Practical Exploitation of Embedded Systems

Andrea Barisani
<andrea@inversepath.com>

Daniele Bianco
<daniele@inversepath.com>

em·bed

verb /em'bed/

em·bed·ded, past participle; em·bed·ded, past tense;

- implant within something else
- (often as adjective *embedded*) design and build as an integral part of a system or device

Embedded System

An embedded system is a computer system designed for specific control functions within a larger system. It is *embedded* as part of a complete device often including hardware and mechanical parts.

Source: Wikipedia

Examples

Routers, Printers, Point-of-Sales, Smart Cards, Automotive equipment, Avionics, etc.

Peripheral controllers (keyboard), LAN controllers, System Management controller, etc.

Employed OS range from standard Linux to real-time systems such as VxWorks, ThreadX, LynxOs, PikeOS.

Exploitation

Compromising Embedded Systems has been a “hot” topic for several years and plenty of presentations/material are available.

The general interest for exploitation ranges from feature enhancements to auditing purposes and, inevitably, malicious activity.

We focus on some unorthodox and difficult reverse engineering challenges encountered during the course of different penetration tests and the techniques to approach them.

Discovering debugging/console interfaces

The vast majority of debugging/programming ports on Embedded Systems are either serial interfaces (RS232) or JTAG.

The discovery and usage of interface pin-out for serial interfaces is straightforward.

With JTAG however the process of finding the interface pin-out can be complex and time consuming as board manufacturers sometime implement hardware protections (0 Ohm resistors / burned fuses) as well as software protections (custom initialization sequence) in order to prevent JTAG operations.

