
Applied Risk

# 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 trough 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

# Used ARM Instruction Set

| **Mnemonics** | **Description** |
| --- | --- |
| ADD | add two 32 bit values |
| AND | logical bitwise AND of two 32 bit values |
| B | branch relative +/- 32MB |
| BL | relative branch with link |
| CMP | compare two 32 bit values |
| LDR(B) | load a single value from a virtual address in memory |
| LDMFD | store multiple 32 bit registers to memory |
| MOV | move a 32 bit value into a register |
| MRS | move to ARM register from a status register (cpsr or spsr) |
| MSR | move to a status register (cpsr or spsr) from an ARM register |
| ORR | logical bitwise OR of two 32 bit values |
| STMFD | store multiple 32 bit registers to memory |
| STR(B) | store register to a virtual address in memory |
| SUB | subtract two 32 bit values |
| LSL | left shift a 32 bit register |

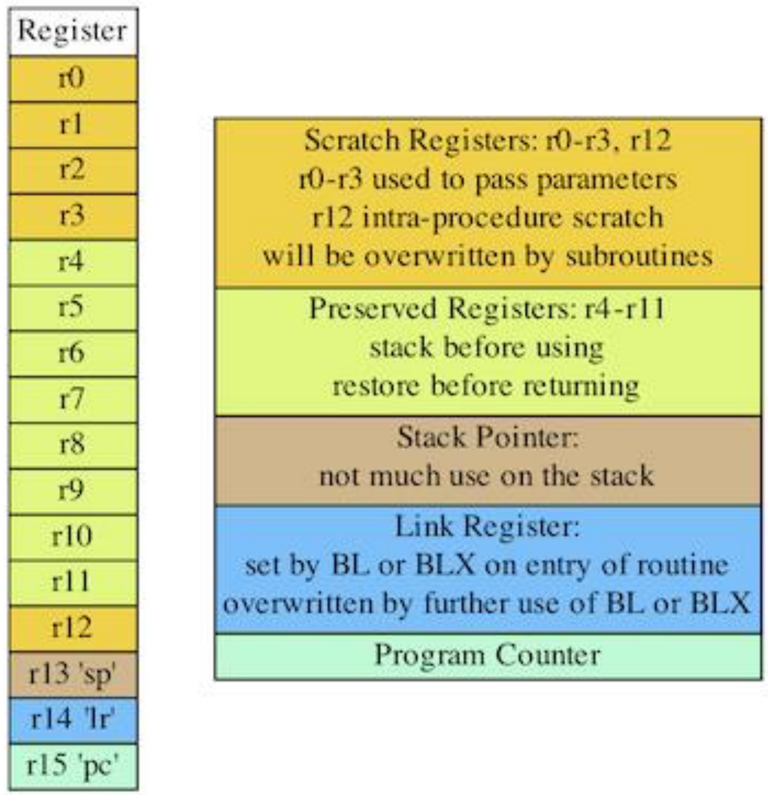
Applied Risk

# ARM Condition field modifiers

Table A3-1 Condition codes

| Opcode [31:28] | Mnemonic extension | Meaning | Condition flag state |
|---|---|---|---|
| 0000 | EQ | Equal | Z set |
| 0001 | NE | Not equal | Z clear |
| 0010 | CS/HS | Carry set/unsigned higher or same | C set |
| 0011 | CC/LO | Carry clear/unsigned lower | C clear |
| 0100 | MI | Minus/negative | N set |
| 0101 | PL | Plus/positive or zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No overflow | V clear |
| 1000 | HI | Unsigned higher | C set and Z clear |
| 1001 | LS | Unsigned lower or same | C clear or Z set |
| 1010 | GE | Signed greater than or equal | N set and V set, or N clear and V clear (N == V) |
| 1011 | LT | Signed less than | N set and V clear, or N clear and V set (N != V) |
| 1100 | GT | Signed greater than | Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V) |
| 1101 | LE | Signed less than or equal | Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V) |
| 1110 | AL | Always (unconditional) | - |
| 1111 | - | See *Condition code 0b1111* | - |

