Black Box Debugging of Embedded Systems
Introduction: Alexandru Ariciu

- Background in hacking
- “Worked” as a hacker for my whole life
- Worked in corporate security before (Pentester)
- Currently an ICS Penetration Tester / Vulnerability Researcher for Applied Risk

- Linkedin: https://www.linkedin.com/in/alexandru-ariciu-856bb566
- Twitter: @1n598
Target Device

- Data acquisition and transmission device
- Used in distributed control systems
- Over 300 directly connected to the internet.
- More connected internally, mostly in level 0 and level 1 SCADA networks

Prerequisites

- ARM knowledge
- Reverse engineering
- Some hardware hacking (we will cover that part)
Goal

- Provide a means to develop additional advanced functionality to this device; such as execution of arbitrary code

By means of:
- Injecting the device with modified firmware, to enable debugging on the device, and step through custom code without bricking the device

Complication:
- There is no datasheet of the chip available, firmware is proprietary
ARM crash course

- ARM is a 32 bit RISC processor
- Processor is little endian
- Opcodes are 32bit, and align on a 4 byte boundary
- All code, data and peripherals share the same 32 bit memory address space
- After a (re)boot code starts running from 0x00000000
- ARM has 16 32-bit base registers and a CPSR (current program status register)
- Most opcodes can be modified by condition fields
- A 3 stage pipeline is used, so PC is 2 instructions (8 bytes) ahead of the currently executed instruction
- A branch flushes the pipeline
<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>add two 32 bit values</td>
</tr>
<tr>
<td>AND</td>
<td>logical bitwise AND of two 32 bit values</td>
</tr>
<tr>
<td>B</td>
<td>branch relative +/- 32MB</td>
</tr>
<tr>
<td>BL</td>
<td>relative branch with link</td>
</tr>
<tr>
<td>CMP</td>
<td>compare two 32 bit values</td>
</tr>
<tr>
<td>LDR(B)</td>
<td>load a single value from a virtual address in memory</td>
</tr>
<tr>
<td>LDMFD</td>
<td>store multiple 32 bit registers to memory</td>
</tr>
<tr>
<td>MOV</td>
<td>move a 32 bit value into a register</td>
</tr>
<tr>
<td>MRS</td>
<td>move to ARM register from a status register (cpsr or spsr)</td>
</tr>
<tr>
<td>MSR</td>
<td>move to a status register (cpsr or spsr) from an ARM register</td>
</tr>
<tr>
<td>ORR</td>
<td>logical bitwise OR of two 32 bit values</td>
</tr>
<tr>
<td>STMFD</td>
<td>store multiple 32 bit registers to memory</td>
</tr>
<tr>
<td>STR(B)</td>
<td>store register to a virtual address in memory</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract two 32 bit values</td>
</tr>
<tr>
<td>LSL</td>
<td>left shift a 32 bit register</td>
</tr>
<tr>
<td>Opcode [31:28]</td>
<td>Mnemonic extension</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>0000</td>
<td>EQ</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
</tr>
<tr>
<td>0010</td>
<td>CS/HS</td>
</tr>
<tr>
<td>0011</td>
<td>CC/LO</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
</tr>
</tbody>
</table>
ARM Registers

Register Use in the ARM Procedure Call Standard

- **Scratch Registers**: r0-r3, r12
  - r0-r3 used to pass parameters
  - r12 intra-procedure scratch
  - will be overwritten by subroutines

- **Preserved Registers**: r4-r11
  - stack before using
  - restore before returning

- **Stack Pointer**: not much use on the stack

- **Link Register**: set by BL or BLX on entry of routine
  - overwritten by further use of BL or BLX

- **Program Counter**
Pipelining

- Initially implemented a 3-stage pipeline organization. (upto ARM7)
  - Fetch
  - Decode
  - Execute
Outline

• Output console
• Initial Infection
• Initial firmware patch with basic debugger
• Advanced debugging capabilities implementation
The process we will follow

1. Evaluate ways to interact with the device (UART, webserver, …)
2. Evaluate ways to make the device run custom code (buffer overflow, firmware modification, …)
3. Find a suitable place to inject code, to create reliable behavior (e.g. when calling a certain function)
4. Reverse engineer how the device interacts with the interfaces, and find functions such as printf that we can reuse
5. Combine existing functionality with our custom code, to provide (printf) feedback to our code execution

• From here on out we can reverse engineer more functions to perform more advanced actions such as;
  • getc, so we can interact with our running code
  • read/write memory to evaluate and modify other system behavior
Process: Step 1

