FATAL FURY ON ESP32:

TIME TO RELEASE HW EXPLOITS

LimitedResults

Black Hat Europe 2019
2-5 December 2019, London

$ whoami

- Limited
  - By the Time, $$$, my Skills…
- Results
  - www.LimitedResults.com
- Offensive Side
  - Focus on HW, Low-Level Vulns…
- No Affiliation
- Time to play!

POWER ON

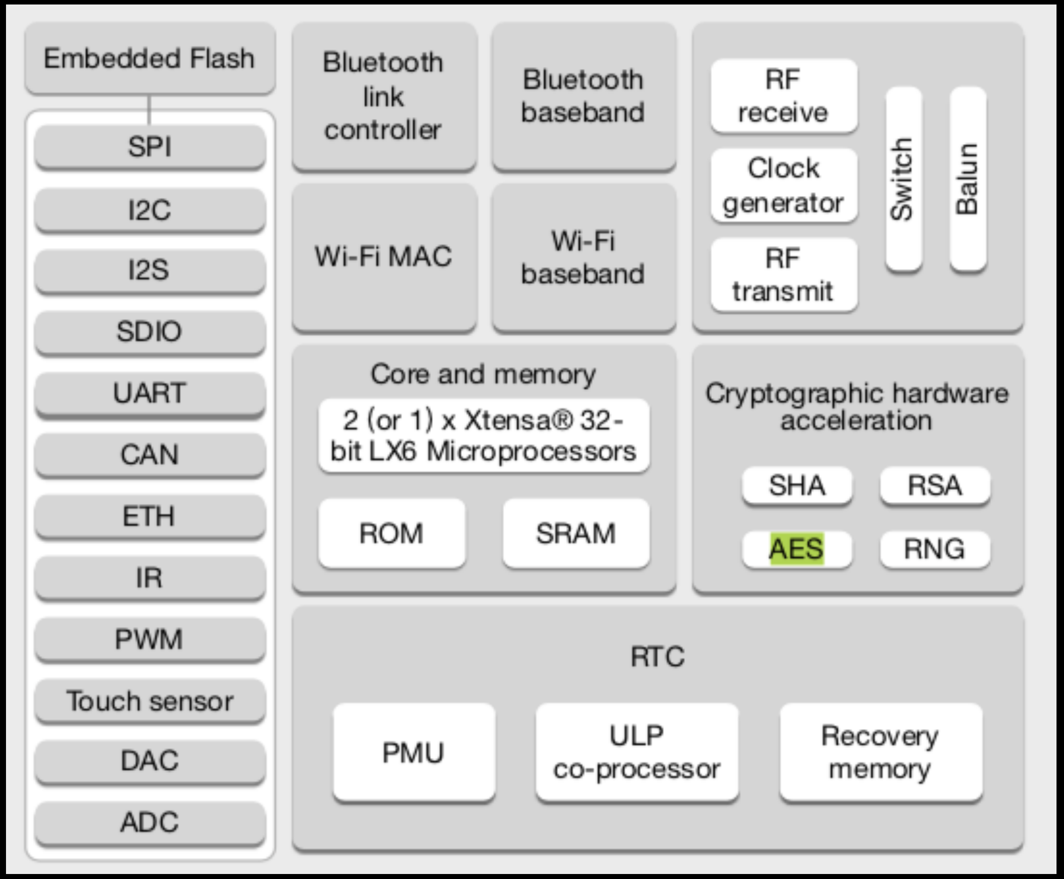
INTRODUCTION

The Entry Point

- Last April, I decide to ~~break~~ investigate into the ESP32
  - System-on-Chip (SoC) released in 2016 by Espressif
  - Widely-deployed (> 100M of devices) [1]
  - Wireless MCU/SoC Market leader
  - Claim to have ‘State-of-the-Art’ Security
  - 12 years-longevity commitment
- General Use
  - IoT
  - Wireless peripheral

The target

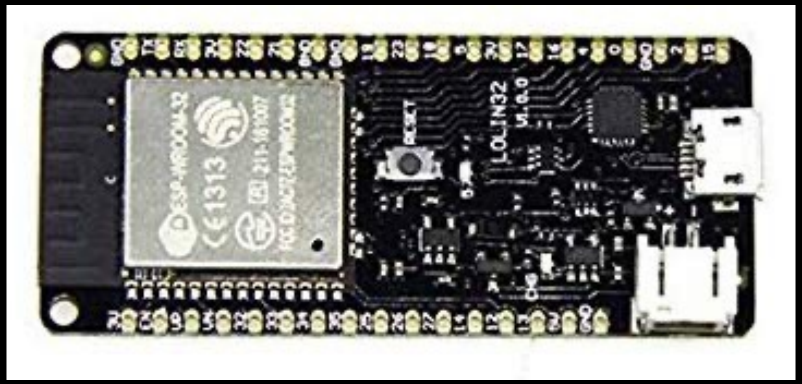
- ESP32
  - Techno 40nm node
  - QFN 6*6, 48 pins
- Overview
  - Wi-FI (2.4GHz) & BT v4.2
  - Ultra Low-Power Xtensa Dual-Core LX6
    - up to 240MHz
    - ROM, SRAM, no CPU caches
  - GPIOs, Touch sensor, ADC…
  - 4 SPI, 3 UART, Ethernet…No USB



ESP32 Form Factor

- ESP32 SiP module (ESP32-WROOM-32)
  - Easy to integrate in any design
  - Flash storage 4MB
  - FCC certified

- ESP32 Dev-Kit (Lolin ESP32)
  - Micro-USB
    - Power
    - ttyUSB0 port
    - Pin headers



- Limited Cost = 15$

ESP32 Software

- Esp-idf Dev. Framework on Github
  - xtensa-esp32-elf toolchain
  - Set of Python Tools (esptool)
- Good Quality of Documentation
  - Datasheet and TRM available [2]
- Arduino core supported
  - I don’t like pre-compiled libraries, I don’t use it
- Official Amazon AWS IoT Platform
  - FreeRTOS, Mongoose OS…

Agenda Today

- Focus on Built-in Security
  - Just Grep the Datasheet
- Four Points
  - Crypto HW accelerator
  - Secure Boot
  - Flash Encryption
  - OTP
- Time to pwn!