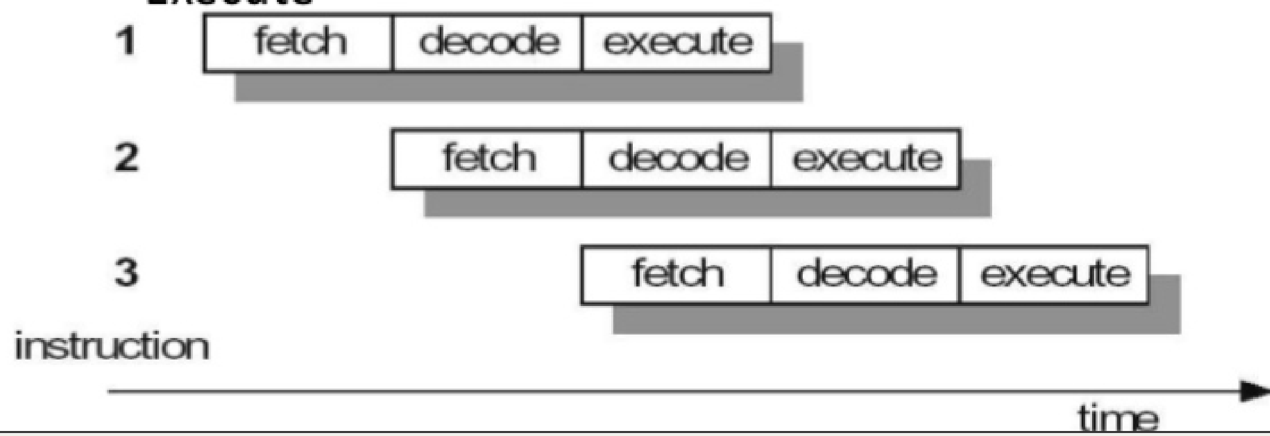
# ARM Registers




Register Use in the ARM Procedure Call Standard

# Pipelining

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



Applied Risk

# Outline

- Output console
- Initial Infection
- Initial firmware patch with basic debugger
- Advanced debugging capabilities implementation

