AMD x86 SMU firmware analysis

Do you care about Matroshka processors?

Rudolf Marek

r . marek @ assembler.cz

Outline

- History of platform processors
- Current platform processors
- Analysis
  - Hardware
  - Firmware

The (little) helpers to x86

- x86 the main CPU – since 1978
- IBM PC XT
  - Intel 8048 – since 1983
- IBM PC AT
  - Intel 8042 – since 1984, keyboard, A20, reset
- Notebooks
  - Various embedded controllers
  - usually 8051 compatible or H8 CPUs

Where is firmware found?

- In peripherals
  - GPU, wireless, network, storage, USB etc...
- In system controllers & integrated processors on x86
  - Intel
    - Management Engine/AMT etc...(ME, ARC, SPARC)
  - AMD
    - System Management Unit (SMU, ?)
    - Integrated Microcontroller (IMC, 8051, southbridge)
    - USB3.0 controllers (USB, your homework)
    - Platform Security Processor (PSP, ARM Cortex A5 with TrustZone)
  - ARM, PowerPC too (DMA or channel controllers with firmware)

Bugs & malware

Where?	Bugs	Malware
Application	X	X
App Libraries	X	X
Operating System	X	X
Platform processors	X	?

Call for security awareness

- Increasing security problem for the platforms
  - Firmware blobs are widely used on x86, PowerPCs, ARMS etc...
- Hardware engineers != Software engineers
- First opensource firmware on unusal processors:
  - Psychson (badUSB) – 8051, USB3.0 controller
  - Sprite_tm – ARM, MMU-less Linux on the Harddrive PCB
- And closed source firmware:
  - SNA SAN ANS ASN NSA NAS ...

Begin of the story

- Someone reads books...
- Someone also reads datasheets...
  - AMD BIOS and Kernel Developers Guide (BKDG)
    - for fam15h & fam16h

“The system management unit (SMU) is a subcomponent of the northbridge that is responsible for a variety of system and power management tasks during boot and runtime. The SMU contains a microcontroller to assist”

  - The Northbridge is part of the CPU
  - a processor inside a processor (or SoC) - matroshka processor!

Which microcontroller?

- Lets use google: "AMD" "system management unit"
- You get links to various other datasheets
  - with bit more details, we need them later
- But also to linkedin.com

> “Developed a NLMS Adaptive Filter in firmware C for an embedded **Lattice LM32 microprocessor** in the **SMU IP**, to dynamically compute the coefficients used for calculating the GPU dynamic power consumption. This also includes creating a Matlab model of the adaptive filter and running simulations with real silicon data to verify the algorithm functionality.”

LatticeMico lm32

- LatticeMico32 Open, Free 32-Bit Soft Processor
- Check “LatticeMico32 Processor Reference Manual”
- SDK from LatticeMico System for Diamond 3.3
- You can compile own toolchain:
  - gcc, objdump, as, gdb...
- http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx

LM32 Architecture

- 32-bit processor, 32-bit instruction
- 32 general-purpose registers (R0-R31)
  - R0 – zero
  - R26 – GP (global pointer)
  - R27 – FP (frame pointer)
  - R28 – SP (stack pointer)
  - R29 – RA (return address, like LR on PowerPC/ARM)
  - R30 – EA (exception address)
  - R31 – BA (breakpoint address)

This is nice ... but where is the firmware?

- Try to locate the flash chips
- Try the BIOS image
  - part of it is AMD AGESA
  - AMD Generic Encapsulated Software Architecture (AGESA)
  - Blob which contains all AMD silicon initialization
- Download various BIOSes for the target platform (FM2 motherboard)
- Why not to try text search?

This is nice … but where is the firmware?

- Try searching for* “SMU”
- You find it for every processor family supported:
  - Trinity, Richland, Kaveri, Kabini etc

```
007acf30  00 00 00 00 5f 53 4d 55  5f 53 4d 55 80 dd 00 00  |...._SMU_SMU....|
007acf40  00 20 00 00 00 00 01 00  e7 62 54 d9 54 4e 00 00  |. .......bT.TN..|
007acf50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
007acf60  00 00 00 00 0a 00 0a 00  40 00 00 00 14 37 00 00  |........@....7..|
007acf70  00 01 01 00 10 98 c4 cd  97 53 2d 93 cc 9d 09 f7  |.........S-.....|
007acf80  c4 6e ea f7 1e a0 9c f8  c0 d9 01 00 d0 da 01 00  |.n..............|
007acf90  00 00 00 00 f1 da 01 00  00 db 01 00 14 da 01 00  |................|
007acfa0  38 dc 01 00 00 00 00 00  00 00 00 00 00 00 00 00  |8...............|
007acfb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
007ad060  55 aa 55 aa 00 00 00 98  00 00 00 98 00 00 00 d0  |U.U.............|
```

* Note: To get AGESA version search for “AGESA” string

How to access it from the x86 processor?