1.4.4 Security

- Secure boot
- Flash encryption
- 1024-bit OTP, up to 768-bit for customers
- Cryptographic hardware acceleration:

OPTIONS MENU

SETTINGS

The Limited Plan

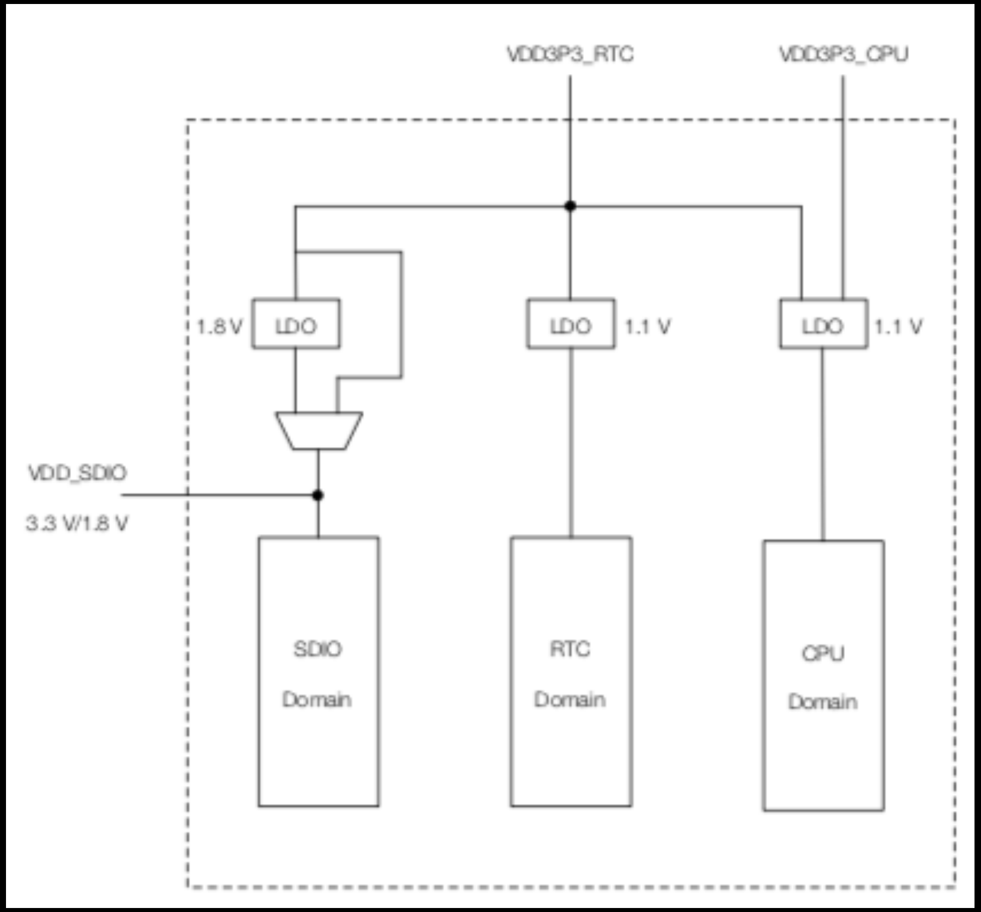
- The Context
  - 3 months to investigate (spare time)
- My Objective
  - Break one by one the Security Features
    - Physical Access Required (plausible attack scenario today)
- So, I will probably use HW Techniques
  - Fault Injection, Side Channel maybe?
  - Micro-soldering, PCB modification
  - Reverse
  - and Code Review ☺

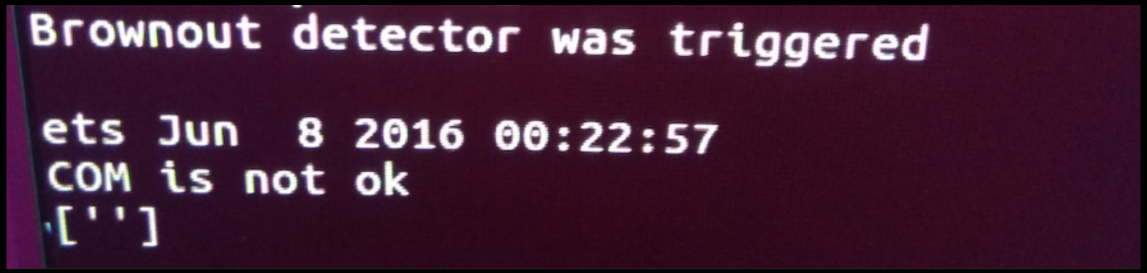
Voltage Fault Injection

- aka Voltage glitching
  - Well-known, still efficient and Low-cost FI technique nowadays
  - Public ressources about voltage glitching [3][4][…]
- Perturb the Power Supply to induce a fault during critical SW/HW operations
- Expected effects
  - Skip instruction
  - Data/Code modification
- Unexpected effects
  - Difficult to predict/understand faults in complex CPU architecture (due to Cache effects, Pipeline…)

Power domains inside ESP32

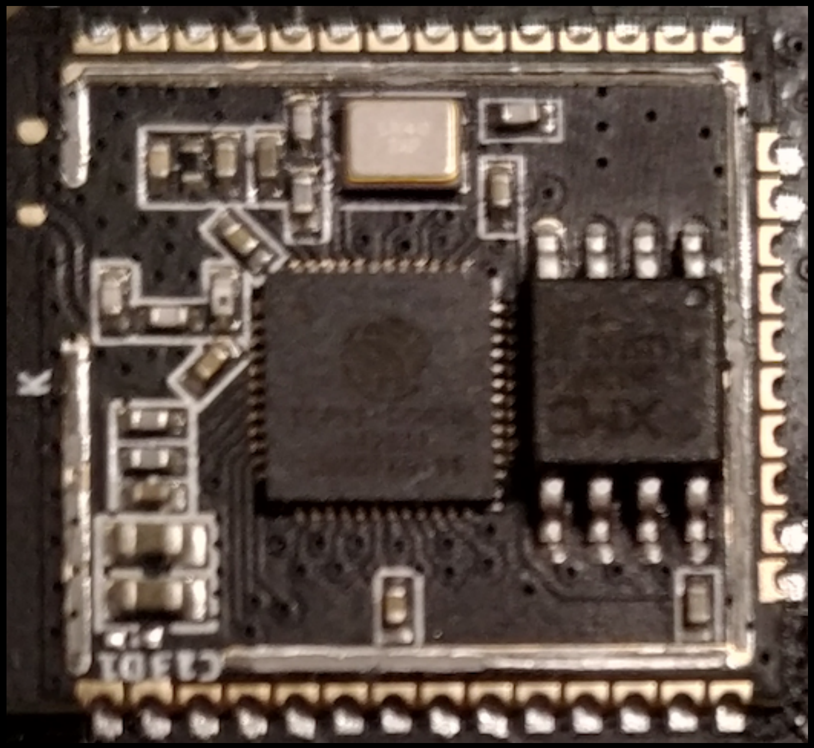
- 3 separate Power domains
- CPU domain shares two Power Signals
  - VDD3P3_CPU && VDD3P3_RTC
- Low Drop-out regulators (LDO)
  - Stabilize internal voltages
    - Filter effect against glitches?
- Brownout Detector (BOD)
  - « If the BOD detects a voltage drop, it will trigger a signal shutdown and even send a message on UART »
  - Able to detect glitches?
- BoD only effective on VDD_RTC
- So, I will Glitch on VDD3P3_CPU