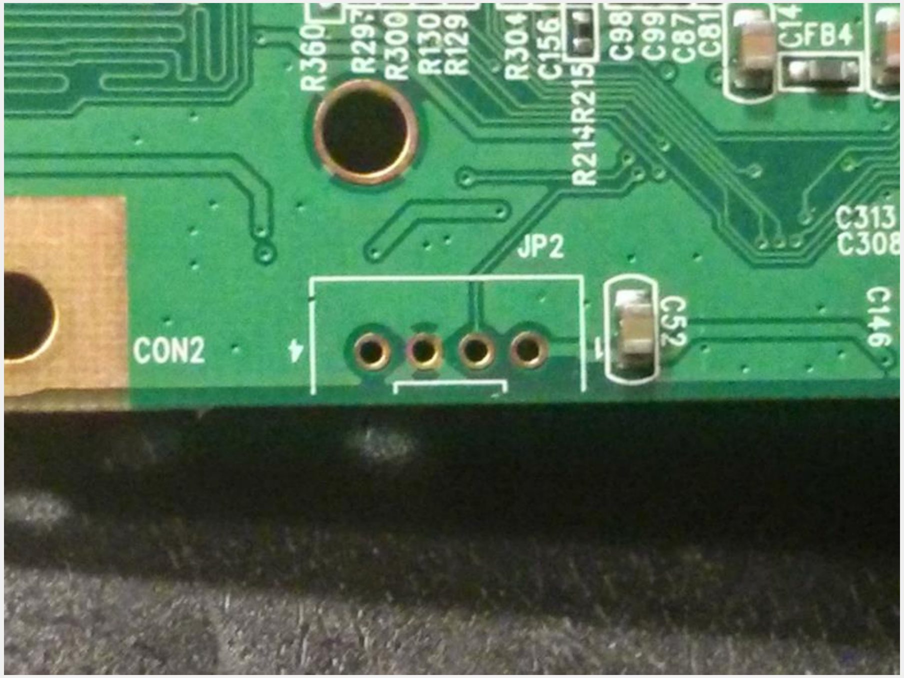
# The process we will follow

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

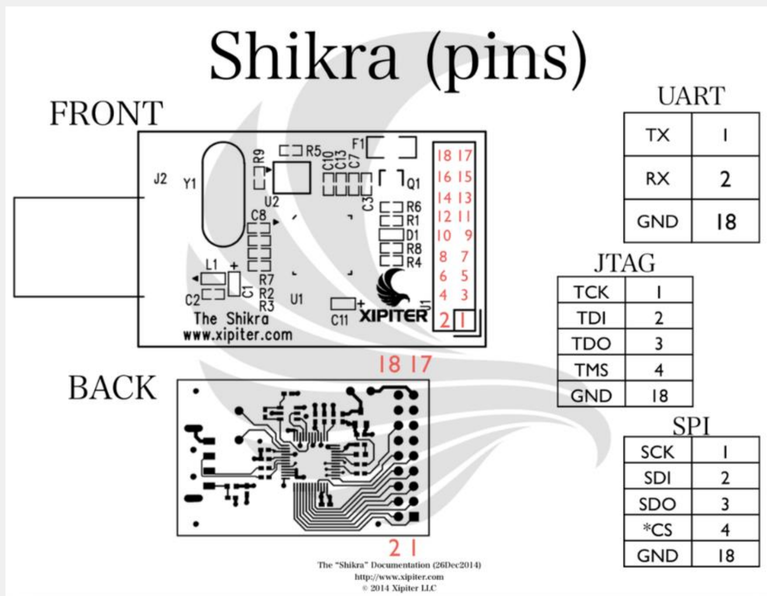
# Process: Step 1

**Evaluate ways to interact with the device (UART, webserver, …)**



# Interacting with the device: Output Console

- UART, Serial, JTAG, others



# Process: Step 2

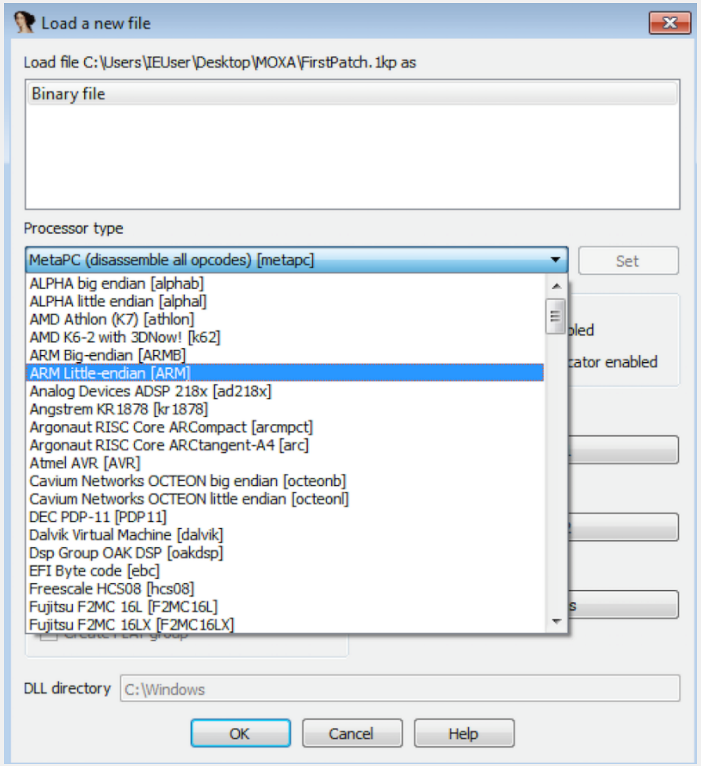
**Evaluate ways to make the device run custom code**

# 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)

```
E1242>>
E1242>>sys mem
Free/Total memory => 1431740/1620888
```

# Initial basic debugger

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

```
E1242>>
E1242>>sys mem
Free/Total memory => 1431740/1620888
```

Applied Risk

# Initial basic debugger

- Let’s find the function in IDA

```
ROM:000026F0 sub_26F0                                ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                         ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                 LDR             R0, =aFreeTotalMemor ; "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718                 LDMDB           R11, {R11,SP,PC}
```

# Initial basic debugger

- This is the assembly for the function “sys mem”
- We will patch this to jump to our ”initial patch”

```
ROM:000026F0 sub_26F0                                ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                         ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                 LDR             R0, =aFreeTotalMemor ; "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718                 LDMDB           R11, {R11,SP,PC}
```

# Process: Step 4

**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

```
ROM:000026F0 sub_26F0                                  ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                           ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                                           mor ; "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718
```

Applied Risk

# 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)

```
ROM:000026F0 sub_26F0                                ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                         ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                 LDR             R0, =aFreeTotalMemor ; "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718                 LDMDB           R11, {R11,SP,PC}
```

# 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

```
ROM:000026F0 sub_26F0                                ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                         ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                                          "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718
```

# 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

```
ROM:000026F0 sub_26F0                                       ; CODE XREF: sub_27B4+60↓j
ROM:000026F0                                                ; DATA XREF: ROM:0012A180↓o
ROM:000026F0
ROM:000026F0 var_38          = -0x38
ROM:000026F0 var_10          = -0x10
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                 LDR             R0, =aFreeTotalMemor ; "Free/Total memory => %d/%d\r\n"
ROM:00002714                 BL              sub_644C0
ROM:00002718                 LDMDB           R11, {R11,SP,PC}
```

Applied Risk

# 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

```
000026F4   00 D8 2D E9
00002704   16 78 01 EB
00002714   69 87 01 EB
00002724   30 D8 2D E9
00002734   40 50 4B E2
00002744   05 00 A0 E1
```

