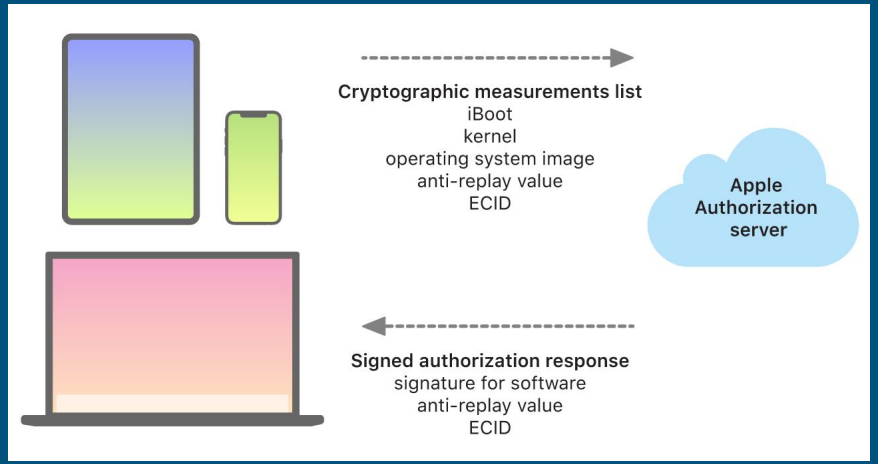
Trusted Firmware In Embedded Systems

Isaac Garfield

Security Background

Most software is verified by comparing a known hash against the software's computed hash

- "Apple digitally signs all software updates"
- Software Update verifies Apple's digital signature before installation."



[1] Verifying Apple Software Authenticity

Security Background 2

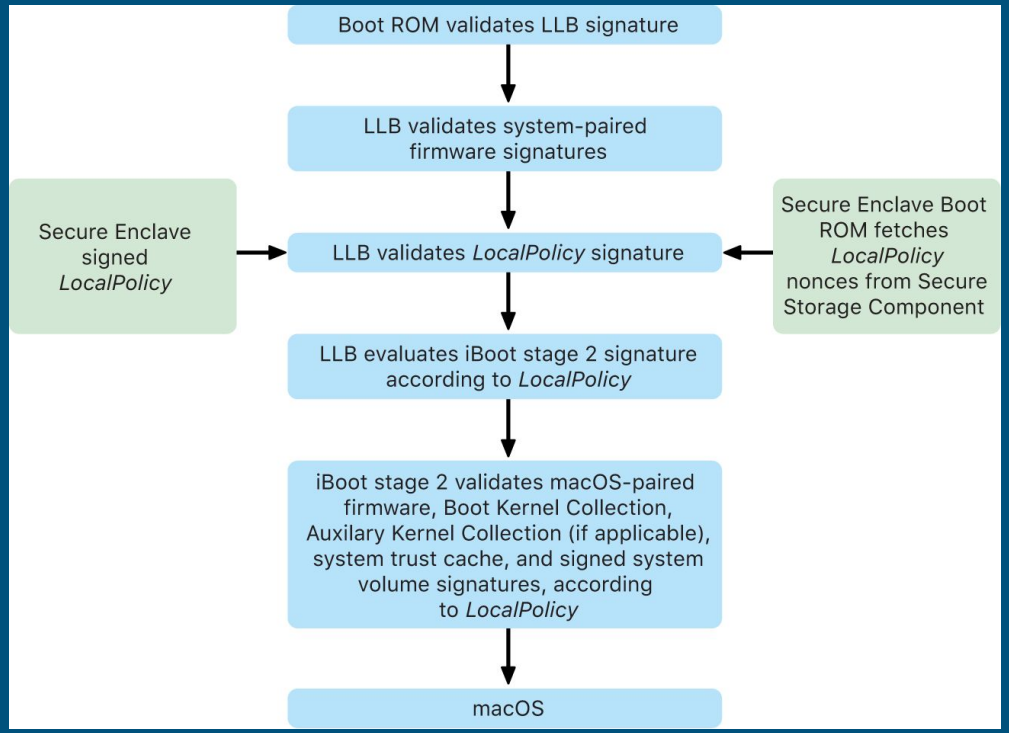
Each stage in the bootloader verifies the signature of the next stage. Only the BootROM is “unverified”.