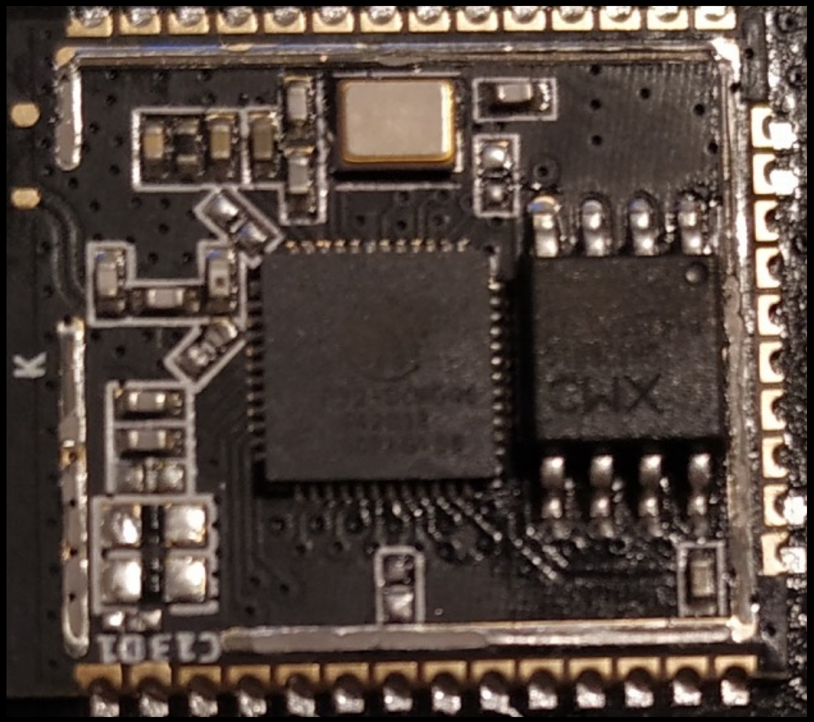




Target Preparation

- ESP-WROOM-32 Module
  - Shield is removed
- No silkscreen but Schematic available
- I remove Capacitors connected to VDD_CPU and VDD_RTC



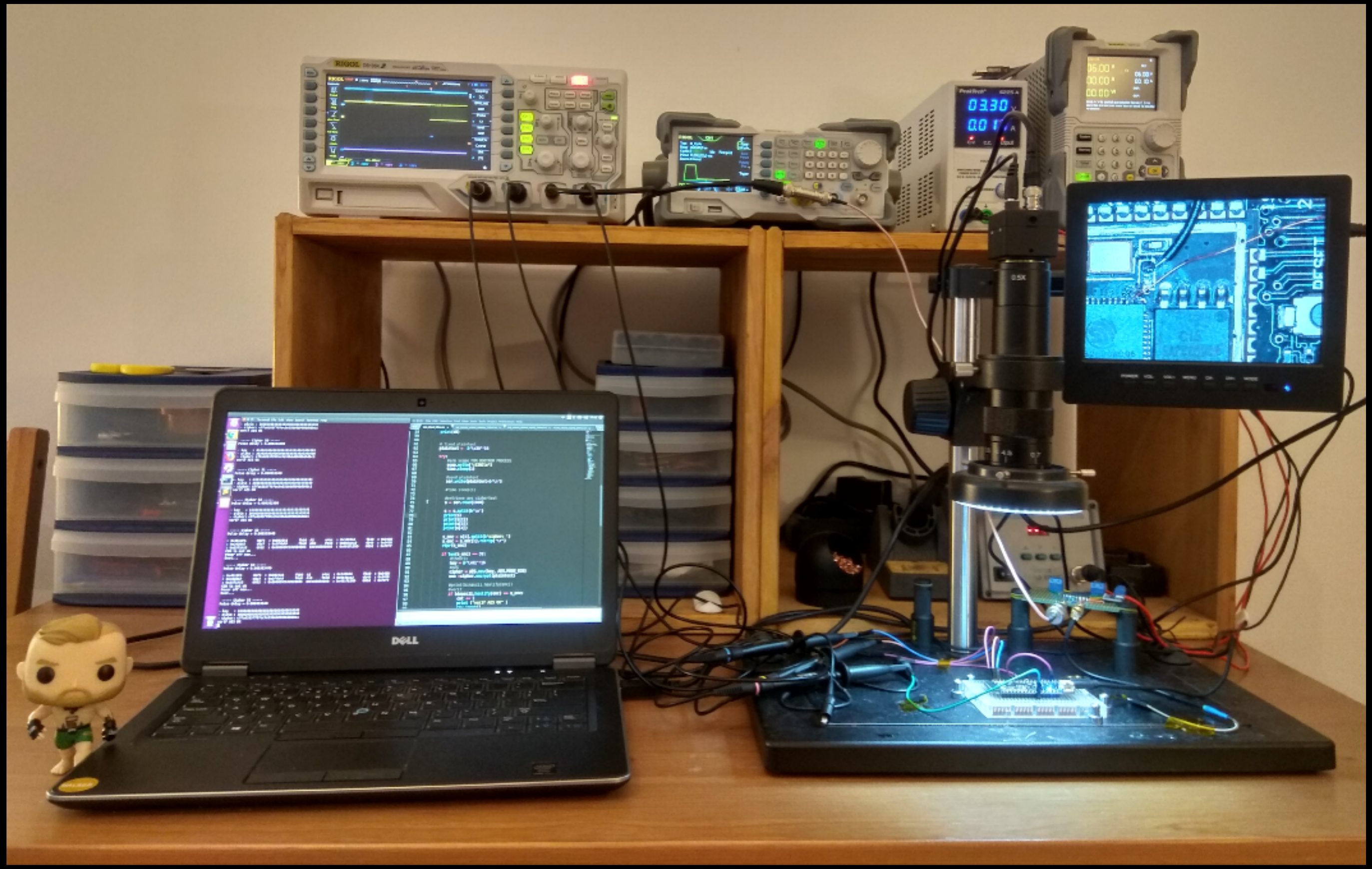


PCB Modification

- Three steps
  - Exposing the VDD_CPU trace (Pin 37)
  - Cutting the trace
  - Soldering the glitch output to VDD_CPU and GND