Evaluate ways to interact with the device (UART, webserver, …)
Interacting with the device: Output Console

- UART, Serial, JTAG, others
Process: Step 2

Evaluate ways to make the device run custom code
Process: Step 3

Find a suitable place to inject code, to create reliable behavior
Finding where to put our code: Initial basic debugger

- Load the code into IDA
- Select ARM Little Endian as the processor type
- Select 0x20 as file offset
- Click OK
Initial basic debugger

- The idea now is to find a function that is printing stuff in the console
- We will patch that function to print a string whenever we call that console command
- We used sys mem (function for displaying memory in the console)
Initial basic debugger

- We will modify this so that when we call sys mem in the console "Hello !" will appear

```
35  E1242>>
36  E1242>>sys mem
37  Free/Total memory => 1431740/1620888
```
Initial basic debugger

- Let's find the function in IDA
Initial basic debugger

- This is the assembly for the function “sys mem”
- We will patch this to jump to our ”initial patch”
Reverse engineer how the device interacts with the interfaces, and find functions such as “printf” that we can reuse.
Initial basic debugger

• It seems our sys mem function already calls a **printf** function we can reuse
Process: Step 5

Combine existing functionality with our custom code, to provide (printf) feedback to our code execution
Initial basic debugger

• The initial patch will contain three simple elements

  1. Load in R0 a string of our choosing
  2. Call to printf
  3. A return to where we left off (so that the device continues working)

• Let’s dive into some more details
Initial basic debugger

- We will modify the BL sub_0x644c0 to jump to our code
- Our code will be hosted in a place in the binary (our choosing)
Initial basic debugger

- BL is relative
- That is, it will jump not to definitive addresses but to addresses that are calculated relative to the PC position at the moment of the jump
- BL is more like ADD PC, PC, #jump
- It also stores the current address in the link register so that the processor knows where to come back
Initial basic debugger

- The BL instruction has the ARM opcode 0xEB
- The remainder of the three bytes are the address where to make the jump
- We need to calculate this
Initial basic debugger

- If we are at position X in the binary (0x2714 for us)
- And we want to jump at Y
- Then the address where to make the jump is like this \((Y - X)/4 - 0x2\)
- This is because we can only jump 4-bytes at a time (1 instruction = 4 bytes), and the PC is always 2 instructions ahead due to instruction pipelining
- For BL to 0x6440 this would be 
  \((0x644C0 - 0x2714)/4 - 0x2 = 0x18769\)
Initial basic debugger

- Find space to host the shellcode (for the initial infection)
- We will terminate one of the strings early so that we don’t modify the length of the binary
- A good place is where the Free/Memory string is
Initial basic debugger

- We will modify the string at address 0x10F6A2 (with the initial offset it will be 0x10F6A2 +0x20)
- Add 0D 0A 00 00 00 00 (We will end the string with a null terminator)
Initial basic debugger

- We will store the shellcode starting with 0x10F6A8
- So our BL should be like BL *(0x10F6A8)
- Based on the calculation it is (0x10F6A8-0x2714)/4 – 0x2 = 0x433E3
- So the opcode should be 0xEB0433E3
- Let’s patch and watch in IDA
Initial basic debugger

- When opening in IDA, the assembly should point to our location if we did the right calculations
- It seems it’s working
Initial basic debugger

• Well the code is not really what we expected
• But we didn’t patch it to contain our code
• Yet
Initial basic debugger

- This location must be patched to do the following:
  - Patch a string in memory
  - Load a string in the R0 register
  - Jump to printf
  - Return (load link register in PC)
Initial basic debugger

- Loading a string can be done using
  - SUB R0, PC, 0x22
  - This basically will load in R0 a string located 22 bytes before the program counter.
  - So we need to calculate \((0x10F6A8 + 0x8) - 0x22 = 0x10F68E\)
  - The string is Hello!
  - In the hexeditor, starting at address 0x0010F6AB modify the code to contain 0D 0A 00 48 65 6C 6C 6F 21 0D 00 00 00
  - We patched the „Hello!“ in the memory without breaking stuff around
Initial basic debugger

