chipset
- Shares flash with the BIOS but is completely independent from the main CPU
- Can be active even when the system is hibernating or turned off (but connected to mains)
- Has a dedicated connection to the network interface; can intercept or send any data without main CPU's knowledge
ME: High-level overview

Credit: Intel 2009
ME: High-level overview

Communicating with the Host OS and network

- **HECI**: Host Embedded Controller Interface; communication using a PCI memory-mapped area
- Network protocol is SOAP based; can be plain HTTP or HTTPS
ME: High-level overview

Some of the ME components

- Active Management Technology (AMT): remote configuration, administration, provisioning, repair, KVM
- System Defense: lowest-level firewall/packet filter with customizable rules
- IDE Redirection (IDE-R) and Serial-Over-LAN (SOL): boot from a remote CD/HDD image to fix non-bootable or infected OS, and control the PC console
- Identity Protection: embedded one-time password (OTP) token for two-factor authentication
- Protected Transaction Display: secure PIN entry on a remote server not visible to the host software
Intel Anti-Theft

- PC can be locked or disabled if it fails to check-in with the remote server at some predefined interval; if the server signals that the PC is marked as stolen; or on delivery of a "poison pill"
- Poison pill can be sent as an SMS if a 3G connection is available
- Can notify disk encryption software to erase HDD encryption keys
- Reactivation is possible using previously set up recovery password or by using one-time password
ME: Low-level details
ME: Low-level details

Sources of information

- Intel's whitepapers and other publications (e.g. patents)
- Intel's official drivers and software
  - HECI driver, management services, status checkers
  - AMT SDK, code samples
  - Linux drivers and supporting software; coreboot
- BIOS updates for boards on Intel chipsets
  - Even though ME firmware is usually not updateable using normal means, it's usually still included in the BIOS image
  - Sometimes separate ME firmware updates are available too
ME: Low-level details

Sources of information

- Intel's ME Firmware kits are not supposed to be distributed to end users
- However, many vendors still put up the whole package instead of just the drivers, or forget to disable the FTP listing

With a few picked keywords you can find the good stuff :)
ME: Low-level details

- The SPI flash is shared between BIOS, ME and GbE
- For security, BIOS (and OS) should not have access to ME region
- The chipset enforces it using information in the Descriptor region
- The Descriptor region must be at the lowest address of the flash and contain addresses and sizes of other regions, as well as their mutual access permissions.
**ME: Low-level details**

- ME region itself is not monolithic
- It consists of several partitions, and the table at the start* describes them

<table>
<thead>
<tr>
<th>Partition table header</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

*Starting from ME 3.x the table begins at offset 0x10 (table version 2.0)
### ME: Low-level details

<table>
<thead>
<tr>
<th>Partition type (Flags&amp;0x7F):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>

- **ME Flash Partition Table**
  - NumEntries: 10
  - Version: 2.0
  - EntryType: 10
  - HeaderLen: 30
  - Checksum: 9F
  - FlashCycleLifetime: 7
  - FlashCycleLimit: 100
  - UMASize: 16
  - Flags: FFFFFFFE07
    - EFFS present: 1
    - ME Layout Type: 3
  - Partition: 'FOVD'
  - Owner: 'KRID'
  - Offset/size: 00000400/00001C00
  - TokensOnStart: 00000001
  - MaxTokens: 00000001
  - ScratchSectors: 00000000
  - Flags: 0783
    - Type: 3 (Generic)
    - DirectAccess: 1
    - Read: 1
    - Write: 1
    - Execute: 1
    - Logical: 0
    - WOPDisable: 0
    - ExclBlockUse: 0
ME: Low-level details

- Code partitions have a header called "manifest"
- It contains versioning info, number of code modules, but also an RSA signature of the whole partition
- Format of the header is very close to TXT AC modules

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Type</td>
<td>SubType</td>
<td>HdrLen</td>
<td>HdrVer</td>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Vendor</td>
<td>Date</td>
<td>Size</td>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>NumMods</td>
<td>Version</td>
<td>Reserved==</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>&lt;=Reserved</td>
<td>KeySize</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80-17F</td>
<td></td>
<td></td>
<td></td>
<td>RsaPubKey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>RsaPubExp</td>
<td></td>
<td>RsaSig==&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280</td>
<td>&lt;=RsaSig</td>
<td></td>
<td>PartitionName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An example code partition header

