21:04 Anti-debugging tips and tricks for Cortex-M microcontrollers

by Balda

ARM-based microcontrollers are ubiquitous in the so-called *smart* devices we all live around. If you take the time to open these up, you will most certainly find an accessible JTAG interface, or more often now using SWD. As a security-aware person you might say that these interfaces should be disabled at the factory, but they most of the times are not for multiple and often non-relevant reasons like failure analysis and such. As a firmware developer, this interface is also a nightmare as any curious person with the right tools would be able to access the internal secrets held inside the flash memory.

The purpose of this article is to provide fellow firmware developers some ways to detect a debug access from the firmware itself and react to such undesirable intrusion. We also will focus on ARM's Cortex-M family of microcontrollers.

Debugging a Cortex-M core

For nearly all of the Cortex-M cores out there, the most used way to access the debug interface is through the SWD port. The SWD protocol itself is extensively described in ARM's Debug Interface Architecture Specification or ADI, which is freely available from ARM's website. From that document, we know that the interface uses a memory access controller which can read and write to arbitrary locations called the MEM-AP. This means that all subsequent debug operations are performed using memory reads and writes to specific memory-mapped registers.

One of these registers is the Debug Halting Control and Status Register, DHCSR for short. This 32-bit register is located at address `0xE000EDF0` and is used by debuggers to control the core execution state and contains several control bits. Two of them are very interesting: `C_HALT[1]` halts the core execution, and `C_DEBUGEN[0]` enables core debug.

To set `C_HALT` and stop the core, `C_DEBUGEN` must already be set to 1. This means that a debugger has to perform two writes to this register in order to stop the core. As this register can also be read from the core itself, it is possible to detect if a debugger is trying to connect by looking at the `C_DEBUGEN` bit value.

```
1 uint8_t detect_debug(void) {
    uint32_t *DHCSR=(uint32_t *) 0xE000EDF0;
3   if(*DHCSR&1) {   // Detect C_DEBUGEN bit
      // debugger detected
5     return 1;
    } else {
7     return 0;
    }
9 }
```

In practice, we used this simple detection method in a CTF challenge by placing the detection routine inside a FreeRTOS thread to clear a secret key from RAM whenever a debug interface tries to connect. If the action is simple enough like in this example, it will complete before the core halts and protect the secret key. Note that this technique cannot be used on Cortex-M0 cores because DHCSR is not reachable from the CPU on this architecture.

Hardware breakpoints

Like their x86 cousins, ARM cores have two kinds of breakpoints: software and hardware. Hardware breakpoints use a dedicated core component, called the BreakPoint Unit (BPU) on Cortex M0 and M1, or the Flash Patch BreakPoint Unit (FPB) on Cortex M3 and later.

The BPU uses a control register `BP_CTRL` and up to four comparator registers `BP_COMPx`. If the PC register matches the value of one of the `BP_COMP` registers and the BPU is enabled, the core will halt the execution. By default, OpenOCD will enable the BPU when connecting to a Cortex-M0 core, it is therefore possible to look for this value in the same way as with the DHCSR register above:

```
1 uint8_t detect_debug(void){
    uint32_t *BP_CTRL=(uint32_t *) 0xE0002000;
3   if(*BP_CTRL&1) {      // detect ENABLE bit
      // debugger detected
5     return 1;
    } else {
7     return 0;
    }
9 }
```

The FPB replaces the BPU and has the same functionality and conveniently uses the same address for its control register `FP_CTRL` as for `BP_CTRL`, so the detection and breakpoint features work the same way. However, there is an added functionality called the Flash Patch, which allows to redirect the execution flow to a different path based on a comparator and a destination address. Instead of breaking when the PC register matches the comparator, the core will update the PC value with the value stored in a remap table located in RAM. The remap table is a pointer array, and if comparator x matches and is enabled, the xth entry of the table replaces the PC value.

In the following example, we use the FPB remap to call the `return_zero()` function instead of `return_one()`. This would produce a valid binary and headaches to any reverse engineer trying to understand what the code does.

```
1  uint8_t return_one(void)
   { return 1; }
3
   uint8_t return_zero(void)
5  { return 0; }

7  void* FP_REMAP_TABLE[6] = {
     (void *)&return_zero
9  };

11 void setup_fpb(void) {
     // Point FP_REMAP register to our remap table
13   uint32_t * FP_REMAP = (uint32_t *)0xE0002004;
     *FP_REMAP = (uint32_t)FP_REMAP_TABLE;
15
     // Setup the compatator
17   uint32_t * FP_COMP0 = (uint32_t *)0xE0002008;
     uint32_t comp_value = 0;
19   // Set comparison address
     comp_value |= (uint32_t)&return_one;
21   // Enable comparator
     comp_value |= 1;
23   *FP_COMP0 = comp_value;

25   // Enable FPB unit
     uint32_t * FP_CTRL = (uint32_t *)0xE0002000;
27   *FP_CTRL |= *FP_CTRL | 0b11;
   }
29
   int main(void) {
31   [...]
     setup_fpb();
33
     while (1) {
35     if (return_one()) {
         // This branch will NEVER execute
37     } else {
         // This branch will ALWAYS execute
39     }
     }
41 }
```

Another nice feature of the FPB remap for obfuscation is that OpenOCD resets all FPB registers when connecting to a target. This means that as soon as the debugger is connected to a target running the previous example, the core will execute the first branch instead of the second one, effectively hiding the correct code flow from unauthorized eyes.