- It should look like this in the Hex Editor

<table>
<thead>
<tr>
<th>Hex Address</th>
<th>Hex Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10F680</td>
<td>6F640000</td>
<td>ad</td>
</tr>
<tr>
<td>10F684</td>
<td>3D3D3D3D</td>
<td>====</td>
</tr>
<tr>
<td>10F688</td>
<td>3D3D3D3D</td>
<td>====</td>
</tr>
<tr>
<td>10F68C</td>
<td>3D3D3D3D</td>
<td>====</td>
</tr>
<tr>
<td>10F690</td>
<td>3D3D3D3D</td>
<td>====</td>
</tr>
<tr>
<td>10F694</td>
<td>73797320</td>
<td>sys</td>
</tr>
<tr>
<td>10F698</td>
<td>636666D6</td>
<td>comm</td>
</tr>
<tr>
<td>10F69C</td>
<td>6E6E6420</td>
<td>and</td>
</tr>
<tr>
<td>10F6A0</td>
<td>75756167</td>
<td>usag</td>
</tr>
<tr>
<td>10F6A4</td>
<td>653D3D3D</td>
<td>e==</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex Address</th>
<th>Hex Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10F6AC</td>
<td>04004865</td>
<td>He</td>
</tr>
<tr>
<td>10F6B0</td>
<td>6C6C6F21</td>
<td>llo!</td>
</tr>
<tr>
<td>10F6B4</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>10F6BC</td>
<td>2F54F74</td>
<td>/Total</td>
</tr>
<tr>
<td>10F6C0</td>
<td>6D6D0D0D</td>
<td>al</td>
</tr>
<tr>
<td>10F6C4</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>10F6C8</td>
<td>79203D3E</td>
<td>y =&gt;</td>
</tr>
<tr>
<td>10F6CC</td>
<td>202562F</td>
<td>%d/</td>
</tr>
<tr>
<td>10F6D0</td>
<td>256400DA</td>
<td>%d</td>
</tr>
<tr>
<td>10F6D4</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>
Initial basic debugger

- Load the String in R0
- 0x0010F6C8 : 22 00 4F E2 # SUB R0, PC, 0x22
- Let’s load again in IDA
Initial basic debugger

- We need to jump to `printf`
- The BL is relative to 0x10F6A8
- The opcode is 0x0010F6CC : 83 53 FD EB # BL printf(0x644C0)
Initial basic debugger

- Now we need to return

- 0x0010F6D0 : 0E F0 A0 E1 # MOV PC, LR (RET)

- Looks perfect

```
ROM:0010F6A8 ; START OF FUNCTION CHUNK FOR sub_26F0
ROM:0010F6A8
ROM:0010F6A8 loc_10F6A8 ; CODE XREF: sub_26F0+24↑j
ROM:0010F6A8 ADR
ROM:0010F6AC BL
ROM:0010F6B0 RET
ROM:0010F6B0 ; END OF FUNCTION CHUNK FOR sub_26F0
ROM:0010F6B0 ;
```
Process: Moving on to advanced debugging

- From here on out we can reverse engineer more functions to perform more advanced actions
- What we developed so far can be used for debugging, in case something goes wrong.
- We will focus on reverse engineering;
  - getchar(), so we can interact with our running code
  - read/write memory functions to evaluate and modify system behavior
Advanced Debugging Capabilities: Goals

We want to create an interactive debugger using the serial input and output

- We will implement a loop that in pseudocode would be similar to this:
  ```
  While (1):
    char = getchar()
    if char == 0:
      exit
    if char == 6:
      print ”tst”
    if char == x:
      do_y()
  ```

- We already have the printf() – for writing output
- We need the getchar() function – for capturing user input
Advanced Debugging Capabilities: finding getchar()

```c
int sub_23C8()
{
    signed int v0; // r4@3
    int result; // r0@2
    char v2; // [sp+0h] [bp-6Ch]@1
    char v3; // [sp+20h] [bp-4Ch]@3

    sub_61908(&v2, "\r\n%s>>", 1816207461);
    if ( sub_5C0D4("/dev/serial0", 1287876) )
    {
        result = sub_61B2C("cyg_io_lookup error\r");
    }
    else
    {
        do
        {
            sub_5D320(&v3, 0, 51);
            sub_644C0(&v2);
            sub_2108(&v3);
            v0 = sub_2358((int)&v3);
            result = sub_5D4AC(100, 0);
        }
        while ( !v0 );
    }
    return result;
}
```
Advanced Debugging Capabilities: finding `getchar()`

```c
int sub_23C8()
{
    signed int v0; // r4@3
    int result; // r0@2
    char v2; // [sp+0h] [bp-6Ch]@1
    char v3; // [sp+20h] [bp-4Ch]@3

    sub_61908(&v2, "\r\n\%s\", 1816207461);
    if ( sub_5C0D4("/dev/serial0", 1287876) )
    {
        result = sub_61B2C("cyg_io_lookup error\r");
    }
    else
    {
        do
        {
            sub_5D32E(&v3, 0, 51);
            sub_61BC8(&v2);
            sub_2108(&v3);
            v0 = sub_2358((int)&v3);
            result = sub_5D4AC(100, 0);
        }
        while ( !v0 );
    }
    return result;
}
```
Advanced Debugging Capabilities
Advanced Debugging Capabilities
Advanced Debugging Capabilities: our `getchar()`

