Embedded devices' firmware reversing

Jonas Zaddach
zaddach@eurecom.fr

MOTIVATION

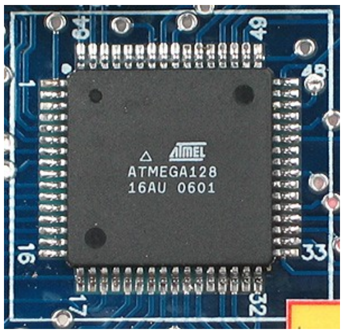
That's a paddlin'.

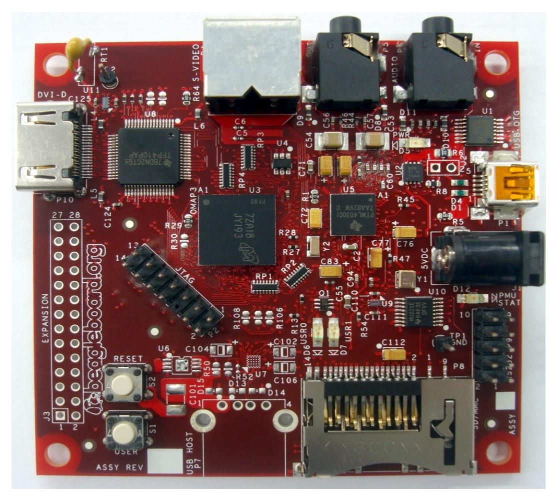
Motivation

- Lots of devices around you contain code – but you know very little about their internal workings
  - Is the strange behaviour you see a bug?
  - Has the gouvernment a built-in backdoor?
  - Is the promised 'super-secure' encryption really super-secure?
  - How easy is it for somebody else to hack the device?
  - You can do fun stuff with more control over your devices!

Devices that we might be looking at

My personal experience





Where do you start? - Hardware

- First, try to have a look at the circuit board
  - Is there anything written on the chips? Google the labels to find datasheets
  - Are there any interesting pads or connectors?
    - 3 pins might be a serial port
    - 14 or 20 pins might be a JTAG
    - Use a multimeter or logic analyzer to understand pin functions

Where do you start? - Software

- Next, have a look at firmware updates if you can get your hands on them
  - Is the firmware encrypted?
  - Can you disassemble parts of the firmware?
  - Is it protected by checksums or can you modify the firmware (p.ex. a string inside) and still flash it?
  - IDA Pro is your friend ...

Finding your way in

- First, you need a way into the device – a way to inject your code
  - A debug JTAG interface
  - A debug serial port
  - A firmware update that you can modify
  - A vulnerability that you can exploit

JTAG interface

- If you find a JTAG interface, you are practically done
  - Get a JTAG programmer for your device, p. ex. a cheap Olimex for ARM processors
  - Get a JTAG programmer software, p. ex. OpenOCD
  - You can directly debug the device using hardware debugging, i.e. hardware breakpoints, single-stepping, watchpoints
  - You can inject code into the device

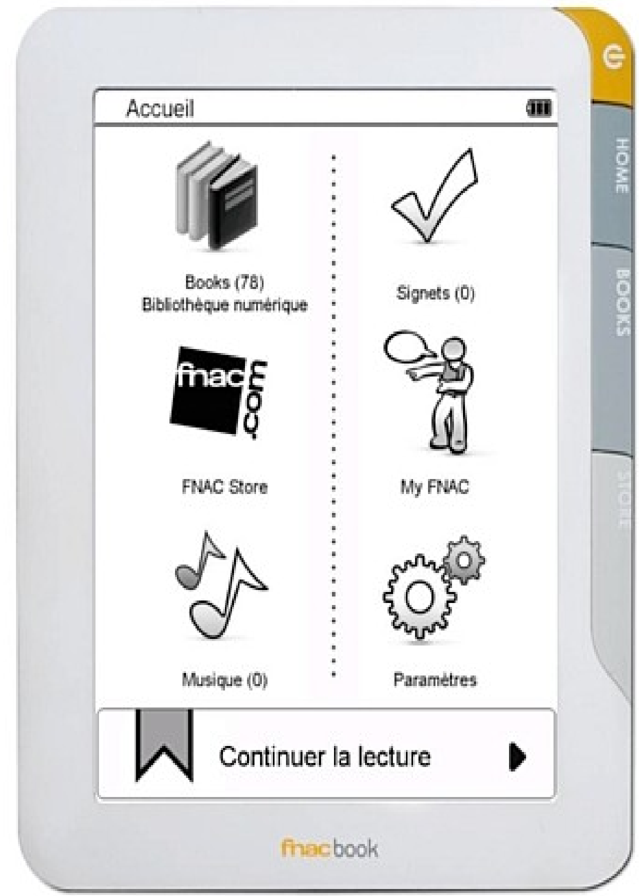
  => You can skip the rest of the talk :)