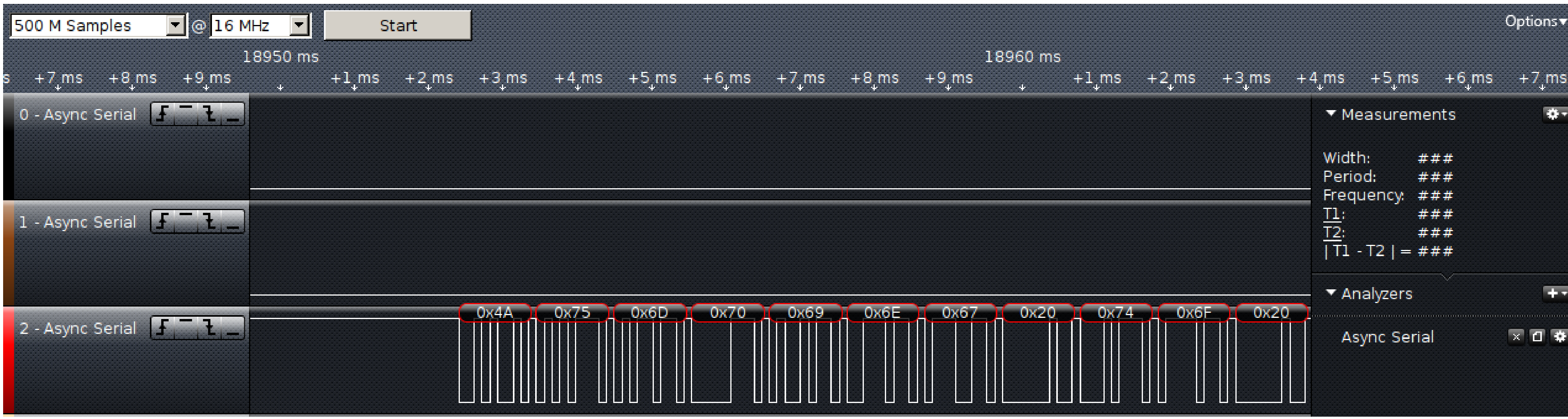
INVERSE PATH

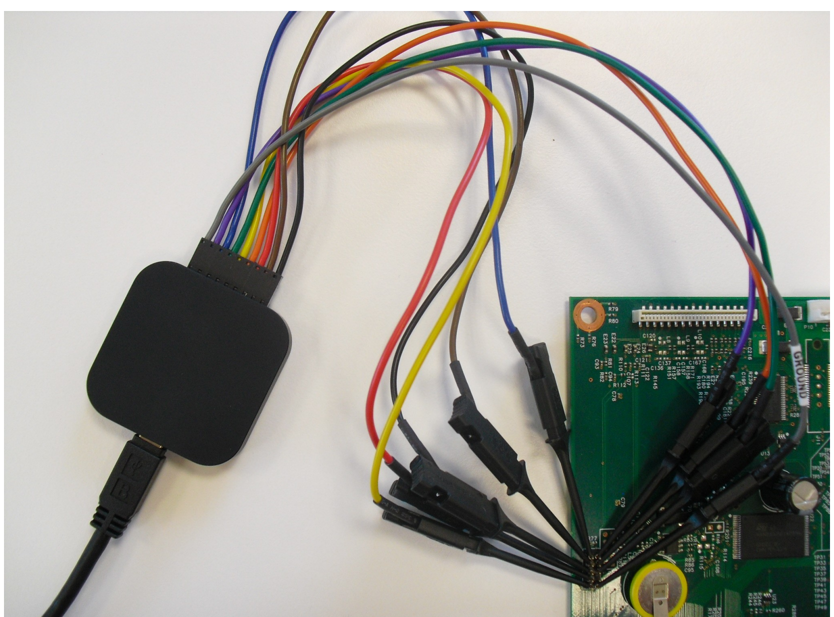
Serial Interfaces

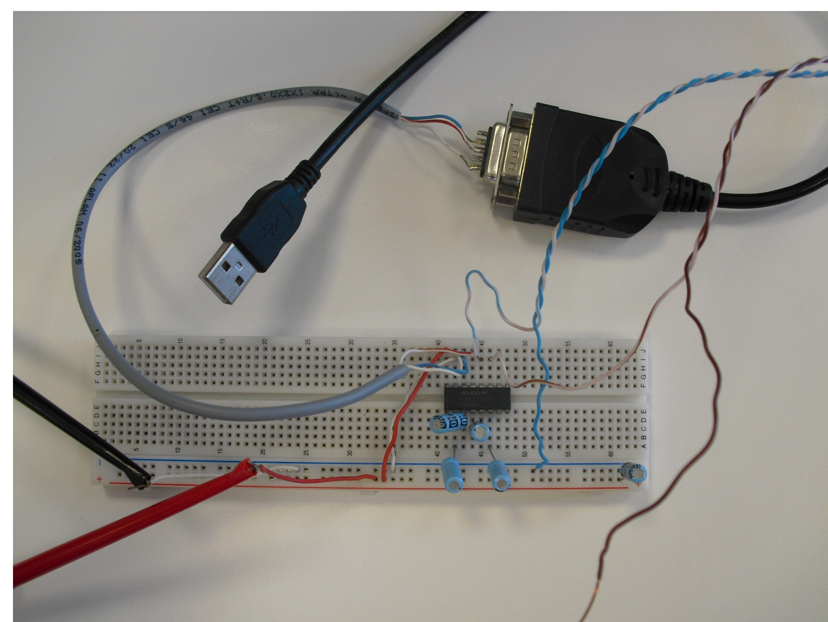
The blind (though usually fast and efficient) approach for the pin-out discovery consists of the following steps:

- connect a logical analyzer to every pin exposed by the interface
- start intercepting TTL levels
- reboot the target device
- wait for data coming out from any of the monitored pins (TX candidate)
- estimate the serial protocol parameters in terms of baud rate, data bits, stop bits, parity, bit order (MSB/LSB) and the interface logic (standard/inverted)
- probe remaining pins in order to find the RX

Serial Interfaces







JTAG

The JTAG (Joint Test Action Group) interface is not fully standardized as the number and position of pins differ across vendors/devices, the features implemented and exposed via the JTAG interface are also dependent on the specific board/chip manufacturer.

Boundary scan is an important helper when testing connections between different ICs on a certain JTAG chain but not interesting for further debugging.

“In-circuit” debugging, where implemented, allows operations such as CPU single stepping, breakpointing and full memory R/W access.

JTAG Scan

The relevant pins used by the TAP controller are the following:

- TDI (Test Data In) / TDO (Test Data Out)
- TCK (Test Clock)
- TMS (Test Mode Select)
- TRST (Test Reset) optional / SRST (System Reset)

Vcc, GND need to be found before starting the actual scan, using a probe resistor (300-500 Ohms) we try to pull-down/pull-up all the exposed pins.

This electrical probing also helps in finding high-impedance pins (input candidates).

JTAG

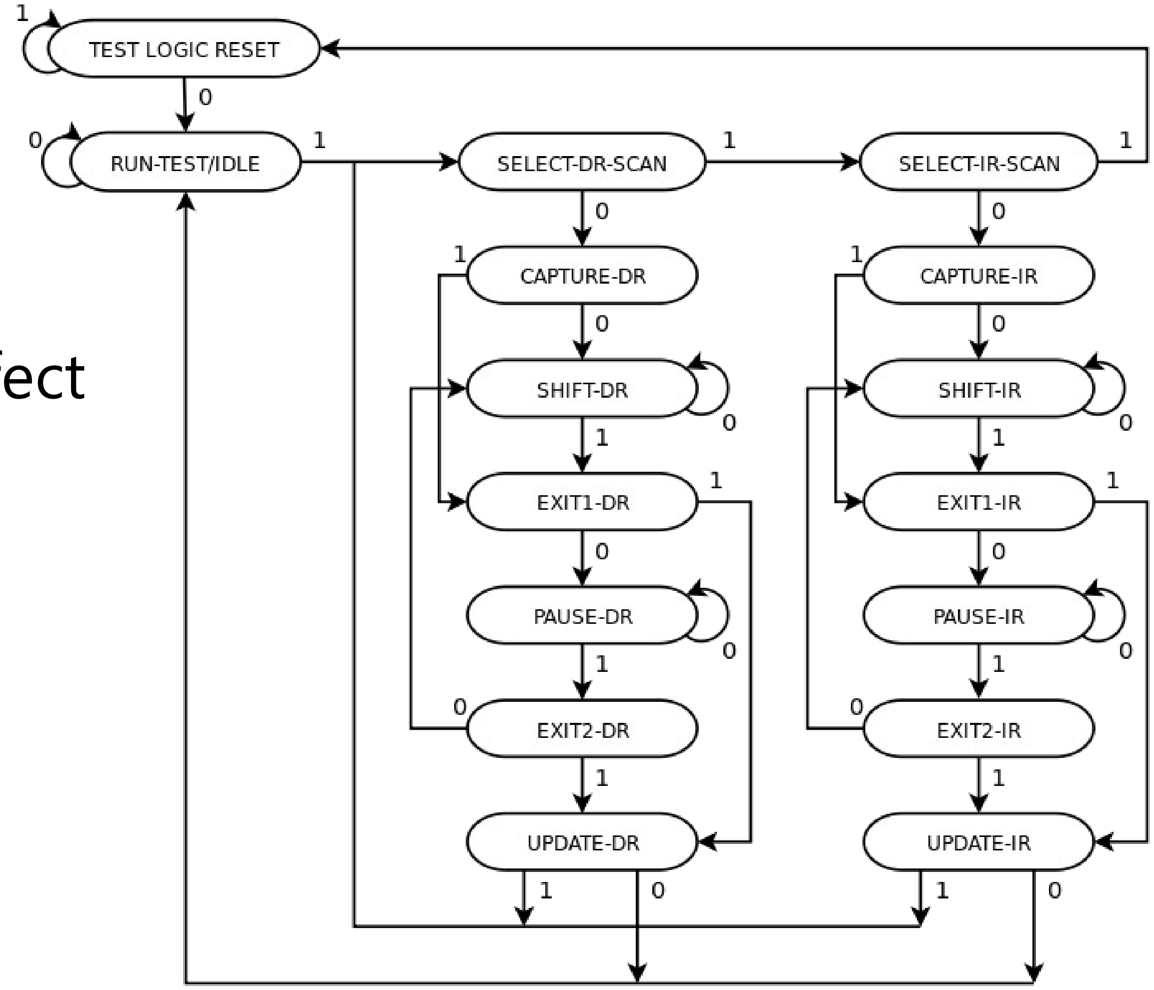
Scanning device features:

- a large number of GPIOs
- I/O speed is not relevant

Microcontrollers are the perfect tool for the job.