- Check any recent AMD documentation (BKDG 15h, 16h)
- There is a gateway to the lm32 address space
  - Using PCI registers 0xB8 (address) and (0xBC data) of PCI 0:0.0
  - Lets write a simple utility and see what we get
- If it's not documented it does not mean it does not exist
  - Just a delay
  - **Please buy only platforms which are documented!**

SMU address space dump

```
00000000  55 55 aa aa 55 55 aa aa  55 55 aa aa 55 55 aa aa  |UU..UU..UU..UU..|
*
00010000  0a 00 0a 00 40 00 00 00  14 37 00 00 00 01 01 00  |....@....7......|
00010010  10 98 c4 cd 97 53 2d 93  cc 9d 09 f7 c4 6e ea f7  |.....S-......n..|
00010020  1e a0 9c f8 c0 d9 01 00  d0 da 01 00 60 f1 01 00  |............`...|
00010030  f1 da 01 00 00 db 01 00  14 da 01 00 38 dc 01 00  |............8...|
00010040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000100f0  00 00 00 00 00 00 00 00  00 00 00 00 55 aa 55 aa  |............U.U.|
00010100  55 55 aa aa 55 55 aa aa  55 55 aa aa 55 55 aa aa  |UU..UU..UU..UU..|
*
0001dc50  55 55 aa aa ac c3 01 00  40 2f 01 00 fc 72 01 00  |UU......@/...r..|
0001dc60  78 ac 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |x...............|
0001dc70  3c b1 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |<...............|
0001dc80  74 af 01 00 38 2f 01 00  88 07 01 00 88 07 01 00  |t...8/..........|
0001dc90  88 07 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |................|
0001dca0  88 07 01 00 88 07 01 00  88 07 01 00 78 d9 01 00  |............x...|
0001dcb0  7c d9 01 00 88 07 01 00  bc 21 01 00 88 07 01 00  ||........!......|
0001dcc0  88 07 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |................|
0001dcd0  88 07 01 00 88 07 01 00  44 33 01 00 a0 42 01 00  |........D3...B..|
0001dce0  88 07 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |................|
*
0001dd30  88 07 01 00 88 07 01 00  00 00 00 00 00 01 00 00  |................|
0001dd40  c1 46 0c 00 01 00 00 00  00 00 00 00 00 00 00 00  |.F..............|
0001dd50  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
0001dd60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0001de50  00 00 00 00 00 00 00 00  c4 da 01 00 01 01 00 00  |................|
```

Address space of the SMU

Start	End	Usage	Comment
0x00000000	0x0000ffff	Unknown	Contains repeated pattern of 0x55 0xaa 0x55 0xaa
0x00010000	0x000100ff	Visible firmware header	Same as found in BIOS image
0x00010100	0x0001dc53	Hidden firmware???	0x55 0xaa...
0x0001dc54	0x0001ffff	“Random” data	
0x00020000	0x000203ff	Unknown	0x55 0xaa...
0x00020400	0x...?	Unknown	
0xe0000000	?	Hardware registers	

The Firmware Header

Offset	Value	Usage
0x0000	0x0a 0x00 0x0a 0x00	Version (Minor, Major)
0x0004	0x40 0x00 0x00 0x00	Length of header (in 4 bytes)
0x0008	0x14 0x37 0x00 0x00	Length of body (in 4 bytes, without header)
0x000C	0x00 0x01 0x01 0x00	0x10100 Entrypoint?
0x0010	Random data	Checksum?
0x00FC	0x55 0xaa 0x55 0xaa	Signature
0x0100	0x00 0x00 0x00 0x98	First instruction?

```
00000000  0a 00 0a 00 40 00 00 00  14 37 00 00 00 01 01 00  |....@....7......|
00000010  10 98 c4 cd 97 53 2d 93  cc 9d 09 f7 c4 6e ea f7  |.....S-......n..|
00000020  1e a0 9c f8 c0 d9 01 00  d0 da 01 00 60 f1 01 00  |............`...|
00000030  f1 da 01 00 00 db 01 00  14 da 01 00 38 dc 01 00  |............8...|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 55 aa 55 aa  |............U.U.|
```

What is the first instruction?