# 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

| | | |
|---|---|---|
| 10F6C0 | 616C0D0A 00000000 | al |
| 10F6C8 | 79203D3E 2025642F | y => %d/ |
| 10F6D0 | 25640D0A 00000000 | %d |
| 10F6D8 | 2D2D2D2D 00000000 | ---- |
| 10F6E0 | 49443A25 3032582C | ID:%02X, |
| 10F6E8 | 204E616D 653A2532 | Name:%2 |
| 10F6F0 | 35732C20 5072696F | 5s, Prio |
| 10F6F8 | 72697479 3A253032 | rity:%02 |
| 10F700 | 642C2053 74617465 | d, State |
| 10F708 | 3A253032 642C2055 | :%02d, U |
| 10F710 | 73656453 7461636B | sedStack |
| 10F718 | 3A25642F 25640D0A | :%d/%d |
| 10F720 | 00000000 6D69696E | miin |
| 10F728 | 655F666C 6173685F | e_flash_ |
| 10F730 | 77726974 655F6461 | write_da |
| 10F738 | 74613A20 6D616C6C | ta: mall |
| 10F740 | 6F632062 75666665 | oc buffe |
| 10F748 | 72206661 696C2E0D | r fail. |
| 10F750 | 0A000000 72657365 | rese |

# Initial basic debugger

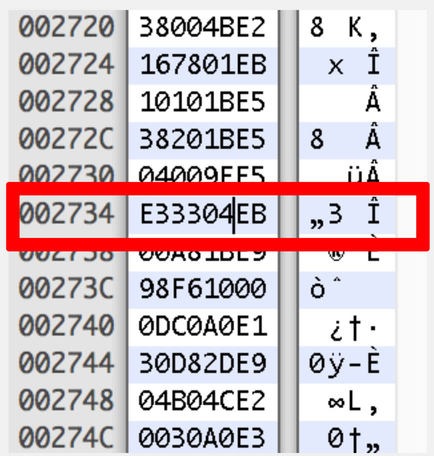
- We will modify the string at address 0x10F6A2 (with the initial offset it will be 0x10F6A2 +0x20 )
- Add 0D 0A 00 00 00 00 (We will end the string with a null terminator)

| | | | | | |
|---|---|---|---|---|---|
| 0010F690 | 3D | 3D | 3D | 3D | ==== |
| 0010F694 | 0D | 00 | 00 | 00 | .... |
| 0010F698 | 46 | 72 | 65 | 65 | Free |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 0010F6A0 | 61 | 6C | 0D | 0A | al.. |
| 0010F6A4 | 00 | 00 | 00 | 00 | .... |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 0010F6AC | 20 | 25 | 64 | 2F | %d/ |
| 0010F6B0 | 25 | 64 | 0D | 0A | %d.. |
| 0010F6B4 | 00 | 00 | 00 | 00 | .... |
| 0010F6B8 | 2D | 2D | 2D | 2D | ---- |
| 0010F6BC | 00 | 00 | 00 | 00 | |

| | | | |
|---|---|---|---|
| 10F6C0 | 616C0D0A | 00000000 | al |
| 10F6C8 | 79203D3E | 2025642F | y => %d/ |
| 10F6D0 | 25640D0A | 00000000 | %d |
| 10F6D8 | 2D2D2D2D | 00000000 | ---- |
| 10F6E0 | 49443A25 | 3032582C | ID:%02X, |
| 10F6E8 | 204E616D | 653A2532 | Name:%2 |
| 10F6F0 | 35732C20 | 5072696F | 5s, Prio |
| 10F6F8 | 72697479 | 3A253032 | rity:%02 |
| 10F700 | 642C2053 | 74617465 | d, State |
| 10F708 | 3A253032 | 642C2055 | :%02d, U |
| 10F710 | 73656453 | 7461636B | sedStack |
| 10F718 | 3A25642F | 25640D0A | :%d/%d |
| 10F720 | 00000000 | 6D69696E | miin |
| 10F728 | 655F666C | 6173685F | e_flash_ |
| 10F730 | 77726974 | 655F6461 | write_da |
| 10F738 | 74613A20 | 6D616C6C | ta: mall |
| 10F740 | 6F632062 | 75666665 | oc buffe |
| 10F748 | 72206661 | 696C2E0D | r fail. |
| 10F750 | 0A000000 | 72657365 | rese |

# 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

```
ROM:000026F0
ROM:000026F0                 MOV             R12, SP
ROM:000026F4                 STMFD           SP!, {R11,R12,LR,PC}
ROM:000026F8                 SUB             R11, R12, #4
ROM:000026FC                 SUB             SP, SP, #0x2C
ROM:00002700                 SUB             R0, R11, #-var_38
ROM:00002704                 BL              sub_60764
ROM:00002708                 LDR             R1, [R11,#var_10]
ROM:0000270C                 LDR             R2, [R11,#var_38]
ROM:00002710                 LDR             R0, =aFreeTotal ; "Free/Total\r\n"
ROM:00002714                 BL              loc_10F6A8
ROM:00002718                 LDMDB           R11, {R11,SP,PC}
```

