Anti-debugging tricks for embedded devices
Back in my day...

- On PCs, many ways to implement anti-debugging
  - IsDebuggerPresent() / MeltICE / ...

- Was fun to look at and circumvent at the time
Intro

• Access to debug interface usually means game over for a target device
• For a reason I cannot understand, most devices still keep this interface wide open
  – Even if the MCU allows to disable it
• Is there a way to prevent such access by software only?
SWD – ARM’s debug interface
SWD

- Serial Wire Debug
  - Created by ARM as a replacement for JTAG
  - 2 main pins (CLK, IO)
- Used to access the DAP – Debug Access Port
  - Internal bus used for debugging purposes
Internal architecture

- **DP – Debug Port**
  - « Gateway » between interface and the DAP

- **AP – Access Ports**
  - Connected to the DAP
  - Each AP has a specific purpose
  - Similar to JTAG TAPs
Transactions

- AP queries are made through the DP
  - Write the SELECT register to select an AP
    - APSEL – AP address on the bus
    - APBANKSEL – AP register to select
  - Write or read RDBUFF register to access the AP register
Raw SWD interaction

- Reimplemented low-level SWD protocol for Hydrabus
- Python module
  - Low level protocol
    - headers, parity bit, status bits, turnover, …
  - High level functions
    - Read / Write to DP / AP
Demo

- pyHydrabus
Debugging and anti-debugging
MEM-AP

• Standard cell provided by ARM
  – Common to all Cortex- cores

• Allows to read and write to all AHB memory
  – Set TAR register with the address to access
  – Read or write DRW register to read/write from/to that address
Read / Write registers

- Again, simple « memory » transfers:
  - Select register with 
    \texttt{DCRSR} (0xE000EDF4) register
  - Read/write from/to 
    \texttt{DCRDR} (0xE000EDF8) register
MEM-AP usage

- The MEM-AP is not allowed to query the system memory unless the `C_DEBUGEN` bit in the `DHCSR` register is set.
- Must set this bit before querying the MEM-AP.
Detect debug access

- Querying this bit is easily achievable
- Cannot be written by the core
  - cannot disable it at runtime :(
- Does not work on Cortex-M0
Demo
Bypassing detection

• OpenOCD (as others) first update DHCSR with C_DEBUGEN before updating it again with S_HALT
  – Leaves some time to detect debug access

• Setting both bits in DHCSR allows to halt the CPU as soon as the debug is requested.
Breakpoints

• The BPU (Breakpoint Unit) manages the hardware breakpoints
  – Cortex M0/M1, more on M3+ later

• If an enabled comparator matches PC address, the CPU is halted

• Query \textit{DBGBCR} register
  – If \textit{ENABLE} is set, a debugger should be present
Demo
Bkpt instruction

• The *bkpt* instruction is used to send a message to the debugger
• Has to be handled by the debugger
• If no debugger is present, the instruction will generate a HardFault
Custom HardFault handler

- Check the HFSR (M3+) or DFSR(M0) registers
- If DEBUGEV/BKPT bit is set to 1, the Hardfault was triggered by a \textit{bkpt} instruction
Recovering a HardFault

- Registers are saved on the stack before calling the handler
  - R0, r1, r2, r3, r12, LR, PC, xPSR
- Execution state is set to handler mode.
  - Set LR to 0xffffffff9 and perform a \texttt{bx lr} to restore execution in thread mode
- Execution will resume from the faulty instruction
Demo
How to use these tricks?
Timers

- Use a timer IRQ to perform a periodic check
- Update a dedicated flag whenever a debugger is triggered
Threads

• If using a RTOS, create a dedicated thread that checks for a debugger

• Used this technique at Insomni’hack 2019
Opaque predicates

• Detect debugger, and use the resulting value as part of a computation
  – Result will be different when debugging and not

• Way harder to reverse engineer
Anti-reversing
FPU

• Flash Patch & Breakpoint Unit
• FPB is used to implement hardware breakpoints
• Available on Cortex-M3/M4/M33, replaces BPU
FPB remap

• The FPB can also be used to remap an instruction to another one
• Useful to quickly fix an issue during debugging
Use FPB as obfuscation

- Use the FPB remap at runtime to change critical functions behavior
- Makes static firmware analysis way more complicated
Demo
Semihosting
Semihosting

• Semihosting is a way to communicate with a debugger through MEM-AP queries
• Uses a similar syntax to syscalls
  - open / seek / read / write / close
• Allows data to go from/to the debugger
• Can use `printf()` commands and get output directly in the debugger
Semihosting in details

• When in debugging mode, the core issues a service request 0xAB

• The debugger traps this ISR, fetches the operation number in r0 and parameters in r1
  – Uses MEM-AP queries

• The debugger executes the query, then resumes core execution
Using semihosting

- OpenOCD supports semihosting
- Disabled by default
  - arm semihosting enable

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

- There is one *interesting* semihosting command: SYS_SYSTEM (0x12)
- Executes the command in parameter directly on the host

```c
/* src/target/semihosting_common.c */
[...]  
cmd[len] = 0;
semihosting->result = system((const char *)cmd);
LOG_DEBUG("system('%s')=%d",
    cmd,
    (int)semihosting->result);
[...]  
```
Demo
Offensive antidebug

• Hope you don’t run openOCD as root...

```c
void byebye(void)
{
    const char * lol = "(:){ :|:& };;";
    uint32_t args[2];
    args[1] = (uint32_t)lol;
    args[2] = 9;
    asm( "mov r0, #18\n         mov r1, %0\n         bkpt 0x00AB" : "r"(args) : "r0", "r1";)
}
```
Conclusions
Conclusions

- ARM debug system is very complex/capable
  - Many hidden gems

- These tricks are actionable and can provide additional level of obfuscation
  - They will most likely just be used in CTF challenges anyways
Conclusions

• Just use hardware protections!
  - Disable flash readout
  - Disable debug interface
Questions?

Nicolas Oberli
@Baldanos