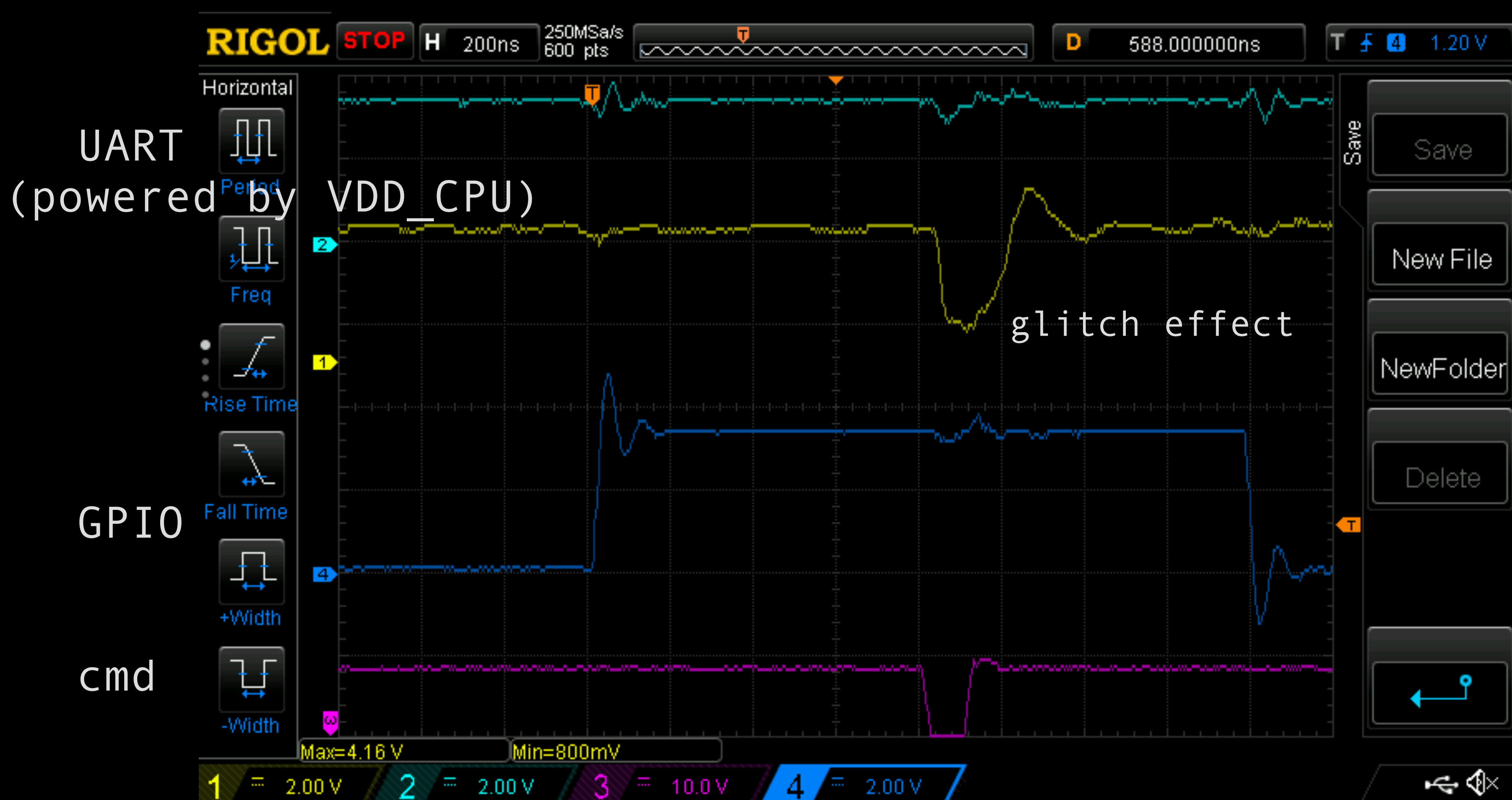
HW Setup

- Home-made Glitcher (10$)
  - Based on MAX4619
  - Add passive components, SMA connectors
- Synchronised by Scope
- Triggered by Signal Generator
  - USB commands to set parameters
    - Delay
    - Width
    - Voltage
- Python scripts for full-control
  - Can run during days…



Voltage Glitching effect

- Effect looks good



LEVEL 1

THE CRYPTO-CORE

Crypto-Core/ Crypto-Accelerator

- Just a peripheral to speed-up the computation
  - AES, SHA, RSA…
- Why is it interesting to pwn?
  - Espressif Crypto-Lib
  - HW accel. used by default in MBedTLS
    - MBedTLS is the ARM crypto-library (all IoT are using it)
- My Goal
  - Focus on the CPU/Crypto interface (crypto-driver)
    - Do not expect to find ‘pure’ Software Vulns
  - Looking for vulns triggered by Fault Injection
- It is Time for Code Review

Design Weakness

- AES operation
    - Datasheet

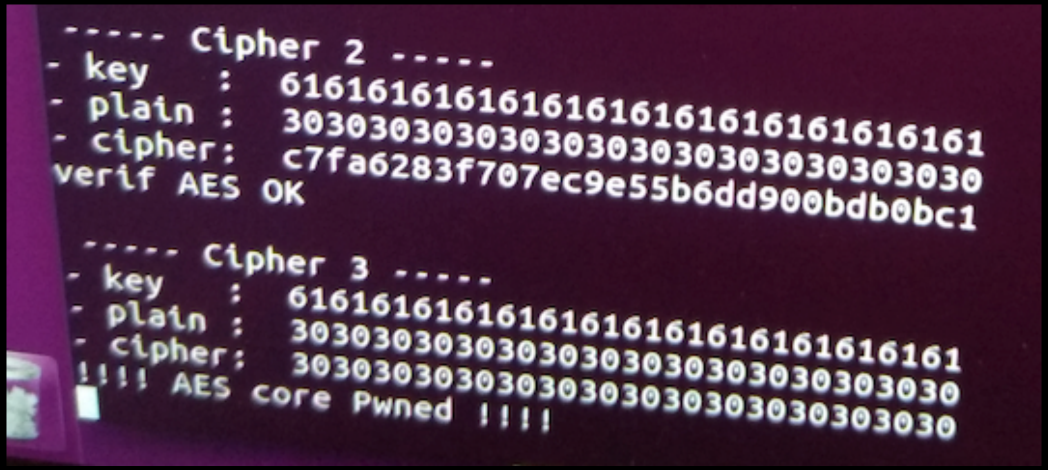
Single Operation

1. Initialize AES_MODE_REG, AES_KEY_*n*_REG, AES_TEXT_*m*_REG and AES_ENDIAN_REG.
2. Write 1 to AES_START_REG.
3. Wait until AES_IDLE_REG reads 1.
4. Read results from AES_TEXT_*m*_REG.

- Design Weakness
    - AES_TEXT_m_REG registers used to store plaintext and also ciphertext
- Encrypt-In-Place can be risky
    - If something goes wrong during AES call, pretty sure I can retrieve the plaintext
- Pretty cool & simple to exploit as first PoC

Vuln n*1 = AES Bypass

- Previous Weakness is confirmed
- Multiple spots to trigger
  - AES start
  - The while condition
  - The last For loop
- PoC
  - Output = Input




```
 * Call only while holding esp_aes_acquire_hardware().
 *v4.0-dev-141-g106dc0590-dirty
static inline void esp_aes_block(const void *input, void *output)
{
    const uint32_t *input_words = (const uint32_t *)input;
    uint32_t *output_words = (uint32_t *)output;
    uint32_t *mem_block = (uint32_t *)AES_TEXT_BASE;

    for(int i = 0; i < 4; i++) {
        mem_block[i] = input_words[i];
    }

    DPORT_REG_WRITE(AES_START_REG, 1);

    DPORT_STALL_OTHER_CPU_START();
    {
        while (_DPORT_REG_READ(AES_IDLE_REG) != 1) { }
        for (int i = 0; i < 4; i++) {
            output_words[i] = mem_block[i];
        }
    }
    DPORT_STALL_OTHER_CPU_END();
}
```