Module Type: 4, Subtype: 0
Header Length: 0xA1 (0x284 bytes)
Header Version: 1.0
Flags: 0x00000000 [production signed] [production flag]
Module Vendor: 0x8086
Date: 20120705
Total Manifest Size: 0xFD (0x3F4 bytes)
Tag: $MN2
Number of modules: 2
Version: 8.1.0.1265
Unknown data 1: [0L, 1L, 2L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L]
Key size: 0x40 (0x100 bytes)
Scratch size: 0x01 (0x4 bytes)
RSA Public Key: [skipped]
RSA Public Exponent: 17
RSA Signature: [skipped]
Partition name: MDMV
Unknown data 2: [0L, 0L]
ME: Low-level details

- The format of module headers depends on the version (header tag $MAN$ or $MN2$)
- Module headers include module name, hash, sizes (compressed and uncompressed), flags and runtime info (load address, entrypoints)
- Modules can be stored uncompressed, or compressed with LZMA or Huffman

<table>
<thead>
<tr>
<th>Header tag:</th>
<th>$MME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module name:</td>
<td>JOM</td>
</tr>
<tr>
<td>Hash:</td>
<td>AC A3 [...] C1 6C</td>
</tr>
<tr>
<td>Offset:</td>
<td>0x00015F7A</td>
</tr>
<tr>
<td>Data length:</td>
<td>0x00019F6D</td>
</tr>
<tr>
<td>LoadBase:</td>
<td>0x200B1000</td>
</tr>
<tr>
<td>Flags:</td>
<td>0x0012D42A</td>
</tr>
<tr>
<td>Power Type:</td>
<td>POWER_TYPE_M0_ONLY (1)</td>
</tr>
<tr>
<td>Compression:</td>
<td>COMP_TYPE_LZMA (2)</td>
</tr>
<tr>
<td>API Type:</td>
<td>API_TYPE_KERNEL (2)</td>
</tr>
</tbody>
</table>
There have been two generations of the processor core, and corresponding changes in firmware layout.

<table>
<thead>
<tr>
<th></th>
<th>Gen 1</th>
<th>Gen 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME versions</td>
<td>1.x-5.x</td>
<td>6.x-8.x</td>
</tr>
<tr>
<td>Core</td>
<td>ARCTangent-A4</td>
<td>ARCTangent-A5(?)</td>
</tr>
<tr>
<td>ISA</td>
<td>ARC (32-bit)</td>
<td>ARCompact (32/16)</td>
</tr>
<tr>
<td>Manifest tag</td>
<td>$MAN</td>
<td>$MN2</td>
</tr>
<tr>
<td>Module header tag</td>
<td>$MOD</td>
<td>$MME</td>
</tr>
</tbody>
</table>
The OS running on the chip is ThreadX RTOS from Express Logic.

OS provides APIs for managing threads (tasks), semaphores, message queues, event flags, timers, memory allocations etc.

The ME firmware wraps those APIs in a module called KERNEL, and uses it from other modules (via tables of pointers).

Express Logic provides a demo version (binary only) of ThreadX for ARC, which helps in identifying APIs in ME.

Unfortunately Gen2 uses the Huffman compression (which I have not figured out yet) for the KERNEL :(

So the going is somewhat slow for the newer firmwares.
ME: communications

- If AMT option is enabled, ME listens for packets on several ports (e.g. 16992 for HTTP and 16993 for HTTPS) for HTTP requests from browsers (for Web UI) or SOAP requests.
- Since it has a separate IP and MAC for the OOB interface, this does not interfere with the host.
- ME is also exposed by the chipset as a PCI device to the CPU, and can exchange messages with it using Host Embedded Controller Interface (HECI) protocol over a memory-mapped IO area (MMIO).
- The protocol itself is described in public documentation [DCMI-HI], but the higher-level messages are not well documented.
- ME can expose various clients to the host, each identified by a unique UUID or a numeric ID, and host can talk to each client independently.
- Several core clients have fixed low IDs, the rest gets dynamic numbers.
ME: communications