Scanning strategies:

- BYPASS
- IDCODE
- SHIFT IR / SHIFT DR



SPI, I^2C Devices

Often firmwares are stored on dedicated flash ICs which expose SPI or I^2C interfaces.

Vendors can implement restrictions (example: Intel descriptor mode) to protect certain memory areas from r/w access from the OS.

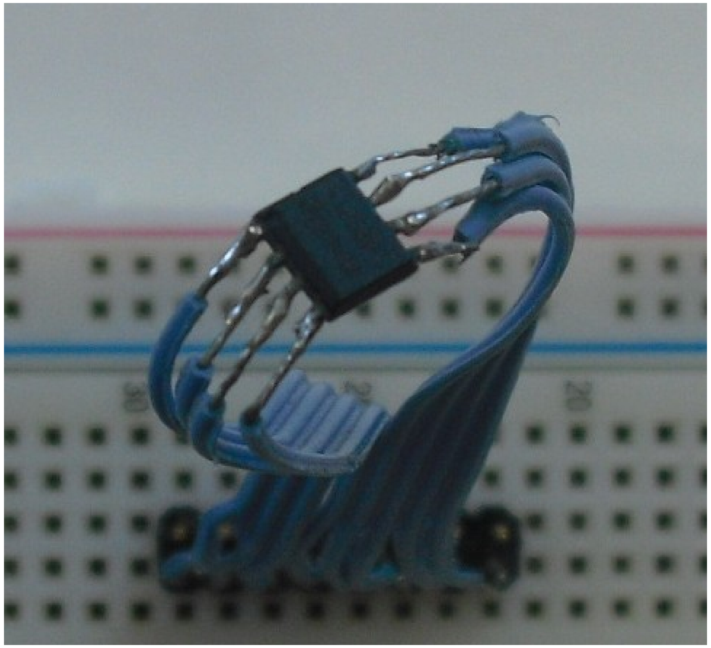
Physical memory access against the bare memory chip is one way of bypassing such protections.

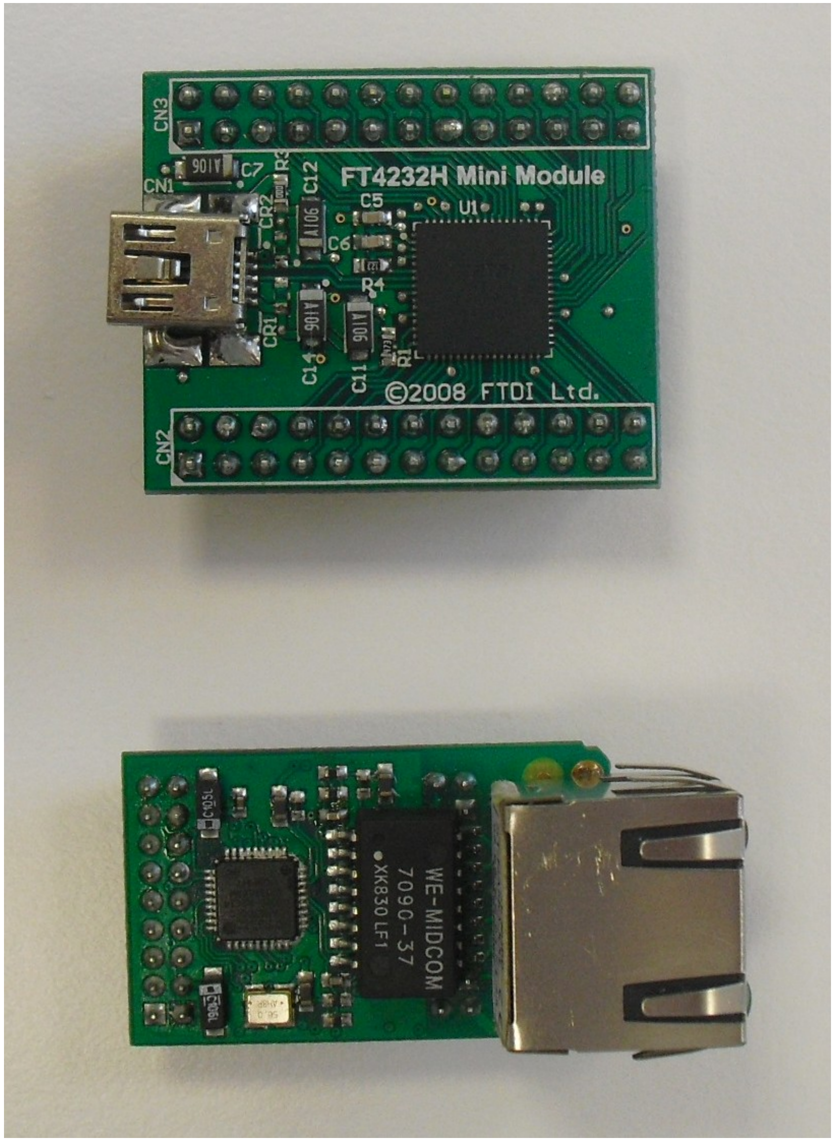
INVERSE PATH

SPI, I^{2}C Devices

Direct access options:

- custom programmed microcontroller
- flashrom + USB-serial/FIFO (FT4232H Mini)
- SPI/I2C converters (XTNano)





Checksum Algorithm

The reverse engineering of checksum algorithms is one of the first challenges of modifying existing firmware images.

The large majority of embedded systems employs only checksums to secure the firmware re-flashing process against errors, without security protection (i.e. signature verification).