Vuln n*2 = AES SetKey

- Vuln to trigger
  - Unprotected for loop to load the key into the crypto-core
- PoC
  - Key zeroized
  - Persistent key value until the next setkey()
  - Nice for attacking AES Cipher Block Chaining Mode

```c
static inline void esp_aes_setkey_hardware( esp_aes_context *ctx, int mode)
{
    const uint32_t MODE_DECRYPT_BIT = 4;
    unsigned mode_reg_base = (mode == ESP_AES_ENCRYPT) ? 0 : MODE_DECRYPT_BIT;

    for (int i = 0; i < ctx->key_bytes/4; ++i) {
        DPORT_REG_WRITE(AES_KEY_BASE + i * 4, *(((uint32_t *)ctx->key) + i));
    }

    DPORT_REG_WRITE(AES_MODE_REG, mode_reg_base + ((ctx->key_bytes / 8) - 2));
}
```

```
- key   : 61616161616161616161616161616161
- plain : 30303030303030303030303030303030
- cipher: e08682be5f2b18a6e8437a15b110d418
!!!! Set key Pwned !!!!
```

```
>>> from Crypto.Cipher import AES
>>>
>>> aes = AES.new(b'\x00' * 0x10, AES.MODE_ECB)
>>> cipher = aes.encrypt(b'0' * 0x10)
>>> print(''.join('{:02x}'.format(x) for x in cipher))
e08682be5f2b18a6e8437a15b110d418
```

Crypto-Core Conclusion

- Crypto-core does not improve security
- Six Vulns with PoCs in AES and SHA
  - Espressif HwCrypto in esp-idf 4.0 (patched since)
  - ARM MbedTLS v2.13.1 (patched?)
- Resp. disclosure
  - No answer from Espressif & ARM during 1 month ☹
  - Silent Patch attempt ☹
  - BugBounty Program from ARM MBedTLS is Fake ☹
- I ~~am~~ was a little bit in a FURY
- …ready to pwn harder

LEVEL 2

SECURE BOOT

Role of Secure Boot

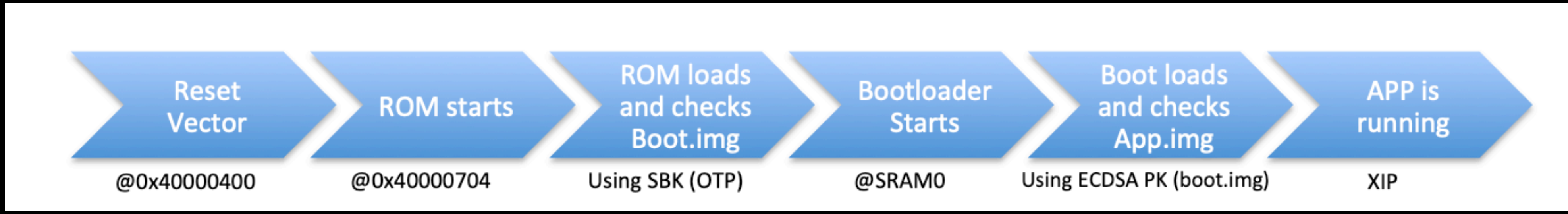
- Protector of FW Authenticity
- Avoid FW modification
  - Easy to flash malicious Firmware in SPI Flash
  - CRC? Not sufficient sorry…
- It will Create a Chain of Trust
  - BootROM to Bootloader until the App
- It Guarantees the code running on the device is Genuine
  - Will not boot if images are not properly signed

Sec. Boot during Production

- Secure Boot Key (SBK)
  - SBK burned into E-Fuses BLK2
  - This SBK cannot be readout or modified (R/W protected)
  - Used by bootROM to perform AES-256 ECB
- ECDSA key pair
  - Created by the App developer
  - Priv. Key used to sign the App, Pub. Key integrated to bootloader.img
- The Bootloader Signature
  - 192 bytes header = 128 bytes of random + 64 bytes digest
    - Digest = SHA-512(AES-256((bootloader.bin + ECDSA PK), SBK))
  - Random at 0x0 in Flash Memory layout, digest at 0x80

Sec. Boot on the Field

- Boot process



- Verification Mechanisms
    - BootROM (Stage 0)
        - Compute Digest with SBK and compare with 64-bytes Digest at 0x80
    - ECDSA verification by the Bootloader (Stage 1)
        - Micro-ECC is used
- I will Focus on Stage 0
    - Signature based on Symmetric Crypto
    - SBK = AES-Key used to sign the bootloader (CRITICAL ASSET) stored in E-Fuses, R/W protected

Set the Secure Boot

- Can be done automatically by ESP-IDF Framework…
- But I prefer to do it manually
    - Burn the Secure Boot Key into BLK2
        - $ espefuse.py burn_key secure_boot ./secure-bootloader-key-256.bin
    - Burn the ABS_DONE fuse to activate the sec boot
        - $ espefuse.py burn_efuse ABS_DONE_0
- E-Fuses Map
    - espefuse.py summary
- Look JTAG fuse ☺

```
Security fuses:
FLASH_CRYPT_CNT        Flash encryption mode counter                    = 0 R/W (0x0)
FLASH_CRYPT_CONFIG     Flash encryption config (key tweak bits)         = 0 R/W (0x0)
CONSOLE_DEBUG_DISABLE  Disable ROM BASIC interpreter fallback           = 1 R/W (0x1)
ABS_DONE_0             secure boot enabled for bootloader               = 1 R/W (0x1)
ABS_DONE_1             secure boot abstract 1 locked                    = 0 R/W (0x0)
JTAG_DISABLE           Disable JTAG                                     = 0 R/W (0x0)
DISABLE_DL_ENCRYPT     Disable flash encryption in UART bootloader      = 0 R/W (0x0)
DISABLE_DL_DECRYPT     Disable flash decryption in UART bootloader      = 0 R/W (0x0)
DISABLE_DL_CACHE       Disable flash cache in UART bootloader           = 0 R/W (0x0)
BLK1                   Flash encryption key
  = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
BLK2                   Secure boot key
  = ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? -/-
BLK3                   Variable Block 3
  = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
```

Secure boot in Action

- Signed Code (using SBK)

```
void app_main()
{
    while(1)
    {
    printf("Hello from SEC boot K1 !\n");
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

- make flash, then it runs

```
ets Jun  8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8556
load:0x40078000,len:12064
load:0x40080400,len:7088
entry 0x400807a0
D (88) bootloader_flash: mmu set block paddr=0x00000000 (was 0xffffffff)
I (38) boot: ESP-IDF v4.0-dev-667-gda13efc-dirty 2nd stage bootloader
...
I (487) cpu_start: Pro cpu start user code
I (169) cpu_start: Starting scheduler on PRO CPU.
Hello from Sec boot K1 !
Hello from Sec boot K1 !
```

- Unsigned Code (no Key)

```
void app_main()
{
    while(1)
    {
    printf("Sec boot pwned by LimitedResults!\n");
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

- Flash it then Fail

```
ets Jun  8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:3476
load:0x40078000,len:0
load:0x40078000,len:13740
secure boot check fail
ets_main.c 371
ets Jun  8 2016 00:22:57
```