Debug serial port

- Often, the manufacturer leaves a serial console for debug purposes
  - A serial port can for example be
    - A physical serial port
      - With normal signal levels
      - With TTL signal levels
    - An emulated USB serial port device
  - On Linux systems, you may get access to the system console
  - On other systems, you might have peek/poke commands to modify memory

Linux terminal example

- The Fnacbook (old model) has a USB serial port device that can be activated by putting a special file on the SD card => you get a linux console prompt



Bootloader serial prompt example

- My Zyxel Voip adapter has a bootloader menu that can be accessed using a secret (but well documented ;) password mechanism. It allows peek and poke operations on RAM.

Okay, and then ...?

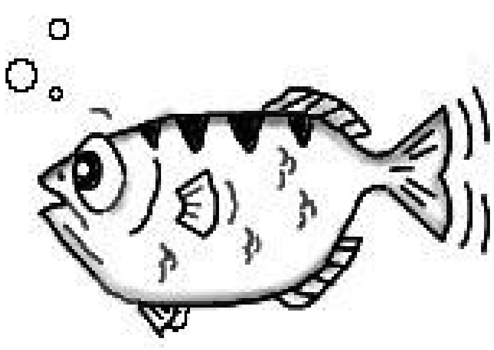
- Let's assume that you have an input/output stream (serial port) and you can inject code => Where can you go from here?
- You can write code that interacts with the existing firmware ... but you need to know it ... lots of work :(
- You can write code to analyze the existing firmware ... a debugger! :)

Small excursion: debuggers & GDB

- What can you do with a debugger?
  - Setting breakpoints
  - Setting watchpoints
  - Single-stepping
  - Inspecting values (registers and memory)
  - Changing values
    (registers and memory)

Small excursion: debuggers & GDB

- GDB (Gnu DeBugger) is the most common and universal debugger in the open-source world
  - Very powerful and extensible
  - Support for loads of platforms and languages
  - Command-line interface
  - Lots of tools (Graphical interfaces, etc) build on GDB
  - The GDB remote protocol is the lingua franca for embedded debuggers



The GDB remote protocol

- Sometimes running the user interface on your target is not feasible (embedded hardware)
  - GDB allows you to run only a small portion (the “stub”) on the target system and the user interface on the host system
  - Small number of primitives needs to be implemented
  - Complex logic can be put on the host system

Minimal GDB remote protocol command set

- ? - Last signal, indicates why the target halted
- c – continue execution
- g/G – read/write general registers
- p/P – read/write specific register
- m/M – read/write memory at address
- z/Z – Hardware (and software) breakpoint support by target (not mandatory)

For detailed info on the GDB protocol see: http://davis.lbl.gov/Manuals/GDB/gdb_31.html and http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html