- Byte `getchar(void)`, returns 1 char in R0

```assembly
/// int getc(), returns 1 character
len = -0x28
buffer = -0x21

getc:
  MOV R12, SP
  STMFD SP!, {R1-R8,R11,R12,LR,PC}
  SUB R11, R12, #4
  SUB SP, SP, #8
  MOV R7, #0
  MOV R3, #1
  STRB R7, [R11,#buffer]
  STR R3, [R11,#len]
  SUB R6, R11, #-buffer
  SUB R5, R11, #-len

lloop_char:
  @ CODE XREF: getline_uart
  LDR R3, =0x13A6C4
  MOV R1, R6
  LDR R0, [R3]
  MOV R2, R5
  BL cyg_io_read
    @ relative!!!!!
  CMP R0, #0
  BNE loop_char
  LDRB R2, [R11,#buffer]
  AND R0, R2, #0xFF
  LDMDB R11, {R1-R8,R11,SP,PC}
```
Advanced Debugging Capabilities: the assembler

- We need the assembler to know where cyg_io_read is (and also printf)
- The ORG directive helps us doing that
- The _start directive tells the assembler at what memory address the code is located, so that Branches can be calculated accordingly
Advanced Debugging Capabilities: calling getchar()

- Basic loop that will read a character and exit if it’s 0
- If the character is 6, then the shellcode will print “tst”
Advanced Debugging Capabilities: printf()

- If the character is 6, then the shellcode will print “tst”
- We define the string; 'tst/n', at the bottom, in the code section
Advanced Debugging Capabilities: first compile

- Write the whole assembly code in a text file
Advanced Debugging Capabilities

• Assemble everything
  ```
  rem Build the object code
  MinGW-arm-eabi-glo-5.2.0\bin\arm-eabi-as.exe -EL -o build\code.o src\code.s
  ```

• Link and convert the object code to binary
  ```
  rem running the linker
  MinGW-arm-eabi-glo-5.2.0\bin\arm-eabi-ld.exe build\code.o -o release\code.bin -Ttext-segment 0x00000000 -s --gc-sections
  MinGW-arm-eabi-glo-5.2.0\bin\arm-eabi-objcopy.exe -O binary release\code.bin
  ```

• Split the file, so that only the shellcode at 0x11C394 is left
  ```
  rem strip the first bytes 1164180 (0x0011c394), so we are left with just the gdb code
  del release\code.part_aa
  del release\gdb
  ren release\code.part_ab gdb
  PAUSE
  ```
Advanced Debugging Capabilities

- We have getc
- We have printf
- We can exit
- The initial loop is done (just with exit functionality)
- Time to add some debugging capabilities
Advanced Debugging Capabilities

- Adding functionality for reading words (so that we can use them to read/write memory addresses)
Advanced Debugging Capabilities

- Adding read memory capabilities (read and display memory to the user)
Advanced Debugging Capabilities

• Adding write memory capabilities (write into the device memory)

```
tst2:  CMP R6, #2
BNE tst3
BL getw
MOV R2, R0
BL getw
STR R0, [R2]
```

@ memory address to write
@ and store in R2
@ memory value to write
@ write memory
Advanced Debugging Capabilities

Breakpoints: stop normal execution for analysis of current processor state, and code stepping

Creating our breakpoint mechanism:
- stopping code execution
- calling our debugger from the running code

We just patch a branch-instruction from where we want to call our debugger, and restore the original instruction on exit
Advanced Debugging Capabilities: calling our debugger