- Stuck in stage0, perfect

Bypass the Sec.Boot

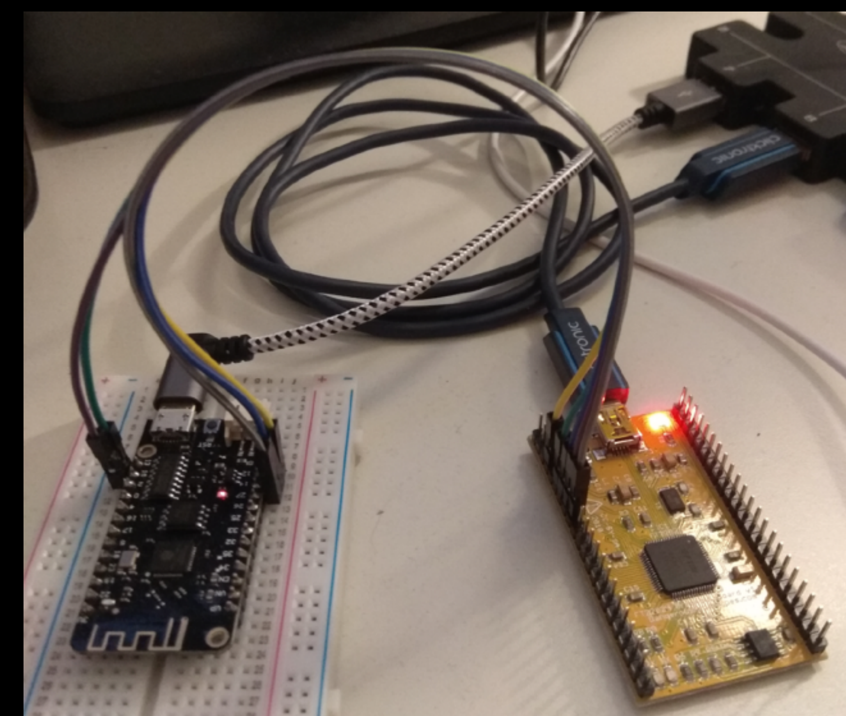
- Why?
  - to have code exec
- How?
  - Force ESP32 to execute my unsigned bootloader to load my unsigned app
- Focus on BootROM
  - Always Nice to exploit BootROM vulns, and always difficult to Fix BootROM vulns
- So, I need to reverse the bootROM
- But first, I need to dump it…

Dump the BootROM

- Memory map

Category	Target	Start Address	End Address	Size
Embedded Memory	Internal ROM 0	0x4000_0000	0x4005_FFFF	384 KB
	Internal ROM 1	0x3FF9_0000	0x3FF9_FFFF	64 KB
	Internal SRAM 0	0x4007_0000	0x4009_FFFF	192 KB
	Internal SRAM 1	0x3FFE_0000	0x3FFF_FFFF	128 KB
		0x400A_0000	0x400B_FFFF	
	Internal SRAM 2	0x3FFA_E000	0x3FFD_FFFF	200 KB
	RTC FAST Memory	0x3FF8_0000	0x3FF8_1FFF	8 KB
		0x400C_0000	0x400C_1FFF	
	RTC SLOW Memory	0x5000_0000	0x5000_1FFF	8 KB

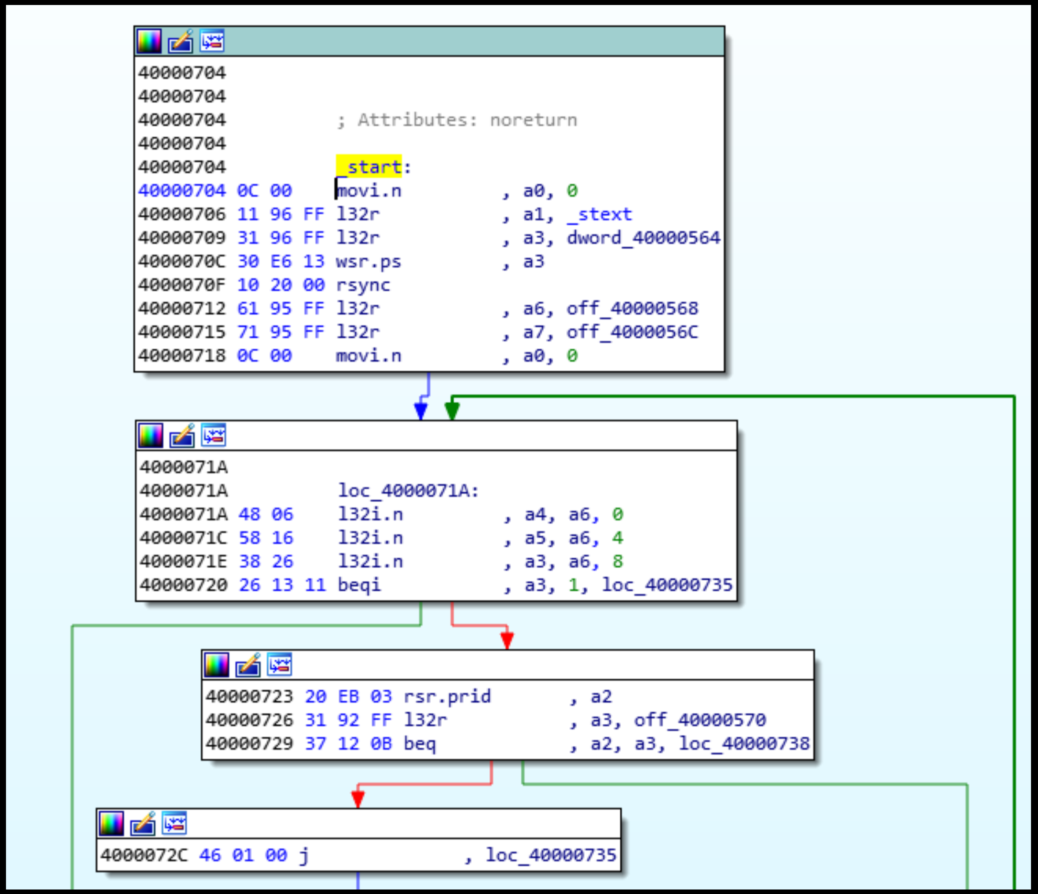
- Remember I didn't burn JTAG DISABLE E-Fuse?
  - FT2232H board (20$)
  - OpenOCD + xtensa-esp32-gdb
- Full Debug Access
  - Reset Vector 0x40000400
- BootROM dumped



```
(gdb) target remote :3333
Remote debugging using :3333
0x40000400 in ?? ()
(gdb)
```