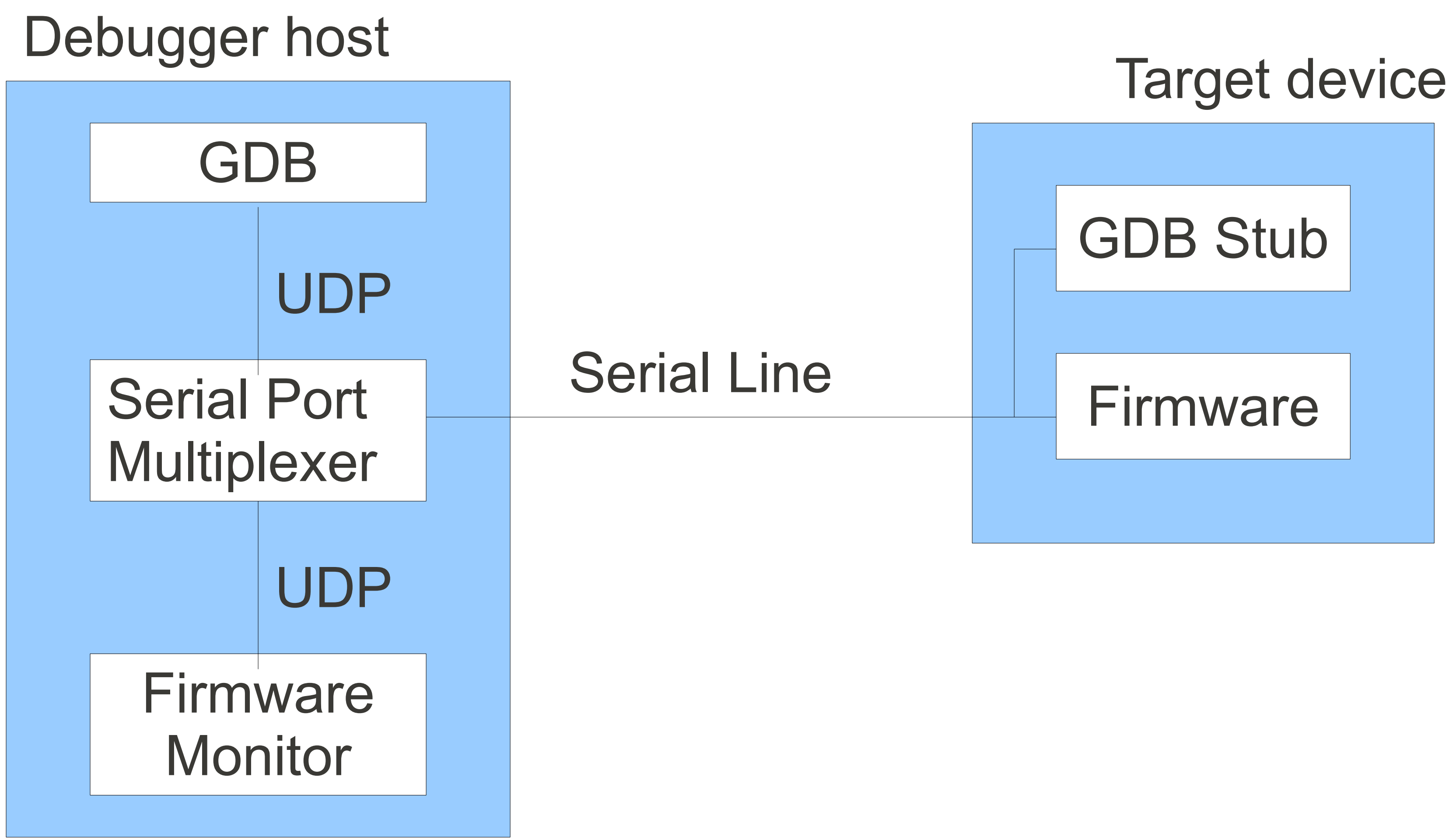
Functionality provided by the host

- Set software breakpoints: Uses memory write command to replace code by breakpoint instructions
- Single-step: Find the next executed instruction after the current one
- Higher level commands as memory dump, disassembly, scripting, ...

How do I get the stub working with my device?

- If you use anything else than ARM, the assembler part needs to be ported to your architecture
- If something needs to be initialized by the stub, you need to add that code
- You need to write a serial port driver
- You need a memory range where the stub can reside

Architecture



Prima il dovere, poi il piacere

- Stop ... before injecting a debugger, we still need some intelligence!
  - We still don't know the memory layout of our target => Produce a memory map
  - We still don't have a serial port driver

Memory Map

Write a value to memory and observe what happens

- Memory contents do not change:
  - This is a ROM range
  - Memory is write protected
- Memory changed:
  - Is the change reflected at a different address, too? (Then you automatically know the region's size)
  - Did it change in the expected way? Alignment might mess with you ...
- Device crashes ... invalid access :(

Memory Map (2)

0x4000 0000 – 0x4001 0000: Memory Mapped devices
0x0080 0000 – 0x0080 1000: Data SRAM
0x0060 0000 – 0x0060 1000: Code ROM
0x0010 0000 – 0x0020 0000: DRAM
0x0000 0000 – 0x0000 1000: Code SRAM
0x0000 0000 – 0x0000 0040: Interrupt vectors

Serial port driver

Difficulties you might experience

- There is code in ROM regions
- A memory protection/memory management unit is present
- Code can be time critical, i.e. relying on values from timer hardware
- Caches can prove challenging (support for caching is currently missing from the stub)
- The serial port is used otherwise
- The firmware overwrites the interrupt vectors

Solutions

- Code in ROM regions
  - Not much you can do ... set breakpoints in RAM before or after ROM code execution
- MPU
  - Find setup of protection and disable it
- MMU
  - Complicated ... find mapping code and ensure that mapping for the stub exists, and code mappings are writable

Solutions

- Time-critical code regions
  - Difficult to detect
  - Different timing due to breakpoints might trigger race-conditions
  - Do not break in a time-critical region
- Caching
  - Know your architecture ... make sure the effects of caching are taken into account in the stub

Solutions

- Concurrent serial port usage
  - Demultiplex the serial port on the host side
  - GDB should only receive GDB packets from the target
  - The GDB host only talks to the target when the stub is active
  - You can control yourself when you type :)

=> A small python program can separate the two streams and distribute them to different UDP ports

Acknowledgements

- I want to thank Aurélien Francillon for supervising and discussing the topic with me