# Initial basic debugger

- Well the code is not really what we expected
- But we didn’t patch it to contain our code
- Yet

```
ROM:0010F6A8 loc_10F6A8                              ; CODE XREF: sub_26F0+24↑p
ROM:0010F6A8                 MRCCC           p0, 1, R2,c13,c9, 3
ROM:0010F6AC                 SVCCS           0x642520
ROM:0010F6B0                 BEQ             0x46874C
ROM:0010F6B4                 ANDEQ           R0, R0, R0
ROM:0010F6B8                 STCCS           p13, c2, [SP,#-0xB4]!
ROM:0010F6BC                 ANDEQ           R0, R0, R0
ROM:0010F6C0                 LDRCS           R4, [R10,#-0x449]!
ROM:0010F6C4                 MRRCCS          p2, 3, R3,R8,c0
ROM:0010F6C8                 STCVSL          p14, c4, [R1,#-0x80]!
ROM:0010F6CC                 EORCC           R3, R5, #0x65000
ROM:0010F6D0                 EORCS           R7, R12, R5,LSR R3
ROM:0010F6D4                 SVCVS           0x697250
ROM:0010F6D8                 LDMVCDB         R4!, {R1,R4-R6,R8,R11,SP,LR}^
ROM:0010F6DC                 EORCCS          R2, R0, #0xE800000
ROM:0010F6E0                 TEQPL           R0, #0x6400
ROM:0010F6E4                 LDRVSB          R6, [R4,#-0x174]!
ROM:0010F6E8                 EORCCS          R2, R0, #0xE800000
ROM:0010F6EC                 STRPL           R2, [R0,#-0xC64]!
ROM:0010F6F0                 CMNPL           R4, #0x1CC00000
ROM:0010F6F4                 BLVS            0x19E7CCC
ROM:0010F6F8                 SVCCS           0x64253A
ROM:0010F6FC                 BEQ             0x468798
ROM:0010F700                 ANDEQ           R0, R0, R0
ROM:0010F704                 CDPVS           p9, 6, c6,c9,c13, 3
ROM:0010F708                 STCVSL          p15, c5, [R6],#-0x194
ROM:0010F70C                 SVCPL           0x687361
ROM:0010F710                 STRVCBT         R7, [R9],#-0x277
```

# Initial basic debugger

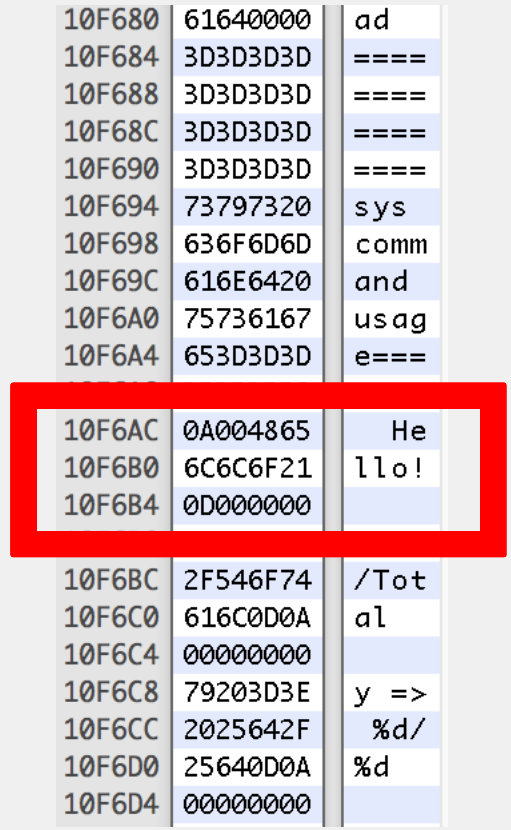
- This location must be patched to do the following:
  - Patch a string in memory
  - Load a string in the R0 register
  - Jump to printf
  - Return (load link register in PC)