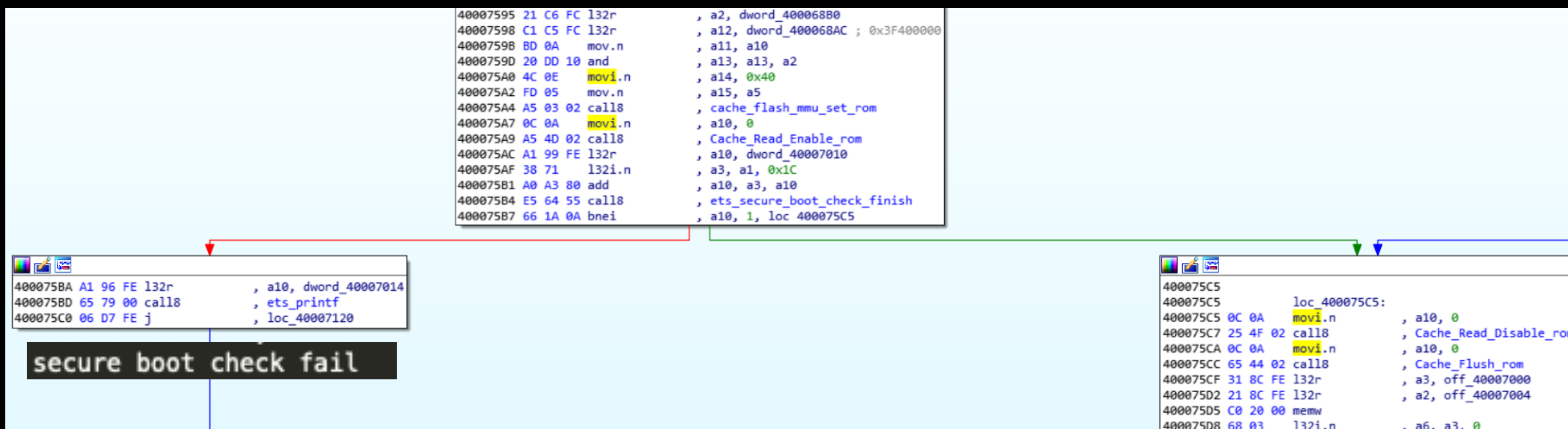
BootROM Reverse

- Xtensa is ‘exotic’ arch
  - registers windowing, lengths of instr…
  - ISA [5]
- IDA
  - ida-xtensa plugin from @themadinventor
- Secure_boot.h
  - List all the ROM functions (deprecated since…)
- Call a friend to check my mess
  - @wiskitki
- At the end, not perfect but doable
  - _start at 0x40000704 (as expected)



The BootROM Vuln

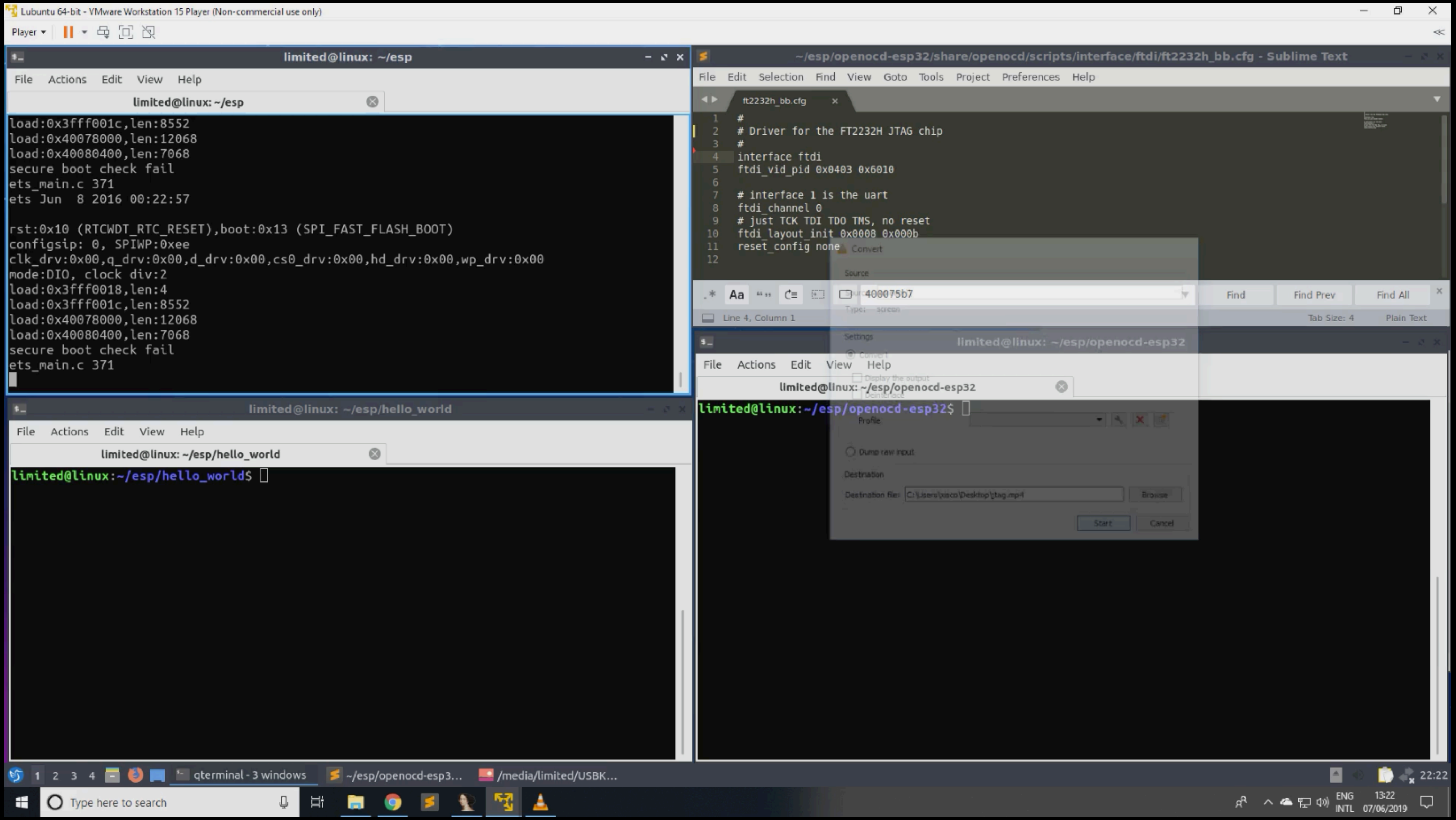
- After ets_secure_boot_check_finish()



- Bnei (Branch if not equal immediate)
  - Depends on a10 Register storing sec_boot_check_finish() retvalue
- I want PC jump to 0x400075C5 to execute the bootloader