- Maybe after the 256 bytes header?
- Then starts with:
  - 0x00 0x00 0x00 0x98 0x00 0x00 0x00 0x98
  - srui r0,r0,152
  - srui r0,r0,152
  - This is a garbage...
- But lets try Big Endian...
  - xor r0, r0
  - xor r0, r0
- Cool, it sets the R0 to be zero

Firmware image II

- Now transform image to ELF
  - Swap file to Big Endian
  - Create ELF
    - lm32-elf-objcopy -S -I binary -O elf32-lm32 -B lm32 --rename-section .data=.text,alloc,load,contents,code,readonly --change-address 0x10000 --set-start=0x100 $1 fw.elf
- Use objdump...

Firmware Image III

- It zeros R0, disables interrupts and exception vectors...

```
10100:      98 00 00 00     xor r0,r0,r0
10104:      98 00 00 00     xor r0,r0,r0
10108:      d0 00 00 00     wcsr IE,r0
1010c:      78 01 00 01     mvhi r1,0x1
10110:      38 21 01 00     ori r1,r1,0x100
10114:      d0 e1 00 00     wcsr EBA,r1
10118:      d1 21 00 00     wcsr DEBA,r1
1011c:      f8 00 00 39     calli 0x10200
```

Firmware Image IV

- And also stack...

```
10200:      98 00 00 00     xor r0,r0,r0
10204:      78 1c 00 01     mvhi sp,0x1
10208:      3b 9c eb fc     ori sp,sp,0xebfc
1020c:      78 1a 00 02     mvhi gp,0x2
10210:      3b 5a 5d 40     ori gp,gp,0x5d40
```

Lets analyze it further...

- Lets write a linker script which add symbols to our ELF!
  - Hint: Exception handling is described in the LM32 manual

```
  .text :.
  {
    . = ALIGN(4);
_start = 0x100;
RESET = 0x100;
BREAKPOINT = 0x120;
INSTRBUSERROR = 0x140;
WATCHPOINT = 0x160;
DATABUS_ERR = 0x180;
DIV = 0x1a0;
INT =  0x1c0;
SYSCALL = 0x1e0;
RESET_INIT = 0x200;
```

Analyze this...

- Typical Function prologue & epilogue
  - Of Interrupt routine
    - Saves/Restores all registers
    - Makes room on stack for local vars
  - Of normal routine
    - Saves/Restores only (r11-r26)
    - Makes room on stack for local vars

```
addi sp,sp,-16
sw (sp+16),r11
sw (sp+12),r12
sw (sp+8),r13
sw (sp+4),ra
…
lw r11,(sp+16)
lw r12,(sp+12)
lw r13,(sp+8)
lw ra,(sp+4)
ret
```

Great idea...

- Did I say that Lattice has a SDK?
- Lets examine it...
  - Crt0ram.S
  - Other C function to register ISR & IRQs
  - Lets analyze the binaries too (for fun)!
- Yes! AMD used that too.
- Lets fill way more to our linker script

```
MicoISRHandler = 0x448;
MicoISRInitialize = 0x500;
MicoRegisterISR = 0x54c;
MicoDisableInterrupt = 0x604;
MicoEnableInterrupt = 0x660;
MicoEnableInterrupts = 0x6d8;
MicoDisableInterrupts = 0x728;
```

Communicating with the firmware

- BIOS can invoke a SMU firmware service request
- Documented in BKDG, but only for a single request
  - Write a service number to a doorbell register
  - Toggle a bit in a doorbell register

SMU Firmware requests in detail

- Mostly power management related
  - Budget Aware Power Management (BAPM)
  - Usually using custom registers at 0x1f000-0x1ffff range
- Interesting and easy to understand services:
  - Flush data cache
  - Flush instruction cache
  - “Ping” request
    - Increase a byte by 1 in the diagnostic register

The Great Temptation

- At this moment we know a lot about the SMU
- Can we run our own code
  - For fun?
  - How?
  - What are the problems?
- Lets check the header again
  - Bytes 0x10 - 0x23 look quite random
  - Total of 20 bytes...
  - Or 160 bits!