Checksum Algorithm

CRC-32 is the most common algorithm with its standard documented polynomial 0x04c11db7, however in assembly code you will find its reversed representation (0x0edb8832).

```
0x04c11db7 == 0b01001100000100011101101110111
0x0edb8832 == 0b11101101101110001000001100010
```

Finding the polynomial is the first essential step in identifying the algorithm and its flavour, the other parameters to be identified generally follow the Rocksoft™ Model.

```
Width, Poly, Init, RefIn, RefOut, XorOut, Check
```

INVERSE PATH

Rocksoft™ Model CRC Algorithm

	Width	Poly	Init	RefIn,RefOut	XorOut	Check
CRC-16	15	0x8005	0x0000	true,true	0x0000	0xbb3d
CRC-32POSIX	32	0x4c11db7	0x00000000	false,false	0xffffffff	0x37aa6011
CRC-32	32	0x4c11db7	0xffffffff	true,true	0xffffffff	0xcbf43926
JamCRC	32	0x4c11db7	0xffffffff	true,true	0x00000000	0x340bc6d9

```
Width:  width of the algorithm
Poly:   generator polynomial
Init:   initialization vector
RefIn:  true  – input bytes bit 7 is most  significant bit (MSB)
        false – input bytes but 7 is least significant bit (LSB)
RefOut: true  – final value is sent to XorOut stage reflected
        false – final value is sent to XorOut stage directly
XorOut: value XORed to the final register value (after RefOut)
Check:  checksum value obtained using ASCII “123456789” as input
```

CRC Algorithm – Table generation

```
def generate_crc32_table
  table = []

  256.times do |i|
    crc = i;

    8.times do
      crc = (crc >> 1) ^ (reversed_poly * (crc & 1))
    end

    table << crc
  end

  table
end
```

CRC algorithm

```
def crc(input, table = false)
  crc = initial_vector

  if table
    crc32_table = generate_crc32_table()

    input.each_byte do |b|
      i = (crc ^ b) & 0xff
      crc = (crc >> 8) ^ crc32_table[i]
    end
  else
    input.each_byte do |b|
      crc ^= b
      8.times { crc = (crc >> 1) ^ (reversed_poly * (crc & 1)) }
    end
  end

  crc ^ xor_out
end
```

CRC 1003.2 draft 11 – Table generation

```
def generate_crc32_draft11_table
  table = []

  256.times do |i|
    crc = i;

    8.times do
      crc = (crc >> 1) ^ (0x0edb8832 * (crc & 1))
    end

    table << ((i == 0) ? 0x7fffffff : crc)
  end

  table
end
```

INVERSE PATH

CRC 1003.2 draft 11 – algorithm (w/ table)

```
def crc(input)
  table = generate_crc32_draft11_table()
  crc = 0x00000000 # initial_vector
  a = 0

  input.each_byte do |b|
    i = (crc >> 24) ^ b

    if i == 0                # intermediate zero is replaced
      i = a                  # with next value in sequence
      a = (a + 1) % 256
    end

    crc = ((crc << 8) ^ table[i]) & 0xffffffff
  end

  crc
end
```

CRC 1003.2 draft 11 – algorithm (w/o table)

```
def crc(input)
  crc = 0x00000000 # initial_vector
  a = 0

  input.each_byte do |b|
    i = (crc >> 24) ^ b

    if i == 0               # intermediate zero is replaced
      i = a                 # with next value in sequence
      a = (a + 1) % 256
    end

    8.times { i = (i >> 1) ^ (0x0edb8832 * (i & 1)) }
    crc = ((s << 8) ^ i) & 0xffffffff
  end

  crc
end
```

Checksum Algorithm Flavours

Non standard CRC algorithms not only can have different Rocksoft™ parameters but might not fit within the parametrization at all, 1003.2 draft 11 algorithm being one example.

The following CRC-32 flavours, for instance, all differ from one another:

Name	**Check**	**Rocksoft™ model**
1003.2 draft 9	**0x828bc708**	**N**
1003.2 draft 11	**0xfc9e4dc1**	**N**
1003.2 draft 12	**0xac65386c**	**N**
1003.2-1992 standard POSIX	**0x377a6011**	**Y**
CRC-32 (PKZIP, Ethernet)	**0xcbf43926**	**Y**

Runtime Kernel Patching

Real Time OSes often employ custom drivers/code to access internal hardware or implement protocol stacks often of interest for attack purposes.

As an example, once access is gained on the target system, it might be necessary to reverse engineer its communication to an internal security module which performs cryptographic keys exchange.

Even without kernel source it can be possible to hijack runtime kernel functions/system calls with debugging wrappers and eventually interception code.

Runtime Kernel Patching

Most embedded systems allow **/dev/mem** O_RDWR access.

```
unsigned long ptr = FUNCTION_POINTER;
unsigned long new_ptr = NEW_FUNCTION_POINTER;
int fd;

fd = open("/dev/mem", O_RDWR, 0);

if (lseek(fd, ptr, 0) == offset) {
    write(fd, (void *) &new_ptr, sizeof(new_ptr));
}
```

Kernel memory can also be inspected/modified with kernel modules with or without target OS development toolkit (available in most cases).

Runtime Kernel Patching

Example of function hijack for argument debugging.

```
/* pointer to wrapping function */
int wrapper_ptr = (int) func_wrapper

/* MIPS J - Jump operation:
   PC = nPC; nPC = (PC & 0xf0000000) | (26_bit_target_addr << 2); */

int jmp = ((2 << 26)|((wrapper_ptr - (func_ptr & 0xf0000000)) / 4));
int nop = 0x00000000;

/* function placeholder */
char func_holder = char[func_size];

/* prototype for function access via placeholder */
void * (*held_func)(void *a0, void *a1) = (void *) func_holder;
```

Runtime Kernel Patching

```
/* without devkit it is possible to use manually identified ptrs */
void * (*memcpy)(void *, void *, size_t) = (void *) MEMCPY_PTR;

/* copy existing function in a placeholder */
memcpy((void *) func_holder, (void *) function_ptr, func_size);

/* replace function with jmp to debugging function */
memcpy((void *) func_pointer, &jmp, sizeof(jmp));
memcpy((void *) (func_ptr + 4), &nop, sizeof(nop));

int func_wrapper(void *a0, void *a1)
{
    /* custom code inspecting or modifying a0, a1 */
    /* exact number of arguments is not necessary */

    held_func(a0, a1);
{
```

Runtime Kernel Patching

Depending on the architecture the exact number of arguments for the function to hijack does not need to be known and can be found by trial and error.

Symbol offsets can be decoded from the extracted kernel image by decoding the debugging symbols (if present), or runtime by identifying the system call table. Pointers can be used with function prototypes.