- An example of enumerating clients (FreeBSD):

```plaintext
heci0: <Intel 82G33/G31/P35/P31 Express HECI/MEI Controller> mem
0xd0526100-0xd052610f irq 16 at device 3.0 on pci0
heci0: using MSI
heci0: [ITHREAD]
heci0: found ME client at address 0x02:
heci0:  status = 0x00
heci0:  protocol_name(guid) = BB875E12-CB58-4D14-AE93-8566183C66C7
heci0: found ME client at address 0x03:
heci0:  status = 0x00
heci0:  protocol_name(guid) = A12FF5CA-FACB-4CB4-A958-19A23B2E6881
heci0: found ME client at address 0x06:
heci0:  status = 0x00
heci0:  protocol_name(guid) = 9B27FD6D-EF72-4967-BCC2-471A32679620
heci0: found ME client at address 0x07:
heci0:  status = 0x00
heci0:  protocol_name(guid) = 55213584-9A29-4916-BADF-0FB7ED682AEB
heci0: found ME client at address 0x27:
heci0:  status = 0x00
heci0:  protocol_name(guid) = 05B79A6F-4628-4D7F-899D-A91514CB32AB
```
A list of some of the known clients, gathered from headers and other sources

<table>
<thead>
<tr>
<th>Fixed ID</th>
<th>GUID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8e6a6715-9abc-4043-88ef-9e39c6f63e0f</td>
<td>MKHI</td>
</tr>
<tr>
<td>9</td>
<td>42b3ce2f-bd9f-485a-96ae-26406230b1ff</td>
<td>ICC</td>
</tr>
<tr>
<td>9</td>
<td>d2ea63bc-5f04-4997-9454-8cadf4e3ef8a</td>
<td>Thermal</td>
</tr>
<tr>
<td>9</td>
<td>309dcde8-ccb1-4062-8f78-600115a34327</td>
<td>Firmware Update</td>
</tr>
<tr>
<td>9</td>
<td>05b79a6f-4628-4d7f-899d-a91514cb32ab</td>
<td>Watchdog</td>
</tr>
<tr>
<td>9</td>
<td>6733a4db-0476-4e7b-b3af-bcfc29bee7a7</td>
<td>LME</td>
</tr>
<tr>
<td>9</td>
<td>12f80028-b4b7-4b2d-aca8-46e0ff65814c</td>
<td>PTHI (AMTHI)</td>
</tr>
<tr>
<td>9</td>
<td>3d98d9b7-1ce8-4252-b337-2eff106ef29f</td>
<td>LMS</td>
</tr>
<tr>
<td>9</td>
<td>6b5205b9-8185-4519-b889-d98724b58607</td>
<td>QST</td>
</tr>
<tr>
<td>9</td>
<td>0f908627d-13bf-4a04-0b91f-0a64e9245323d</td>
<td>CLS</td>
</tr>
<tr>
<td>9</td>
<td>3c4852d6-d47b-4f46-b05e-b5edc1aa430a</td>
<td>TDT (AT-p)</td>
</tr>
</tbody>
</table>
ME: communications

- One of the main users of the HECI interface is the BIOS
- It has to allocate the UMA memory for ME, protect it, and notify ME about it
- It also needs to tell ME about various events, including End-Of-POST (EOP)
- If not disabled at manufacturing time, BIOS can also ask ME to temporarily open its flash region for reading and writing; this functionality is intended to allow ME region updates, and is called Host ME Region Flash Protection Override (HMRFPO)
- An optional module inside BIOS, MEBx (ME BIOS Extension) provides a UI for the user to configure various ME options. It also uses HECI to communicate with ME
- Thus, reverse-engineering BIOS is a good source for info about ME communications
ME: Security
ME: Security

- ME includes numerous security features
- Code signing: all code that is supposed to be running on the ME is signed with RSA and is checked by the boot ROM

“During the design phase, a Firmware Signing Key (FWSK) public/private pair is generated at a secure Intel Location, using the Intel Code Signing System. The Private FWSK is stored securely and confidentially by Intel. Intel AMT ROM includes a SHA-1 Hash of the public key, based on RSA, 2048 bit modulus fixed. Each approved production firmware image is digitally signed by Intel with the private FWSK. The public FWSK and the digital signature are appended to the firmware image manifest.

At runtime, a secure boot sequence is accomplished by means of the boot ROM verifying that the public FWSK on Flash is valid, based on the hash value in ROM. The ROM validates the firmware image that corresponds to the manifest’s digital signature through the use of the public FWSK, and if successful, the system continues to boot from Flash code.”