The checksum

- Its 160 bits of quite random data
- Could be SHA1
- Tried different ways to compute that
  - With a header
  - Without a header
  - Little/big endian
  - Header with zeros
  - Never had the same value!
- Also, BIOS has own checksums...

Runtime code injection?

- Code and data segment is hidden from x86 CPU while operating!
- But, can we change the unprotected ranges?
  - Using the northbridge gateway
  - 0x1dc54 → 0x1ffff binary garbage
  - Does it contain data only?

The open segments dump

- The main CPU can access only two segments
  - The header – 256 bytes
  - And some data area at the end of the 64KB firmware area
    - Offset data
    - Garbage
    - BIOS communication area

```
<Header is here>
00000100  55 55 aa aa 55 55 aa aa  55 55 aa aa 55 55 aa aa  |UU..UU..UU..UU..|
*
0000dc50  55 55 aa aa ac c3 01 00  40 2f 01 00 fc 72 01 00  |UU......@/...r..|
0000dc60  78 ac 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |x...............|
0000dc70  3c b1 01 00 88 07 01 00  88 07 01 00 88 07 01 00  |<...............|
```

The Offset Data

- Offset data matches some function addresses in the firmware
- Wtf ?
- It is part of the binary image from BIOS
- Wtf 2?

```
0001dc50: aaaa5555
0001dc54: 0001c3ac
0001dc58: 00012f40
0001dc5c: 000172fc
0001dc60: 0001ac78
0001dc64: 00010788
0001dc68: 00010788
0001dc6c: 00010788
0001dc70: 0001b13c
0001dc74: 00010788
0001dc78: 00010788
0001dc7c: 00010788
0001dc80: 0001af74
```

Something went terribly wrong

- It looks like there are 256 bytes which are not protected
- The header is also 256 bytes
  - Firmware body has a length without the header!
- Does it allow us to modify the offset data during run-time?
  - Yes!
- Does something invoke a function pointers from offset data?
  - Yes! - The ISR dispatcher uses this for certain interrupts
  - There is firmware SMU request function handler!

How to run our program?

- Load it to the “unused” area at the end of 64KB
- Modify the function pointer in offset data
- When finished, return control to the original function to handle the SMU request for us...

The SMU request function

- Lets re-check the SMU request ISR:
  - ACK IRQ (0xe0003004)
  - Load Request number (0xe0003000)
  - Mask it by 0xfffe
  - Shift by 1 to right
  - Load the array base for SMU request function pointers
  - Shift the request number by 2 to left
  - Load function address to r2
  - Call r2
  - Wtf???
  - No check for bounds!
  - Signal Intdone and intack

```
mvhi r1,0xe000
mvi r3,1
ori r1,r1,0x3000
mvhi r12,0xe000
ori r12,r12,0x3004
sw (r12+0),r3
lw r3,(r1+0)
mvhi r1,0x1
ori r1,r1,0xfffe
and r3,r3,r1
mvhi r1,0x1
srui r3,r3,1
sli r3,r3,2
ori r1,r1,0xbad4
add r3,r3,r1
lw r2,(r3+0)
call r2
mvi r4,3
sw (r12+0),r4
lw r12,(sp+8)
lw ra,(sp+4)
addi sp,sp,8
ret
```

Something went terribly wrong - Again

- We control the SMU request value (15 bits!)
  - The offset of the SMU request table is also known
- We can simply invoke any function we like
  - Including injected functions!
- We need to compute a fake request number
  - It will be multiplied by 4 and added to known offset
  - At that address we save a pointer to our code, loaded elsewhere
  - Fun!