The system call table can be recognized as a list of offsets with values close to each other. The list ordering reflects the syscall number which is often compliant to the OS family (Linux: **`syscall_32.tbl`**, BSD: **`syscalls.master`**)

Practical Example: Apple SMC

The System Management Controller (SMC) is an internal embedded subsystem implemented on Intel based Apple laptops.

The usage of such Embedded Controllers (EC) is not restricted to Apple and can be found on several Intel based products.

Such ECs can be used as SMC, KBC (Keyboard Controller) or both (Keyboard and System Controller).

Apple allows firmware upgrade for their SMC, therefore for educational purposes we detail the process of investigating if and how arbitrary firmwares can be flashed.

Apple SMC

An SMC is generally used for:

- Thermal Management
- Power monitoring
- Battery Management
- SPI Flash Bridge (BIOS storage)
- ACPI Host Interface
- Signal Buffering & Level Shifting
- Custom programmable functionality

On Apple systems it reportedly manages the power button activity, display lid open/close activity, Sudden Motion Sensor, ambient light sensing, keyboard light, indicator lights.

Apple SMC

The Apple SMC is queried by the OS (several tools are available to manually reproduce such queries) to retrieve or set “SMC keys”.

Some examples:

```
#ALA0: ALS analog lux info
#ALT0: ALS ambient light sensor temperature for sensor 1
#BSIn: Battery Status (present, charging, etc.)
#F0Ac: Fan 0 RPM
#FPhz: Programmable fan phase offset
#NATi: Ninja Action Timer (!!!)
#IC0C: CPU 0 core current
#MOCF: Motion sensor configuration register
```

SMC Update file (MacBook Air)

The file is usually named MacBookAirSMCUpdate<version>.dmg, the DMG format (Apple Disk Image) is well known and can be easily extracted.

A gzip compressed cpio archive named **`Payload`** can be found within the package and can be extracted with the following command:

```
$ zcat Payload | cpio -i
```

The interesting files are:

```
./Utilities/MacBook AIR SMC Firmware Update.app/Contents/Resources/SmcFlasher.efi
./Utilities/MacBook AIR SMC Firmware Update.app/Contents/Resources/m82.smc
./Utilities/MacBook AIR SMC Firmware Update.app/Contents/Resources/m96.smc
```

SMC Update file (MacBook Air)

The file **`SmcFlasher.efi`** is a universal EFI binary (i386 + x86_64) which is **`bless(8)`**'ed for execution within the Apple EFI environment during the boot sequence.

The files **`m82.smc`** and **`m96.smc`** (for different specific part numbers) contain the actual firmware image which can be input as argument to **`SmcFlasher.efi`**.

The smc firmware files are checked for integrity by the flasher application when the update is applied.

Let us analyze the format to understand the integrity checksum.

```
# Version: 1.23f20
H:20:BF00000000000000000000000000000000000000:BF
H:20:0200000000000000000000000000000000000000:02

...

H:20:2D00000000000000000000000000000000000000:2D
H:20:9B00000000000000000000000000000000000000:9B
S:20:1B00000000000000000000000000000000000000:1B
D:00000000:64:<64 bytes of data>:2E
+         :64:<64 bytes of data>:90
+         :64:<64 bytes of data>:A0

...

D:00001000:64:<64 bytes of data>:C2
+         :64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0

...

D:00027800:64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0
```

```
# Version: 1.23f20
H:20:BF00000000000000000000000000000000000000:BF # H => hash data
H:20:0200000000000000000000000000000000000000:02

…

H:20:2D00000000000000000000000000000000000000:2D
H:20:9B00000000000000000000000000000000000000:9B
S:20:1B00000000000000000000000000000000000000:1B # S => security data
D:00000000:64:<64 bytes of data>:2E              # D => data block
+         :64:<64 bytes of data>:90
+         :64:<64 bytes of data>:A0

…

D:00001000:64:<64 bytes of data>:C2
+         :64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0

…

D:00027800:64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0
```

```
# Version: 1.23f20
H:20:BF00000000000000000000000000000000000000:BF # H => hash data
H:20:0200000000000000000000000000000000000000:02 # 20 => length

...

H:20:2D00000000000000000000000000000000000000:2D
H:20:9B00000000000000000000000000000000000000:9B
S:20:1B00000000000000000000000000000000000000:1B # S => security data
D:00000000:64:<64 bytes of data>:2E              # D => data block
+         :64:<64 bytes of data>:90
+         :64:<64 bytes of data>:A0              # 64 => length

...

D:00001000:64:<64 bytes of data>:C2
+         :64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0

...

D:00027800:64:<64 bytes of data>:C0
+         :64:<64 bytes of data>:C0
```

```
# Version: 1.23f20
H:20:BF000000000000000000000000000000000000000:BF # H => hash data
H:20:02000000000000000000000000000000000000000:02 # 20 => length

...

H:20:2D000000000000000000000000000000000000000:2D
H:20:9B000000000000000000000000000000000000000:9B
S:20:1B000000000000000000000000000000000000000:1B # S => security data
D:00000000:64:<64 bytes of data>:2E              # D => data block
+         :64:<64 bytes of data>:90
+         :64:<64 bytes of data>:A0              # 64 => length

...

D:00001000:64:<64 bytes of data>:C2              # 0x00001000 => memory
+         :64:<64 bytes of data>:C0                             address
+         :64:<64 bytes of data>:C0

...

D:00027800:64:<64 bytes of data>:C0              # 0x00027800 => memory
+         :64:<64 bytes of data>:C0                             address
```

SMC Update file checksum

Closely analyzing the hash data and data block format, which resembles the Intel HEX/SREC file formats, reveals a simple checksum algorithm.

```
# Version: 1.23f20
H:20:BF00000000000000000000000000000000000000:BF # H => hash data
H:20:0200000000000000000000000000000000000000:02 # 20 => length

...

D:00008000:64:5A0089860000000...................................000:69
+         :64:..F8065A009C0C000000000000000000005A0089C6000000000...:A9

. represent an omitted series of 0

0x5a + 0x89 + 0x86 == 0x169
0xf8 + 0x06 + 0x5a + 0x9c + 0x0c + 0x5a + 0x89 + 0xc6 == 0x3a9
```

SMC Update file checksum

Each 64 bytes data entry is appended a checksum that consists of the least significant byte of the sum of the values.