Software breakpoints

Software breakpoints halt execution when the CPU executes a `bkpt` instruction which is really useful when debugging your firmware. The instruction also takes a byte-sized parameter to further help the developer manage multiple breakpoints.

An interesting property of the `bkpt` instruction is that if it is executed while the core has no debug enabled, it will generate a HardFault. As a developer, we can leverage this property within the firmware and create a dedicated Hardfault handler. The plan is to detect if the fault happened because of a `bkpt` instruction, restore the registers and resume execution to the next instruction.

Looking at the ARM documentation, we can find that a register contains information about the type of fault that happened. On Cortex-M0, it's the `DFSR` register and on Cortex-M3 and later the register is called the `HFSR` (Hard Fault Status Register). On both of these, a bit is set when the fault occured because of an untrapped debug event (ie. a `bkpt` instruction with no debugger): the `BKPT[1]` and `DEBUGEVT[31]` respectively.

Now that we know how to detect the debugger, we need to resume execution. Upon entering a fault, some registers are saved on the stack for further analysis. This process is automatically managed by the core, and the following registers are saved (from top to bottom): `r0`, `r1`, `r2`, `r3`, `r12`, `lr`, `pc`, `xPSR`

When entering the fault handler, the execution context changes to *handler mode*. It is possible to get back into *thread mode* by linking to a special address of `0xFFFFFFF9`, which coincidently is the value of the link register set when entering the fault handler. Jumping to that address will automatically restore the register values and resume execution.

The only thing left is to increment the saved PC value in the stack by 2 to point to the instruction following the `bkpt` instruction and resume execution. In the following example, we update a global variable containing the detection status.

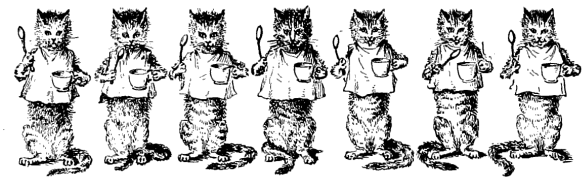
```
uint8_t DEBUGGER_DETECTED = 1;

void HardFault_Handler(void) {
  uint32_t *HFSR=(uint32_t *) 0xE000ED2C;
  if(HFSR & 0x80000000) { // DEBUGEVT bit
    // reset detection variable
    DEBUGGER_DETECTED = 0;
    asm(
    "push {r0}\n"
    "ldr r0, [sp, #28]\n"
    "add r0, r0, #2\n"   // increment saved pc
    "str r0, [sp, #28]\n"
    "pop {r0}\n"
    "bx lr\n"            // resume execution
    );
  } else {
    while(1){}           // other fault
  }
}

int main(void) {
  while(1) {
    DEBUGGER_DETECTED = 1;
    asm("bkpt 8\n");
    if(DEBUGGER_DETECTED) {
      // debugger is present
    } else {
      // debugger not present
    }
  }
}
```

Semihosting

Messing with reverse engineers and people trying to debug your firmware isn't enough? Let's take a look at another ARM debugging feature: semihosting.

Semihosting is a way for the target firmware to access data on the debugger side by using syscall-like operations like `open`, `read`, and `write`. It is typically used to allow functions like `printf` to be used in the firmware, with the output being printed in the debugger console on the host. It uses a clever mechanism to work. If the firmware halts on a `bkpt` instruction while being debugged, the debugger will fetch the argument to the `bkpt` instruction. If the argument value is `0xAB`, the debugger will fetch the operation to be performed in `r0`, and the arguments at a location pointed to by `r1`.

The following code implements semihosting to perform a `SYS_WRITE` operation (semihosting call 5) to the host's `stdout`, file descriptor 1.



```
void print_semihosting(char * data, size) {
  /* use SYS_WRITE to STDOUT */
  uint32_t args[3];
  args[0] = 1;          // FD 1 = STDOUT
  args[1] = (uint32_t)data;
  args[2] = size;
  asm(
  "mov r0, #5\n"  // Op #5 - SYS_WRITE
  "mov r1, %0\n"
  "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
}
```

The same applies to the other semihosting operations, but one in particular is interesting: `SYS_SYSTEM`. As the name implies, this operation asks the debugger to fetch a command from the target and pass it to the `system()` function on the host. It is therefore possible to use any if the debugging detection routines shown in this article to call this function if a debugger is detected. As a mandatory example, this function will spawn the `xcalc` binary on the debugger host:

```
void spawn_calc(void) {
  const char * cmd = "xcalc";
  uint32_t args[2];
  args[1] = (uint32_t)cmd;
  args[2] = 6;
  asm(
  "mov r0, #18\n" // Op #18 - SYS_SYSTEM
  "mov r1, %0\n"
  "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
}
```

Many variations of this trick exist, and can easily mess with the debugger host. Fortunately, semihosting is not enabled by default in OpenOCD. The command `arm semihosting enable` must be entered in OpenOCD's console to activate semihosting support.

Conclusion

ARM microcontrollers are wonderful devices packing lots of hidden gems like the ones I briefly presented to you today. There surely are more of them hidden deep in the documentation or in an obscure corner of a dumped firmware. I hope that this small introduction will trigger your curiosity and help you find other clever ways to practice firmware self-defence. Code is attached.[6]

[6] `git clone https://github.com/Baldanos/cortex-m-antidebug || unzip pocorgtfo21.pdf cortex-m-antidebug.zip`