The BootROM is the root of trust, and is configured at the fab

“System paired firmware” is loaded

After LocalPolicy validation, control flow diverges, and different security levels are applied

“macOS-paired firmware” is loaded



Apple Silicon Mac Boot Process

Board Background

The NUCLEO-L433RC-P is a development board for the STM32L433 series chips

These chips are Low Power Microcontrollers with a 32 bit ARM-Cortex M4 chip, 64KB SRAM, 256KB Flash

STMicroelectronics offers a configuration tool and code generator which provides basic project structure

This project structure includes the Hardware Abstraction Layer (C), the main function (C), a linker script (ld file), and a startup script (ARM asm)

Software Background

Hardware Abstraction Layer (C), the main function (C), a linker script (ld file), and a startup script (ARM asm)

1) When you press reset on the STM32, you start at address 0x8001c9c
2) 0x8001c9c normally contains the Reset_Handler, which puts the processor in a known state
3) Reset_Handler calls our main() function
4) From main(), we call the HAL provided functions to initialize peripherals like UART, Counters, System Clocks (kinda), and LEDs.
5) From there we run whatever code we want - like blink an LED

Design Part 1

- C String Literals are stored in `.rodata`
- In `int main()` we can declare `char* my_str = “char”`
- We can get a pointer to .rodata by
  - Declaring `_my_var` in our linker script at the start of .rodata
  - Declaring `extern uint32_t _my_var` in C
- We can declare `uint32_t word1 = 0x72616863`
- We can then compare `word1 == _my_var`

Design Part 2

- The linker script defines `ENTRY(Reset_Handler)`
- If we add `verify_boot` to the existing `startup.s` we can modify our linker script with `ENTRY(verify_boot)`
- If we add `QUAD(0x000d0a79656b796d)` in our linker script data section (`.data`), `“mykey\n\r”` will be stored in memory
- We can also find in the linker script the address of `.data` , then in `verify_boot` load that address into a register, and compare the expected value in `.data` with immediate values defined in `startup.s - verify_boot`
- If `verify_boot` detects a match, we can jump to `Reset_Handler` and startup normally, otherwise, we can jump to `Infinite_Loop` until we reset the board

Work Accomplished

Got a signature checker working in C (String literal + .rodata read)

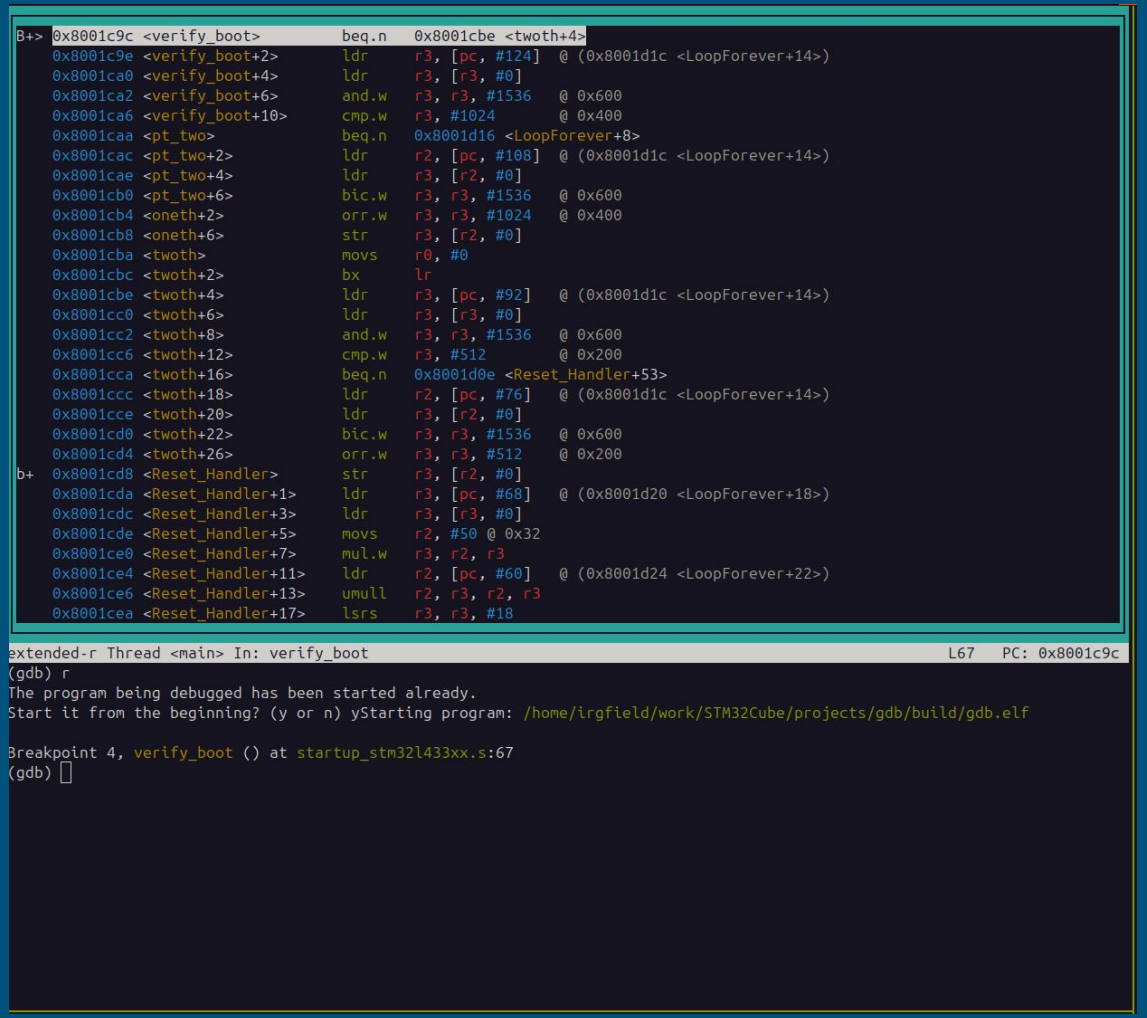
Got a signature checker working in C + linker (Store data with QUAD, .data read)