The hash data sections (H) consists of the sum of the checksums for each 64 bytes of a data block (D).

The security data section (S) consists of the sum of the checksums for each hash data section (H).

It becomes trivial to modify the SMC firmware image.

SMC Architecture

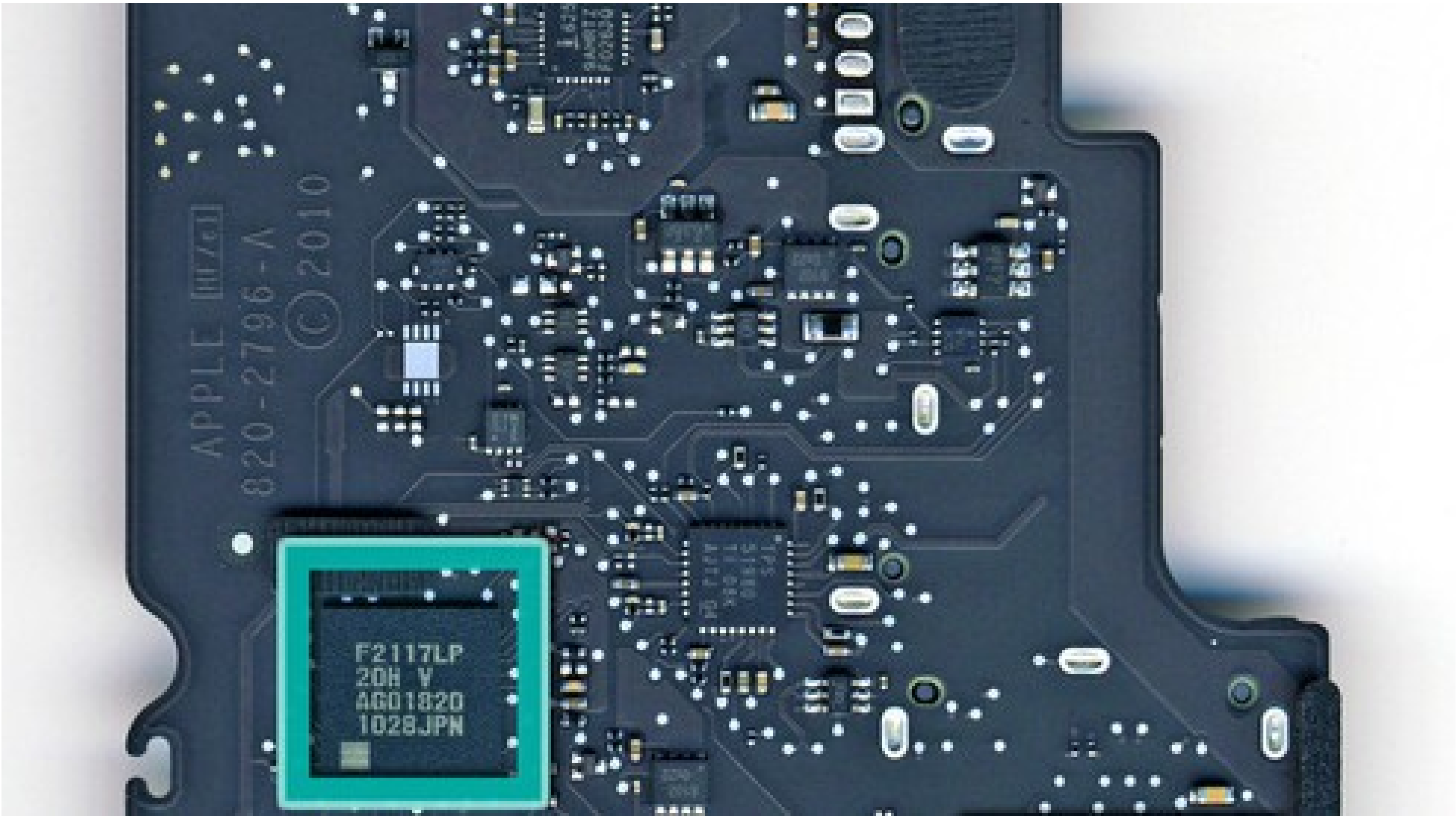
Some data about the architecture of the SMC microcontroller can be inferred from the first relevant data block (memory address **`0x8000)`**.

```
D:00008000:64:5A0089860000000...............................000:69
+         :64:..F8065A009C0C0000000000000000000005A0089C6000000000...:A9
```

The first address in the user code area seems to be **`0x8000`**. The NOP instruction is probably a certain number of **`0x00`**.

An image of the actual SMC microcontroller of course can aid tracking down the exact architecture.

MacBook Air (mid-2011) Motherboard



Source: www.ifixit.com MacBook Air Teardown (CC BY-NC-SA)

2117 => Renesas H8S/2117 family, 16-bit Single-Chip microcomputer

Renesas H8S/2117

CISC microcomputer, 16-bit architecture

160 Kbytes of ROM, 8 Kbytes of RAM

I/O features:

- I^2C bus interface
- A/D converter
- Serial interface
- Keyboard buffer control (PS2) and matrix scan (unused by Apple)
- LPC interface
- lots of generic I/O ports

Renesas H8S/2117

It is a widely used Embedded Controller present in Apple laptops as well as other Intel-based hardware.

```
D:00008000:64:5A0089860000000..............................000:69
+          :64:..F8065A009C0C0000000000000000000005A0089C6000000000...:A9
```

According to its instruction set the absolute JMP instruction code is the following:

```
1st byte    2nd byte    3rd byte    4th byte
5       A   0       0     absolute address
```

Therefore the first 4 bytes represent an absolute JMP to offset **0x8986**. The GNU Development tools for Renesas H8/300 series can be used.