ME: Security

- ME requires some RAM to put unpacked code and runtime variables (MCU's own memory is too limited and slow)
- This memory is reserved by BIOS on ME's request and cannot be accessed by the host CPU once locked.

<table>
<thead>
<tr>
<th>18:12</th>
<th>RV</th>
<th>U</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>RWO</td>
<td>0</td>
<td>Enable for Intel® ME memory region</td>
</tr>
<tr>
<td>10</td>
<td>RWO</td>
<td>0</td>
<td>Lock for Intel ME memory region base/mask. This bit is only cleared upon a reset. MESEGMASK and MESEGBASE cannot be changed once this bit is set.</td>
</tr>
<tr>
<td>9:0</td>
<td>RV</td>
<td>0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

- A memory remapping attack was demonstrated by Invisible Things Lab in 2009, but it doesn't work anymore
- Cold boot attack is probably still possible, though...
- An open question: how does it work with the integrated memory controller on the newer chips?
ME: Security

- Flash access is limited by the chipset according to the flags in the descriptor region (start of the flash chip), and normally ME region is not accessible to others.
- Since the descriptor region itself is marked read-only at end of manufacturing, changing permissions is not trivial.
- One obvious solution is to use a hardware flash programmer to write to the chip directly, bypassing CPU and chipset. This might require unsoldering the chip, however.
- Another option is the HMRFPO message which asks ME to unlock the flash temporarily, but it's tricky to use because it only works before End-Of-POST.
Getting along with the BIOS

- I decided to find a place where the BIOS sends End-Of-Post to the ME
- Extracted BIOS with 7-zip (UEFI Firmware Filesystem)
- Searched for "Post", both ANSI and Unicode
- A strange file appears...
Getting along with the BIOS

- The file contains bunch of Unicode strings, first in English then in couple of other languages
- The strings refer to the BIOS setup items
- File appears next to a .efi executable, meaning they were two sections of the flash file "Setup".
- So obviously this is a kind of a resource file for the BIOS setup UI
- Turns out that (U)EFI provides a standard way to encode strings and forms for UI, called HII (Human Interface Infrastructure)
- And someone already wrote tools[1] to parse them...

Getting along with the BIOS

- After some hacking of the scripts (apparently there were some updates in the format) dumped a list of strings and forms
- And here's the option we need:

```
Suppress If
EQ [0xdb<1>] == 0x0
One Of [0xdc<1>] u'End of Post Message'
\Help text: u'End of Post Message Help'
Option 'Disabled' = 0x0 Flags 0x10 Key 0x0
Option 'Enabled' = 0x1 Flags 0x13 Key 0x0
End One Of
End If
```

- However, it doesn't seem to be present in the actual UI?
Getting along with the BIOS

- This setting is a part of a form named 'ME Subsystem'
- Scrolling a bit around, we find:

```
Suppress If
LIST [0xdb<1>] in (0x0,0x1)
Reference: 'ME Subsystem' Form ID 0x1a Flags 0x0 Key 0x0
\Help text: u'ME Subsystem Parameters'
End If
```

- So, the form is not shown if the byte in the Setup variable at offset 0xDB is either 0 or 1.
- One solution is to patch the form bytecode, pack the file back into the BIOS (updating the checksums) and flash the new BIOS
- But this is rather involved and risky. Is there an easier way?
Getting along with the BIOS

- Examining and editing UEFI variables is rather awkward but doable with the EFI shell and command "dmpstore"

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
000000d0 00 00 00 00 00 00 01 00 00 00 01
```

- Changing EFI vars is much easier than patching actual files in the FFS. Also, no need to reflash.
- Since neither 0 nor 1 will show the form, let's put something else in there... for example, 0xFF

```
> dmpstore Setup -s temp.bin
> hexedit temp.bin
> dmpstore Setup -l temp.bin
> exit
```

- Did it work?
Getting along with the BIOS

Intel ME Subsystem Configuration

ME Version: 7.0.4.1197

ME Subsystem: Enabled

ME Temporary Disable: Disabled

ME Temporary Disable: Disabled

End of Post Message: Enabled

Execute MEBx: Disabled

Integrated Clock Chip Configuration
Getting along with the BIOS

- Now we can change the ME options and disable End-Of-POST
- One minor issue: when you return from the "ME Subsystem" form, the menu item disappears :)
- This happens because the byte 0xDB gets set to 1 (or 0, if you disable ME) again, triggering the "Suppress If" opcode
- So if you need to go there once more, you need to do the dmpstore/hexedit trick again
- By the way, instead of going through the menus we could directly set the necessary value in the Setup variable (byte at offset 0xDC)
Rebooting after changing "End of Post Message" to "Disabled":