# 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



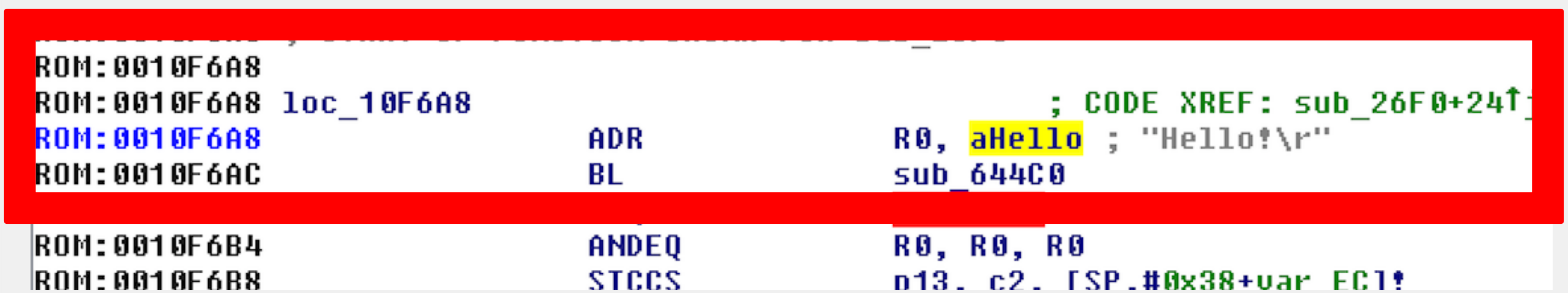
# Initial basic debugger

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

```
ROM:0010F6A8
ROM:0010F6A8 loc_10F6A8                              ; CODE XREF: sub_26F0+24↑p
ROM:0010F6A8                 ADR             R0, aHello ; "Hello!\r"
ROM:0010F6AC                 SVCCS           0x642520
ROM:0010F6B0                 BEQ             0x46874C
```

# 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             R0, aHello ; "Hello!\r"
ROM:0010F6AC                 BL              sub_644C0
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

Applied Risk

# 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()

```
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()

```
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;
}
```

Applied Risk

# Advanced Debugging Capabilities

```
ROM:00002108 sub_2108                                       ; CODE XREF: sub_23C8+64↓p
ROM:00002108
ROM:00002108 var_28          = -0x28
ROM:00002108 var_21          = -0x21
ROM:00002108
ROM:00002108                 MOV             R12, SP
ROM:0000210C                 STMFD           SP!, {R4-R8,R11,R12,LR,PC}
ROM:00002110                 MOV             R3, #1
ROM:00002114                 SUB             R11, R12, #4
ROM:00002118                 SUB             SP, SP, #8
ROM:0000211C                 MOV             R7, #0
ROM:00002120                 MOV             R8, R0
ROM:00002124                 STRB            R7, [R11,#var_21]
ROM:00002128                 STR             R3, [R11,#var_28]
ROM:0000212C                 SUB             R6, R11, #-var_21
ROM:00002130                 SUB             R5, R11, #-var_28
ROM:00002134
ROM:00002134 loc_2134                                       ; CODE XREF: sub_2108+48↓j
ROM:00002134                                                ; sub_2108+6C↓j ...
ROM:00002134                 LDR             R3, =0x13A6C4
ROM:00002138                 MOV             R1, R6
ROM:0000213C                 LDR             R0, [R3]
ROM:00002140                 MOV             R2, R5
ROM:00002144                 BL              sub_5C1F4
ROM:00002148                 CMP             R0, #0
ROM:0000214C                 MOV             R4, R0
ROM:00002150                 BNE             loc_2134
ROM:00002154                 LDRB            R2, [R11,#var_21]
ROM:00002158                 AND             R3, R2, #0xFF
ROM:0000215C                 CMP             R3, #0xD
ROM:00002160                 BEQ             loc_21CC
ROM:00002164                 BGT             loc_21C4
ROM:00002168                 CMP             R3, #8
ROM:0000216C
ROM:0000216C loc_216C                                       ; CODE XREF: sub_2108+C0↓j
ROM:0000216C                 BNE             loc_2190
ROM:00002170                 CMP             R7, #0
ROM:00002174                 BEQ             loc_2134
ROM:00002178                 MOV             R3, #0
ROM:0000217C                 STRB            R3, [R8,R7]
ROM:00002180                 MOV             R0, #1
ROM:00002184                 SUB             R7, R7, #1
ROM:00002188                 BL              sub_207C
```

# Advanced Debugging Capabilities