• Storing registers on the stack on initial call

```
PC_addr = 0x38
_start:  STMFD
         MRS
         MOV
         STMFD

   sp!, {R0, R12, LR, PC}  @store registers on stack(also PC)
   R10, SPSR
   R9, SP
   sp!, {R9,R10}           @store SPSR
   @ breakpoint restore
   @ load addr of heap in r9
   @ store also SP and SPSR

LDMFD
MOV
MSR
LDMFD
   sp!, {R9,R10}          @ restore also SP and SPSR
   SP, R9
   CPSR_cf, R10
   SP!, {R0,R12, LR, PC}  @ restore CPSR
   @ restore registers on stack, and jump out of routine
```
Advanced Debugging Capabilities

- Adding read register capabilities

```assembly
  tst3:
  CMP R6, '#3'
  BNE tst4
  BL getc @ register index to print
  LSL R0, #2
  ADD R2, R0, R9 @ multiply the register-number by 4, and add to R9
  LDR R1, [R2, #-8] @ offset of stack -2, for cspr and SP
  LDR R0, =string
  BL printf @ read registers on stack, relative!!!!!
```
Advanced Debugging Capabilities

- Adding write register capabilities

```assembly
tst4:    CMP     R6, #'4'       @ register index to write
         BNE    tst5
         BL     getc
         LSL    R0, #2
         ADD    R2, R9, R0       @ multiply the register-number by 4, and add to R9
         BL     getw
         STR    R0, [R2, #-8]    @ value to write
                         @ write registers on stack (offset-2, for CSPR and SP)
```
Advanced Debugging Capabilities

- Adding breakpoint capabilities: restoring the original code
Advanced Debugging Capabilities

• Literal pools and string definitions

```assembly
.1torg
@data    = 0xEB010101
@address = 0xFFFF2714
@string  -> "%8x\n"
@tst     -> "tst\n"
@getc_c  = 0x0013A6C4

string: .asciz "%8x\n"
tst:     .asciz "tst\n"
.end
```

@literal pool for data used in main
@variable to hold data at breakpoint
@variable to hold address of breakpoint
@string to be printed, with data in R1 in hex
@test-string
@address of call to cyg_io_read dma-ish buffer
Advanced Debugging Capabilities

- Adding breakpoint capabilities: writing a new breakpoint

```
tst5:    CMP    R6, #'5'
  tst6   getw
  BL     getw
  STR    R0, [PC,#0xC0]
  BL     getw
  MOV    R7, R0
  MOV    R8, #1

  @ get breakpoint address from input
  @ store breakpoint in addr
  @ get instruction from input
  @ the new instruction, that contains: ((((PC-8) - R1)/4 + 2) | 0xEB000000); @relative branch from addr to this code(PC-8), ensure to write it little-endian
```
Advanced Debugging Capabilities

- Adding breakpoint capabilities: writing a new breakpoint, and storing the old instruction for restoration on the next call

```assembly
exit:  CMP      R8, #0
      BNEQ     nopatch

LDR     R0, =0x00002714
LDR     R1, [R0]
STR     R1, [PC, #0x88]
STR     R7, [R0]

nopatch:
  LDMFD   sp!, (R0,R10)
  MOV     SP, R0
  MSR     CPSR_cf, R10
  LDMFD   SP!, (R0-R12, LR, PC)
```

@ check if we need to patch an new instruction
@ jump over the patcher

@ -set new breakpoint--
@ load new addr,
@ load instruction from the addr where we intend to set the breakpoint in R1
@ store the original instruction in data, so it can be restored
@ patch the instruction with the breakpoint at the new
@ address, R0 was already loaded with the right address, R7 contained the instruction, from when we set it, above

@ --return to program--
@ restore also SP and CPSR
@ modify SP, if desired
@ restore CPSR
@ restore registers on stack, and jump out of routine
Advanced Debugging Capabilities

• We now have a basic debugger that can read/write memory, read/write registers and set breakpoints.

• We have written everything in Assembler language, now we need to assemble it again
Advanced Debugging Capabilities

- Uploading the debugger on the device requires some more manual tasks to prepare the binary with a hex editor.

- The `sys mem` function jump that we used at the beginning must be patched to jump to our shellcode, by hex editor.

- The shellcode must be added to the firmware file at a suitable place such as the address `0x11c394` by hex editor.

- Check the code in IDA before uploading. Make sure the code works as expected (it should look almost identical with the code in `code.s`).
Advanced Debugging Capabilities

• Upload the firmware on the device

• Run sys mem in the console

• Press 6 to check if it works. The console should display “tst”

• Test other capabilities

• Next steps…
Conclusions & implications

• Use proper firmware verification
• Use hardware based integrity verification checks
• Don’t connect ICS to the internet 😊
Questions?