```
Get flash master region access status...done
Host Read Access to ME: Disabled
Host Write Access to ME: Disabled
SPI Flash ID #1: EF4016
SPI Flash ID USCC #1: 20052001
SPI Flash BIOS USCC: 20052005
Protected Range Register Base #0 0x0
Protected Range Register Limit #0 0x0
Protected Range Register Base #1 0x0
Protected Range Register Limit #1 0x0
Protected Range Register Base #2 0x0
Protected Range Register Limit #2 0x0
Protected Range Register Base #3 0x0
Protected Range Register Limit #3 0x0
Protected Range Register Base #4 0x0
Protected Range Register Limit #4 0x0
BIOS boot State: Pre Boot
OEM Id: 80000000-0000-0000-0000-000000000000
```
Getting along with the BIOS

- Okay, we have our ME in desired state, what now?
- The specifics of the HMRFPO message are not available in public documentation
- However, some BIOS updates exist that allow updating ME version from 7.0 to 8.0
- ME cannot update itself to the next major version, so this must be done by external (to the ME) code
- From reading the "Bios ME7 to ME8 update SOP" for MSI boards it's clear that the ME update happens on the first boot of the new BIOS
- So the code must be there somewhere...
Getting along with the BIOS

- Several days of reversing later...
- Found the new ME partition (stored as a file in the UEFI volume)
- Found the code that does the ME update ("Updating BIOS ME, please wait")
- Found code which seems to talk to ME and send commands not mentioned in documentation
- Found code in ME which handles these messages (probably)
- Converted an AMT SDK sample to send similar commands
- Unfortunately, didn't work on my test hardware (ASUS)
- However, it was a good learning experience!
A different approach

- After I went through this, I accidentally found a mention that the newest BIOS for my board contains ME 8.0 (this fact was not mentioned in Asus' release notes)
- As a nice side effect, this update completely opens the ME region!
- So now I can read and write the ME region freely (using Intel's FPT)
- I can also analyze the update process in more detail and figure out how it works around the ME lock on the old version
Poking the flash

- One of the partitions in the ME region is "EFFS"
- It contains in turn other, virtual partitions with tags beginning with "NV" (non-volatile variables) and "BI" (block I/O), used by the software components of ME
- Some of these variables are used to enable and disable various ME features which usually depend on the specific chipset model (a single ME binary is used on many configurations)
- For example, ME on my board includes modules TDT (Anti-Theft) and PAVP (Protected Audio/Video Path), but they're disabled in software
- I tried changing some obvious bits, but it seems it's not that simple...
Results

- I have not managed to run my own rootkit on the ME (yet)
- However, I've learned a lot about it and I hope to achieve it in future
- Intel seems to have done a good job on security so far, but there's a lot of code in there (now up to 5MB, compressed)
- I made some tools that should help others in research:
  - ME ROM dumper/extractor
  - ARC processor module for IDA
ME dumper/extractor

- Written in Python
- Supports parsing of the following formats:
  - Full SPI flash image (signature 5A A5 F0 0F)
  - Separate ME region (signature $FPT)
  - Individual ME code module ($MN2 or $MAN)
- Prints detailed header info
- prepares LZMA-compressed modules for easy unpacking with 7-zip
# ME dumper/extractor

<table>
<thead>
<tr>
<th>Header tag:</th>
<th>$MME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module name:</td>
<td>JOM</td>
</tr>
<tr>
<td>Hash:</td>
<td>FB 49 10 CB 04 C8 62 9D BE 53 BB 7A CF 0C 6A D4 1F F9 92 A7 AD 52 2A 55 FE F6 71 74 06 F0 0C 64</td>
</tr>
<tr>
<td>Unk34:</td>
<td>0x20157000</td>
</tr>
<tr>
<td>Offset:</td>
<td>0x0001198E</td>
</tr>
<tr>
<td>Unk38:</td>
<td>0x00029000</td>
</tr>
<tr>
<td>Data length:</td>
<td>0x000133CA</td>
</tr>
<tr>
<td>Unk44:</td>
<td>0x00029518</td>
</tr>
<tr>
<td>Unk48:</td>
<td>0x00029518</td>
</tr>
<tr>
<td>LoadBase:</td>
<td>0x20159000</td>
</tr>
<tr>
<td>Flags:</td>
<td>0x0012D42A</td>
</tr>
</tbody>
</table>

- **Unknown B0**: 0
- **Power Type**: POWER_TYPE_M0_ONLY (1)
- **Unknown B3**: 1
- **Compression**: COMP_TYPE_LZMA (2)
- **Stage**: STAGE 8 (8)
- **API Type**: API_TYPE_KERNEL (2)
- **Unknown B14**: 1
- **Unknown B15**: 1
- **Privileged**: 0
- **Unknown B17_19**: 1
- **Unknown B20_21**: 1
ARC processor: objdump

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200f4a2a</td>
<td>cmp_s</td>
<td>r15,2</td>
<td>0x200f4a2c</td>
<td>bne_s</td>
<td>0x200f4b54</td>
</tr>
<tr>
<td>0x200f4a2e</td>
<td>ld_s</td>
<td>r0,[r13,4]</td>
<td>0x200f4a30</td>
<td>cmp_s</td>
<td>r0,1</td>
</tr>
<tr>
<td>0x200f4a32</td>
<td>cmp_s</td>
<td>r0,4</td>
<td>0x200f4a36</td>
<td>bne_s</td>
<td>0x200f4b9e</td>
</tr>
<tr>
<td>0x200f4a38</td>
<td>ld_s</td>
<td>r13,pcl,0x178</td>
<td>0x200f4a3a</td>
<td>add3</td>
<td>r1,r14,20</td>
</tr>
<tr>
<td>0x200f4a3e</td>
<td>mov_s</td>
<td>r2,1</td>
<td>0x200f4a40</td>
<td>add3</td>
<td>r0,r13,55</td>
</tr>
<tr>
<td>0x200f4a44</td>
<td>bl</td>
<td>0x200f1474</td>
<td>0x200f4a48</td>
<td>mov_s</td>
<td>r0,r13</td>
</tr>
<tr>
<td>0x200f4a4a</td>
<td>add</td>
<td>r0,r0,0x194</td>
<td>0x200f4a4e</td>
<td>ld_s</td>
<td>r15,pcl,0x168</td>
</tr>
<tr>
<td>0x200f4a50</td>
<td>mov_s</td>
<td>r1,r14</td>
<td>0x200f4a52</td>
<td>add</td>
<td>r1,r1,173</td>
</tr>
<tr>
<td>0x200f4a56</td>
<td>mov_s</td>
<td>r2,2</td>
<td>0x200f4a58</td>
<td>mov_s</td>
<td>r12,100</td>
</tr>
<tr>
<td>0x200f4a5a</td>
<td>sub</td>
<td>r3,r15,r12</td>
<td>0x200f4a62</td>
<td>mov</td>
<td>r4,24</td>
</tr>
</tbody>
</table>

What are the results of underlined instructions?
### ARC processor: IDA