INVERSE PATH

Disassembling the firmware image

The quick & dirty way to do it:

```
$ grep -o -E "[A-Z0-9]{64,}" m96.smc | xxd -r -p > m96.bin
$ h8300-hitachi-coff-objdump -start-address=0x1000 -m h8300 \
    -b binary -D m96.bin
```

```
00001000 <.data+0x1000>:
    1000:       5a 00 89 86     jmp     @0x8986:24
        ...
    1060:       f8 06           mov.b   #0x6,r0l
    1062:       5a 00 9d 6c     jmp     @0x9d6c:24
        ...
    106e:       00 00           nop
    1070:       5a 00 89 ca     jmp     @0x89ca:24
        ...
    1080:       f8 08           mov.b   #0x8,r0l
    1082:       5a 00 9d 6c     jmp     @0x9d6c:24
        ...
```

Disassembly: .data offset resolution

```
f7cc:     0c 88                 mov.b   r0l,r0l
f7ce:     47 2c                 beq     .+44 (0xf7fc)
f7d0:     a8 01                 cmp.b   #0x1,r0l
f7d2:     47 20                 beq     .+32 (0xf7f4)
f7d4:     a8 02                 cmp.b   #0x2,r0l
f7d6:     47 14                 beq     .+20 (0xf7ec)
f7d8:     a8 03                 cmp.b   #0x3,r0l
f7da:     46 08                 bne     .+8 (0xf7e4)
f7dc:     7a 00 00 01 ea fe     mov.l   #0x1eafe,er0
f7e2:     54 70                 rts
f7e4:     7a 00 00 01 ea f0     mov.l   #0x1eaf0,er0
f7ea:     54 70                 rts
f7ec:     7a 00 00 01 eb 0f     mov.l   #0x1eb0f,er0
f7f2:     54 70                 rts
f7f4:     7a 00 00 01 ea f8     mov.l   #0x1eaf8,er0
f7fa:     54 70                 rts
f7fc:     7a 00 00 01 eb 1a     mov.l   #0x1eb1a,er0
f802:     54 70                 rts
```

Disassembly: .data offset resolution

```
f7cc:   0c 88               mov.b   r0l,r0l
f7ce:   47 2c               beq     .+44 (0xf7fc)
f7d0:   a8 01               cmp.b   #0x1,r0l
f7d2:   47 20               beq     .+32 (0xf7f4)
f7d4:   a8 02               cmp.b   #0x2,r0l
f7d6:   47 14               beq     .+20 (0xf7ec)
f7d8:   a8 03               cmp.b   #0x3,r0l
f7da:   46 08               bne     .+8 (0xf7e4)
f7dc:   7a 00 00 01 ea fe   mov.l   "LmsBrightNoScale",er0
f7e2:   54 70               rts
f7e4:   7a 00 00 01 ea f0   mov.l   "Unknown",er0
f7ea:   54 70               rts
f7ec:   7a 00 00 01 eb 0f   mov.l   "LmsBreathe",er0
f7f2:   54 70               rts
f7f4:   7a 00 00 01 ea f8   mov.l   "LmsOn",er0
f7fa:   54 70               rts
f7fc:   7a 00 00 01 eb 1a   mov.l   "LmsOff",er0
f802:   54 70               rts
```

Disassembly: .data offset resolution

```
0x1eafe => 0x17afe  LmsBrightNoScale
0x1eaf0 => 0x17af0  Unknown
0x1eb0f => 0x17b0f  LmsBreathe
0x1eaf8 => 0x17af8  LmsOn
0x1eb1a => 0x17b1a  LmsOff
```

```
00017AE0  00 00 00 00 00 00 75 69 31 36 00 00 00 21 31 00 ......ui16...!1.
00017AF0  55 6E 6B 6E 6F 77 6E 00 4C 6D 73 4F 6E 00 4C 6D Unknown.LmsOn.Lm
00017B00  73 42 72 69 67 68 74 4E 6F 53 63 61 6C 65 00 4C sBrightNoScale.L
00017B10  6D 73 42 72 65 61 74 68 65 00 4C 6D 73 4F 66 66 msBreathe.LmsOff
00017B20  00 00 00 00 01 A2 2E F4 02 E6 02 5D 01 8B 00 D7 ...........]....
```

Disassembly: SMC keys constants

```
00016910  0E 9C 0D 30 0E 9C 08 78 00 00 01 10 00 00 00 04 ...0...x........
00016920  23 4B 45 59 80 04 00 00 75 69 33 32 00 01 D9 18 #KEY....ui32....
00016930  24 41 64 72 88 04 00 00 75 69 33 32 00 FF DA A8 $Adr....ui32....
00016940  24 4E 75 6D D0 01 00 00 75 69 38 20 00 01 4F 8A $Num....ui8 ..O.
00016950  2B 4C 4B 53 90 01 00 00 66 6C 61 67 00 01 BF 14 +LKS....flag....
00016960  41 43 43 4C 51 01 00 00 75 69 38 20 00 00 D8 B8 ACCLQ...ui8 ....
00016970  41 43 45 4E D0 01 00 00 75 69 38 20 00 00 D8 C2 ACEN....ui8 ....
00016980  41 43 46 50 80 01 00 00 66 6C 61 67 00 FF DF D2 ACFP....flag....
00016990  41 43 49 44 90 08 00 00 63 68 38 2A 00 00 D5 A8 ACID....ch8*....
000169A0  41 43 49 4E 80 01 00 00 66 6C 61 67 00 FF D2 8F ACIN....flag....
000169B0  41 4C 21 20 C0 02 00 00 75 69 31 36 00 FF D1 64 AL! ....ui16...d
000169C0  41 4C 41 30 C8 06 00 00 7B 61 6C 61 00 FF E0 2A ALA0....{ala...*
000169D0  41 4C 41 31 C8 06 00 00 7B 61 6C 61 00 FF E0 30 ALA1....{ala...0
000169E0  41 4C 41 32 C8 06 00 00 7B 61 6C 61 00 FF E0 36 ALA2....{ala...6
000169F0  41 4C 41 33 C8 06 00 00 7B 61 6C 61 00 FF E0 3C ALA3....{ala...<
```