Built an assembly signature checker (Store data with QUAD, .data read)

- But I had issues getting this working. The immediates or symbols I thought should be loaded were loaded incorrectly. GDB skipped so many lines I couldn't figure out what happened.

But my assembly signature checker was called

Work Accomplished



Work Accomplished

```
70  void check_str() {
 1      uint32_t v1 = 0x6e676953;
 2      uint32_t v2 = 0x000a6465;
 3
 4      extern const char __rodata_start;
 5      uint32_t* rd_ptr = &__rodata_start;
 6
 7      extern const char _sdata;
 8      uint32_t* sdata_ptr = &_sdata;
 9
10      if (v1 == rd_ptr[0]) {
11        HAL_UART_Transmit(&huart2, "1 T\n\r", 5, 0xffffffff);
12      } else {
13        HAL_UART_Transmit(&huart2, "1 F\n\r", 5, 0xffffffff);
14      }
15      if (v2 == rd_ptr[1]) {
16        HAL_UART_Transmit(&huart2, "2 T\n\r", 5, 0xffffffff);
17      } else {
18        HAL_UART_Transmit(&huart2, "2 F\n\r", 5, 0xffffffff);
19      }
20  }
21
22
23  /**
24    * @brief  The application entry point.
25    * @retval int
26    */
27  int main(void)
28  {
29    uint8_t sign[] = "Signed\n";
30    check_str();
```

```
 1  /* Entry Point */
52  ENTRY(verify_boot)
                                                    52,18         16%

154   /* Initialized data sections goes into RAM, load LMA copy after code */
 1  PROVIDE (_sdata = .);
 2    .data :
 3    {
 4      . = ALIGN(8);
 5      _sdata = .;        /* create a global symbol at data start */
 6      QUAD(0x000a64656e676943);
 7      *(.data)           /* .data sections */
 8      *(.data*)          /* .data* sections */
 9      *(.RamFunc)        /* .RamFunc sections */
10      *(.RamFunc*)       /* .RamFunc* sections */
11
12      . = ALIGN(8);
13      _edata = .;        /* define a global symbol at data end */
14    } >RAM AT> FLASH
```

```
63      .section    .text.verify_boot
 1      .weak   verify_boot
 2      .type   verify_boot, %function
 3  verify_boot:
 4    ldr r3, =_sdata
 5  /*  ldr r3, =0x00000020 /* Just pull the variable from linker */
 6  /*  ldr r3, =0x20000000 */
 7
 8    ldr r1, =0x6e676953
 9    ldr r2, [r3]
10    cmp r1, r2
11    bne Infinite_Loop
12    ldr r1, =0x000a6465
13    ldr r2, [r3]
14    cmp r1, r2
15    bne Infinite_Loop
16
17    b Reset_Handler
18
```

Lessons Learned

The reset button does very little. It sets the pc to the Reset_Handler, which does everything else.

The Linker defines the ENTRY of program, which the program uses on startup.

The Reset_Handler is defined and written in assembly for each STM32 product line.

ARM is not one instruction set, it's a whole family with many variants.

GDB has a hard time inserting breakpoints before main is called, with mixed success.

QUAD and LONG statements in a linker allow canaries to be placed at arbitrary locations.

Make doesn't detect LDSCRIPT changes and wouldn't automatically update code, so everything had to be cleaned and re-run.

Assembly is hard, and designing your own project is a challenge. But a good challenge.

Conclusion

Bootloaders are hard and as the root of trust, their security is paramount

Lots of energy and effort is put into protecting the bootloader

The internet allows us to check against a verified source, but some things need to be verified on device, luckily Apple good hardware

Good debugging is as important as good code writing