```
<table>
<thead>
<tr>
<th>ROM</th>
<th>0F4A2A</th>
<th>cmp</th>
<th>r15, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>0F4A2C</td>
<td>bnz</td>
<td>loc_200F4B54</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A2E</td>
<td>ld</td>
<td>r0, [r13,4]</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A30</td>
<td>cmp</td>
<td>r0, 1</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A32</td>
<td>cmp.nz</td>
<td>r0, 4</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A36</td>
<td>bnz</td>
<td>loc_200F4B9E</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A38</td>
<td>ld</td>
<td>r13, =dword_200FF44C</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A3A</td>
<td>add</td>
<td>r1, r14, (aHcismaphore - 0x200F8C4C) # &quot;HciSemaphore&quot;</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A3E</td>
<td>mov</td>
<td>r2, 1</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A40</td>
<td>add</td>
<td>r0, r13, (g_HciSemaphore - 0x200FF44C)</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A44</td>
<td>bl</td>
<td>create_semaphore</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A48</td>
<td>mov</td>
<td>r0, r13</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A4A</td>
<td>add</td>
<td>r0, r0, (g_HciInputQueue - 0x200FF44C)</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A4E</td>
<td>ld</td>
<td>r15, =0x200FF648</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A50</td>
<td>mov</td>
<td>r1, r14</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A52</td>
<td>add</td>
<td>r1, r1, (aHciinputqueue - 0x200F8C4C) # &quot;HciInputQueue&quot;</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A56</td>
<td>mov</td>
<td>r2, 2</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A58</td>
<td>mov</td>
<td>r12, (0x200FF648 - unk_200FF5E4)</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A5A</td>
<td>sub</td>
<td>r3, r15, r12 ; unk_200FF5E4</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A5E</td>
<td>mov</td>
<td>r4, 0x18</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A62</td>
<td>bl</td>
<td>create_queue</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A66</td>
<td>add</td>
<td>r0, r13, (g_HciHeciEventFlags - 0x200FF44C)</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A6A</td>
<td>mov</td>
<td>r1, r14</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A6C</td>
<td>add</td>
<td>r1, r1, (aHcihecievntflags - 0x200F8C4C) # &quot;HciHeciEventFlags&quot;</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A70</td>
<td>bl</td>
<td>create_event_flags</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A74</td>
<td>add</td>
<td>r0, r13, (g_HciBufferQueue - 0x200FF44C)</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A78</td>
<td>mov</td>
<td>r1, r14</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A7A</td>
<td>add</td>
<td>r1, r1, (aHcibufferqueue - 0x200F8C4C) # &quot;HciBufferQueue&quot;</td>
</tr>
<tr>
<td>ROM</td>
<td>0F4A7E</td>
<td>mov</td>
<td>r2, 1    # message_size</td>
</tr>
</tbody>
</table>
```
ARC processor: objdump