0x1d918 => 0x16918   **Number of SMC keys (0x110 => 272)**

Disassembly: I^2C operations

```
a9fc: f9 05                    mov.b  #0x5,r1l
a9fe: 6a 89 fe 8a              mov.b  r1l,@0xfe8a:16
aa02: 6a 18 fe 88 72 70        bclr   #0x7,@0xfe88:16
aa08: 6a 2a 00 01 d8 4a        mov.b  @0x1d84a:32,r2l
aa0e: 6a 8a fe 8f              mov.b  r2l,@0xfe8f:16
aa12: 6a 2a 00 01 d8 4b        mov.b  @0x1d84b:32,r2l
aa18: 6a 8a fe 8e              mov.b  r2l,@0xfe8e:16
aa1c: 6a 18 fe 88 70 70        bset   #0x7,@0xfe88:16
aa22: 6a 18 fe 89 72 00        bclr   #0x0,@0xfe89:16
aa28: 7f c3 70 70              bset   #0x7,@0xc3:8
aa2c: 0f b0                    mov.l  er3,er0
aa2e: 10 33                    shll.l er3
aa30: 0a 83                    add.l  er0,er3
aa32: 78 30 6a 29 00 ff d7 08 mov.b  @(0xffd708:32,r3),r1l
aa3a: 6a 89 fe 8f              mov.b  r1l,@0xfe8f:16
aa3e: 6a 18 fe 88 70 30        bset   #0x3,@0xfe88:16
aa44: 6a 18 fe 88 72 60        bclr   #0x6,@0xfe88:16
aa4a: 6a 18 fe 88 72 50        bclr   #0x5,@0xfe88:16
aa50: 6a 18 fe 88 72 40        bclr   #0x4,@0xfe88:16
aa56: 6a 18 fe 8c 70 70        bset   #0x7,@0xfe8c:16
aa5c: f8 4c                    mov.b  #0x4c,r0l
aa5e: 6a 88 fe 8c              mov.b  r0l,@0xfe8c:16
aa62: 5a 01 1b 54              jmp    @0x11b54:24
```

Disassembly: I^2C operations

```
a9fc: f9 05                   mov.b  #0x5,r1l
a9fe: 6a 89 fe 8a             mov.b  r1l,@i2c_bus_ctrl_init_reg_2 # clear internal latch
aa02: 6a 18 fe 88 72 70       bclr   #0x7,@i2c_bus_ctrl_reg_2     # clear bus interface
aa08: 6a 2a 00 01 d8 4a       mov.b  @0x1d84a:32,r2l              #  .data 0x1684a => 0x10
aa0e: 6a 8a fe 8f             mov.b  r2l,@slave_addr_reg_2        #     slave addr => 0x8
aa12: 6a 2a 00 01 d8 4b       mov.b  @0x1d84b:32,r2l              #  .data 0x1684b => 0x12
aa18: 6a 8a fe 8e             mov.b  r2l,@2nd_slave_addr_reg_2    # 2nd slave addr => 0x9
aa1c: 6a 18 fe 88 70 70       bset   #0x7,@i2c_bus_ctrl_reg_2     # set bus interface
aa22: 6a 18 fe 89 72 00       bclr   #0x0,@i2c_bus_status_reg_2   # clear ACKB
aa28: 7f c3 70 70             bset   #0x7,@0xc3:8
aa2c: 0f b0                   mov.l  er3,er0
aa2e: 10 33                   shll.l er3
aa30: 0a 83                   add.l  er0,er3
aa32: 78 30 6a 29 00 ff d7 08 mov.b  @(0xffd708:32,r3),r1l
aa3a: 6a 89 fe 8f             mov.b  r1l,@i2c_bus_mode_reg_2
aa3e: 6a 18 fe 88 70 30       bset   #0x3,@i2c_bus_ctrl_reg_2     # set ACKE
aa44: 6a 18 fe 88 72 60       bclr   #0x6,@i2c_bus_ctrl_reg_2     # clear interrupts
aa4a: 6a 18 fe 88 72 50       bclr   #0x5,@i2c_bus_ctrl_reg_2     # slave receive mode
aa50: 6a 18 fe 88 72 40       bclr   #0x4,@i2c_bus_ctrl_reg_2     # slave receive mode
aa56: 6a 18 fe 8c 70 70       bset   #0x7,@i2c_bus_ext_ctrl_reg_2 # set STOPIM
aa5c: f8 4c                   mov.b  #0x4c,r0l
aa5e: 6a 88 fe 8c             mov.b  r0l,@i2c_bus_ext_ctrl_reg_2  #
aa62: 5a 01 1b 54             jmp    @0x11b54:24
```

Disassembly: Battery Status SMC key

```
00016D30  42 4E 75 6D 80 01 00 00 75 69 38 20 00 FF E1 0E BNum....ui8 ....
00016D40  42 52 53 43 80 02 00 00 75 69 31 36 00 FF D2 00 BRSC....ui16....
00016D50  42 53 41 43 C0 01 00 00 75 69 38 20 00 FF D2 6C BSAC....ui8 ...l
00016D60  42 53 44 43 80 01 00 00 75 69 38 20 00 FF D2 56 BSDC....ui8 ...V
00016D70  42 53 49 6E 80 01 00 00 75 69 38 20 00 FF D2 6A BSIn....ui8 ...j
```

```
1eb0: 6a 28 00 e1 0e          mov.b @0xffe10e:32,r0l # supported battery count
1eb6: 58 70 00 14             beq   .+20 (0x1ece)    # count == 1 ? --------+
1eba: 28 c1                   mov.b @0xc1:8,r0l                              |
1ebc: 77 18                   bld  #0x1,r0l                                  |
1ebe: 58 50 00 0c             bcs  .+12 (0x1ece)                             |
1ec2: 6a 38 00 ff d2 6a 72 10 bclr #0x1,@0xffd26a:32 # AC not present        |
1eca: 58 00 00 08             bra  .+8 (0x1ed6)                              |
1ece: 6a 38 00 ff d2 6a 70 10 bset #0x1,@0xffd26a:32 # AC present <---------+
```

Apple SMC

In conclusion the Apple SMC can be updated with arbitrary firmware as the checksum algorithm, update mechanism and eventually functionality can be fully reversed engineered.

Knowledge of the architecture makes it straightforward to modify the firmware at will.

DISCLAIMER: this is an educational example only, use the presented information at your own risk.