```
ROM:00002108 sub_2108                                       ; CODE XREF: sub_23C8+64↓p
ROM:00002108
ROM:00002108 var_28          = -0x28
ROM:00002108 var_21          = -0x21
ROM:00002108
ROM:00002108                 MOV             R12, SP
ROM:0000210C                 STMFD           SP!, {R4-R8,R11,R12,LR,PC}
ROM:00002110                 MOV             R3, #1
ROM:00002114                 SUB             R11, R12, #4
ROM:00002118                 SUB             SP, SP, #8
ROM:0000211C                 MOV             R7, #0
ROM:00002120                 MOV             R8, R0
ROM:00002124                 STRB            R7, [R11,#var_21]
ROM:00002128                 STR             R3, [R11,#var_28]
ROM:0000212C                 SUB             R6, R11, #-var_21
ROM:00002130                 SUB             R5, R11, #-var_28
ROM:00002134
ROM:00002134 loc_2134                                       ; CODE XREF: sub_2108+48↓j
ROM:00002134                                                ; sub_2108+6C↓j ...
ROM:00002134                 LDR             R3, =0x13A6C4
ROM:00002138                 MOV             R1, R6
ROM:0000213C                 LDR             R0, [R3]
ROM:00002140                 MOV             R2, R5
ROM:00002144                 BL              sub_5C1F4
ROM:00002148                 CMP             R0, #0
ROM:0000214C                 MOV             R4, R0
ROM:00002150                 BNE             loc_2134
ROM:00002154                 LDRB            R2, [R11,#var_21]
ROM:00002158                 AND             R3, R2, #0xFF
ROM:0000215C                 CMP             R3, #0xD
ROM:00002160                 BEQ             loc_21CC
ROM:00002164                 BGT             loc_21C4
ROM:00002168                 CMP             R3, #8
ROM:0000216C
ROM:0000216C loc_216C                                       ; CODE XREF: sub_2108+C0↓j
ROM:0000216C                 BNE             loc_2190
ROM:00002170                 CMP             R7, #0
ROM:00002174                 BEQ             loc_2134
ROM:00002178                 MOV             R3, #0
ROM:0000217C                 STRB            R3, [R8,R7]
ROM:00002180                 MOV             R0, #1
ROM:00002184                 SUB             R7, R7, #1
ROM:00002188                 BL              sub_207C
```

# Advanced Debugging Capabilities: our getchar()

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

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@ --- 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
loop_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

```
.text
.org    0x0005c1f4                  @offset of function for cyg_io_read
cyg_io_read:
.org    0x000644C0                  @offset of print_text_uart, alternatives are kprintf_2():0x0005D01C, kprintf():0x0005D0A8
printf:

.global _start
.org 0x0011C394                     @free space in binary, setting this will ensure BL offset is correct
```

# 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"

```
loop:   BL          getc            @ selection of option:
        MOV         R6, R0          @store selection into R6

        CMP         R6, #'0'
        BEQ         exit            @ exit the debugger, and resume program until next breakpoint
```

# 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

```
                                                    @b=test()
        MOV             R6, R0                      @store selection into R6

        CMP             R6, #'0'
        BEQ             exit                        @ exit the debugger, and resume program until next breakpoint

tst6:   CMP             R6, #'6'
        BNE             loop
        LDR             R0, =tst
        BL              printf                       @ test/reset the interface, relative!!!
        B               loop                         @loop forever until exit is selected
       tst:    .asciz  "tst\n"
```

# Advanced Debugging Capabilities: first compile

- Write the whole assembly code in a text file

```
.text
.org    0x0005c1f4                                     @offset of function for cyg_io_read
cyg_io_read:
.org    0x000644C0                                     @offset of printf
printf:

.global _start
.org 0x0011C394                                        @free space in binary, setting this will ensure BL offset is correct
@@@            --main code--                         @@@
_start: STMFD           sp!, {R0-R12, LR ,PC}   @store registers on stack(also PC)
loop:   BL              getc                           @ selection of option:
                                                       @0=exit(),
                                                       @6=test()
        MOV             R6, R0                         @store selection into R6

        CMP             R6, #'0'
        BEQ             exit                           @ exit the debugger, and resume program until next breakpoint

tst6:   CMP             R6, #'6'
        BNE             loop
        LDR             R0, =tst
        BL              printf                         @ test/reset the interface,
        B               loop                           @loop forever until exit is selected

exit:   LDMFD           SP!, {R0-R12, LR, PC}          @ restore registers on stack, and jump out of routine
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@ --- 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
loop_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}


tst:    .asciz  "tst\n"
.end
```

# 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
MinGW-arm-eabi-glo-5.2.0\bin\split.exe --bytes=1164180 release\code.bin release\code.part_
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)

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
getw:   stmfd           SP!,{r2, lr}
        BL              getc                        @ memory value to write
        LSL             R1, R0, #0x18               @ shift 24 bits
        BL              getc                        @
        ORR             R1, R0, LSL #0x10           @ shift 16 bits
        BL              getc                        @
        ORR             R1, R0, LSL #0x08           @ shift 8 bits
        BL              getc                        @
        ORR             R0, R0, R1                  @ shift 8 bits  , and store in R0
        ldmfd           sp!,{r2, PC}
```

# Advanced Debugging Capabilities

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

```
        CMP         R6, #'1'
        BNE         tst2
        BL          getw                @ memory address to read
        LDR         R1, [R0]            @load memory at this location
        LDR         R0, =string
        BL          printf              @ read memory, relative!!!!!
string: .asciz  "%08x\n"
```