```
0 [main] arc-elf32-objdump_arccompact 5920 exception::handle: Exception: SIGSEGV, Segmentation fault.
0x61112b58 in strcpy () from /usr/bin/cygwin1.dll
(gdb)
```
ARC processor module for IDA

- Supports ARCTangent-A4 (older, 32-bit only instructions) and ARCompact (newer, mixed 32/16-bit ISA)
- Tracks changes to SP register and creates local variables
- Handles switch tables
- Tracks register values to find more cross-references
- Inlines constant pool loads (PC-relative) for convenience
- In general, makes life not constant pain
- Not production quality yet, but hopefully will appear in the next version of IDA
Future work

- Dynamic Application Loader
  - New feature in 7.1/8.0 firmware: load **Java applets** and run them inside ME
  - Used for things like PIN entry UI and remote authentication
  - The applets provided by Intel are signed, but it's one more vector of entry...
- EFFS parsing and modifying
  - Most of the ME state is stored there
  - If we can modify flash, we can modify EFFS
  - Critical variables are protected from tampering but the majority isn't
  - Complicated format because of flash wear leveling
Future work

- Huffman compression
  - Used in newer firmwares for compressing the kernel and some other modules
  - Couldn't find decompression code; some whitepapers mention hardware decompression...
  - Still, Huffman is a pretty simple protocol, so should be doable from just the compressed data
- ME ↔ Host protocols
  - Most modules use different message format
  - A lot of undocumented messages; some modules seem to be not mentioned anywhere
  - Some client software has very verbose debugging messages in their binaries...
Future work

- BIOS RE
  - In early boot stages ME accepts some things which are not possible later
  - Reversing BIOS modules that talk to ME is a good source of info
  - Even better would be to run custom code early
  - Big room for improvement in tools
- Simulation and fuzzing
  - Open Virtual Platform (www.ovpworld.org) has modules for ARC600 and ARC700 (ARCompact-based)
  - They claim that it's easy to extend the models with emulation for custom hardware
  - The simulator has GDB stub for debugging
  - Debugging and fuzzing should be possible
References and links

http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/100402-Vassilios_Ververis-with-cover.pdf
http://marcansoft.com/blog/2009/06/enabling-intel-vt-on-the-aspire-8930g/
http://flashrom.org/trac/flashrom/browser/trunk/Documentation/mysteries_intel.txt
http://review.coreboot.org/gitweb?p=coreboot.git;a=blob;f=src/southbridge/intel/bd82x6x/me.c
http://download.intel.com/technology/product/DCMI/DCMI-HI_1_0.pdf
Thank you!

Questions?

igor@hex-rays.com
skochinsky@gmail.com