Address space of the SMU

Start	End	Usage	Comment
`0x00000000`	`0x0000ffff`	**Lets find out!**	**Contains repeated pattern of 0x55 0xaa 0x55 0xaa**
`0x00010000`	`0x000101ff`	Visible firmware header	Same as found in BIOS image
`0x00010100`	`0x0001dc53`	Hidden firmware???	0x55 0xaa...
`0x0001dc54`	`0x0001ffff`	“Random” data	
`0x00020000`	`0x000203ff`	Unknown	0x55 0xaa...
`0x00020400`	`0x...?`	Unknown	
`0xe0000000`	`?`	Hardware registers	

Dumping the secret part

- Lets write a program
  - New SMU service request
  - Copy 4 bytes from address we like
- Invoke this new service as many times as we like
- Dump and analyze the first 64KB which were not accessible

The secret ROM

- Most likely a ROM
- It has the same structure as the runtime firmware
- Initialization seems to be more complex
- It only implements one SMU firmware request 0
  - This request verifies the firmware loaded by BIOS
  - Lets finally get idea about:
    - What kind and how the hash is used!
    - Why there is the problem with 256 byte offset

Firmware authentication fun(ction)

- There are strange constants in the related functions
- Like 0x98badcfe, 0x10325476
- Like 0x36363636, 0x5c5c5c5c
- Lets use google again...

```
78 03 98 ba    mvhi r3,0x98ba
38 43 dc fe    ori r3,r3,0xdcfe
```

  - SHA1 and also HMAC uses this...
- How to make sense of big chunk functions?
  - Use QEMU...

QEMU

- It has support for lm32!
- Hack QEMU to support SMU memory layout
- Load QEMU with SMU firmware!
  - ROM and RAM part (0-64KB, 64-128KB)
- Start ROM SMU request function in QEMU
- Also load the firmware which we want to authenticate
- Let the fun begin and debug this via GDB...

The Authenticate request

- It loads data from firmware header
- Flushes the data cache
- Computes hash function
- Flushes data cache again
- Check the hash in the header – but using constant time algorithm
- Sets up the protection registers
  - Problem: Someone forgot to add 256 to the offset protection register
- Sets up reset vector to the authenticated firmware
- Signals back to BIOS the results

The header checksum

- In fact a hash
- Yes, debugging helped a lot
- The algorithm used is HMAC/SHA1 with a secret key!
  - The key is a symmetric key
- Whoever has the key...
  - Can sign their own firmware!
  - Now close your eyes...
    - No, really close your eyes, I'm watching you!
    - And wait for the next slide!

The secret

42?

The secret

Prezident Zeman je

K U N D A

The secret

- Remains a secret!
- Time to write some emails to AMD!
  - What to write?

    “To whom it may concern,

    I have a discovered a security vulnerability in the recent AMD processors which allows arbitrary code execution on the System Management Unit (SMU).”
  - Whom to write?
    - Why not to use linkedin again to find some engineer!
  - How to support the claim?
    - Change the SMU request ping function to add 0x42!
    - And fix the checksum hash!
    - Send it to AMD

Timeline

- Firmware was analyzed during Christmas 2013
- Bug was found sometime in the April 2014
- 30.4.2014 – Request to AMD sent, waiting for reply...
- 15.5.2014 – Reply!
- 16.5.2014 – Encrypted communication, sending details
- 09.7.2014 – AMD acknowledges the problem
- Occasional communication in the meanwhile...
- 25.11.2014 – AMD sends me a list of versions which contain the fix

AMD was responsive and helpful!

Fixed problems

- Both issues are fixed
- The firmware is padded so whole runtime firmware is correctly covered by protection cover (0x55 0xaa ...)
- The SMU request function checks bounds now
- Similar, but not the same problem fixed for Kabini and Kaveri CPUs

Fixed Versions

- The fixed SMU firmware is part of updated AMD AGESA
- **Please ask your mainboard vendor (OEM/ODM) for a fixed AGESA!**
  - This is the only way to push vendors to update BIOSes for older platforms!

Processor	AGESA version	SMU version *	CPU family
Trinity	1.1.0.7	10.14 (0x000a000e)	fam15h
Richland	1.1.0.7	12.18 (0x000c0012)	fam15h
Kabini	1.1.0.2	12.21 (0x000c0015)	fam16h
Kaveri	1.1.0.7	13.52 (0x000d0034)	fam15h

* Note: stored as LSB first in the BIOS image: 0x09 0x00 0x0a 0x00 means version 10.09

Questions?

Thank you!

r . marek @ assembler.cz

Do you care about Matroshka processors?

- This is where YOU can help
  - Be curious!
  - Analyze as much as you can!
  - It is quite fun!
- Be responsible
  - Try to contact the vendor (security@amd.com)
  - Inform them that you want to go with responsible disclosure
- Homework:
  - USB3 controller ? Any takers?