Applied Risk

# Advanced Debugging Capabilities

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

```
tst2:   CMP     R6, #'2'
        BNE     tst3
        BL      getw            @ memory address to write
        MOV     R2, R0          @ and store in R2
        BL      getw            @ memory value to write
        STR     R0, [R2]        @ 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

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@             --main code--                     @@@
PC_addr     = 0x38                                  @top of stack + 15*4 bytes = 60
_start: STMFD           sp!, {R0-R12, LR ,PC}   @store registers on stack(also PC)
        MRS             R10, SPSR                   @store SPSR
                                                    @-breakpoint restore--
        MOV             R9, SP                      @ load addr of heap in r9
        STMFD           sp!,{R9,R10}                @ store also SP and SPSR
```

- Restoring the registers on exit

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

# Advanced Debugging Capabilities

- Adding read register capabilities

```
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

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

# Advanced Debugging Capabilities

- Adding breakpoint capabilities: restoring the original code

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@          --main code--                      @@@
PC_addr      = 0x38                               @top of stack + 15*4 bytes = 60
_start: STMFD           sp!, {R0-R12, LR ,PC}   @store registers on stack(also PC)
        MRS             R10, SPSR                 @store SPSR
                                                  @-breakpoint restore--
        MOV             R9, SP                    @ load addr of heap in r9
        STMFD           sp!,{R9,R10}              @ store also SP and SPSR
        LDR             R0, =0xEB018769           @ load data from literal
        LDR             R1, =0x00002714           @ load addr from literal
        STR             R0, [R1]                  @ store data to addr; restore instuction at breakpoint
        STR             R1, [R9, #PC_addr]        @ load return addr in stack_PC, so overwrite the PC on the stack with the
```

# Advanced Debugging Capabilities

- Literal pools and string definitions

```
.ltorg                                             @literal pool for data used in main
@data     = 0xEB010101                             @variable to hold data at breakpoint
@addres   = 0xFFFF2714                             @variable to hold address of breakpoint
@string   -> "%08x\n"                              @string to be printed, with data in R1 in hex
@tst      -> "tst\n"                               @test-string
@getc_c   = 0x0013A6C4                             @address of call to cyg_io_read dma-ish buffer

string: .asciz  "%08x\n"
tst:    .asciz  "tst\n"
.end
```

# Advanced Debugging Capabilities

- Adding breakpoint capabilities: writing a new breakpoint

```
tst5:   CMP     R6, #'5'
        BNE     tst6
        BL      getw                @ get breakpoint address from input
        STR     R0, [PC,#0xC0]      @ store breakpoint in addr
        BL      getw                @ get instruction from input
        MOV     R7, R0              @ 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
        MOV     R8, #1
```

# Advanced Debugging Capabilities

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

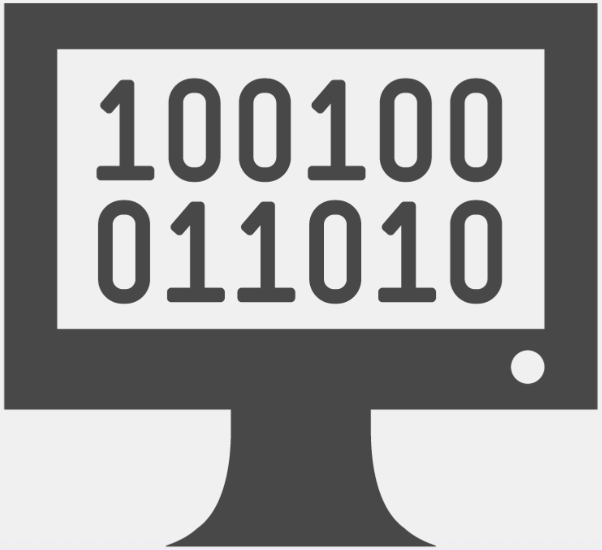
```
exit:   CMP         R8, #0                      @ check if we need to patch an new instruction
        BLEQ        nopatch                     @ jump over the patcher

                                                @--set new breakpoint--
        LDR         R0, =0x00002714             @ load new addr,
        LDR         R1, [R0]                    @ load instruction from the addr where we intend to set the breakpoint in R1
        STR         R1, [PC,#0x88]              @ store the original instruction in data, so it can be restored
        STR         R7, [R0]                    @ patch the instruction with the breakpoint at the new
                                                @address, R0 was allready loaded with the right address, R7 contained the instruction, from when we set it, above

nopatch:                                        @ --return to program--
        LDMFD       sp!,{R9,R10}                @ restore also SP and SPSR
        MOV         SP, R9                      @ modify SP, if desired
        MSR         CPSR_cf, R10                @ restore CPSR
        LDMFD       SP!, {R0-R12, LR, PC}       @ 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?