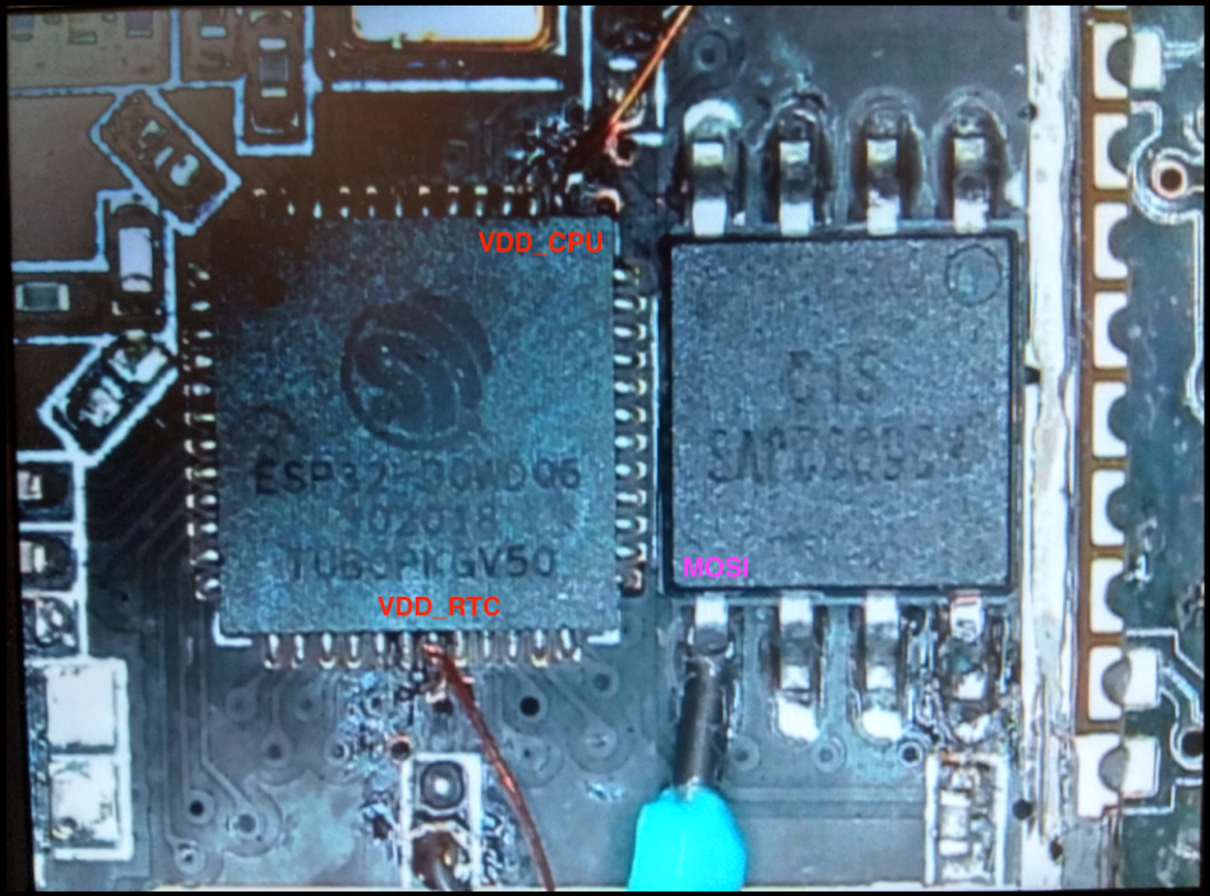
Jtag Exploit Validation

- Set a10 register = 0 via JTAG to bypass secboot



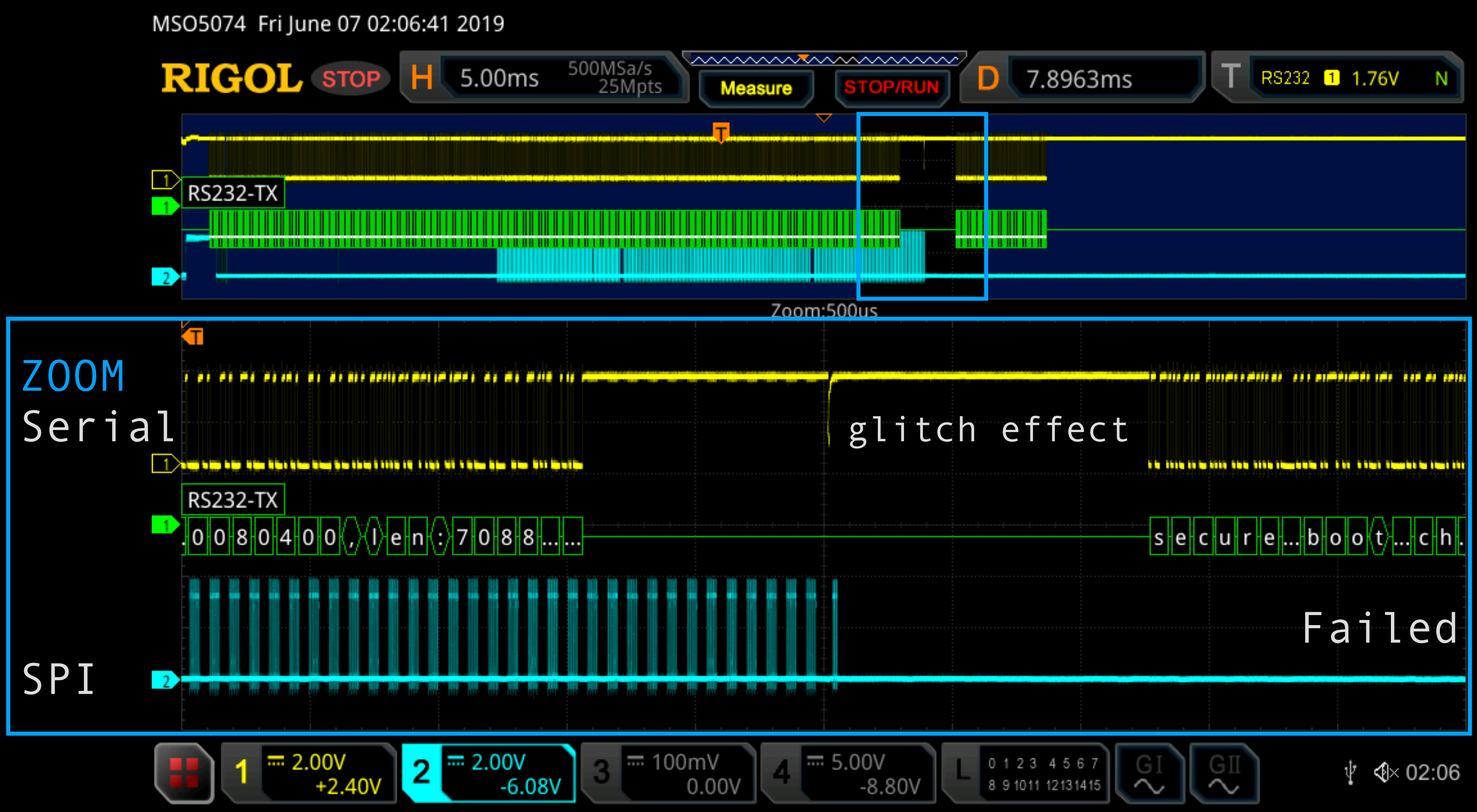
Time to Pwn (for Real)

- Real Life
  - JTAG is disabled
  - I could not find a way to exploit this Vuln by SW
- So, Fault Injection is my only way here
  - Simultaneous glitch on VDD_CPU && VDD_RTC
  - SPI MOSI is probed to have a timing information



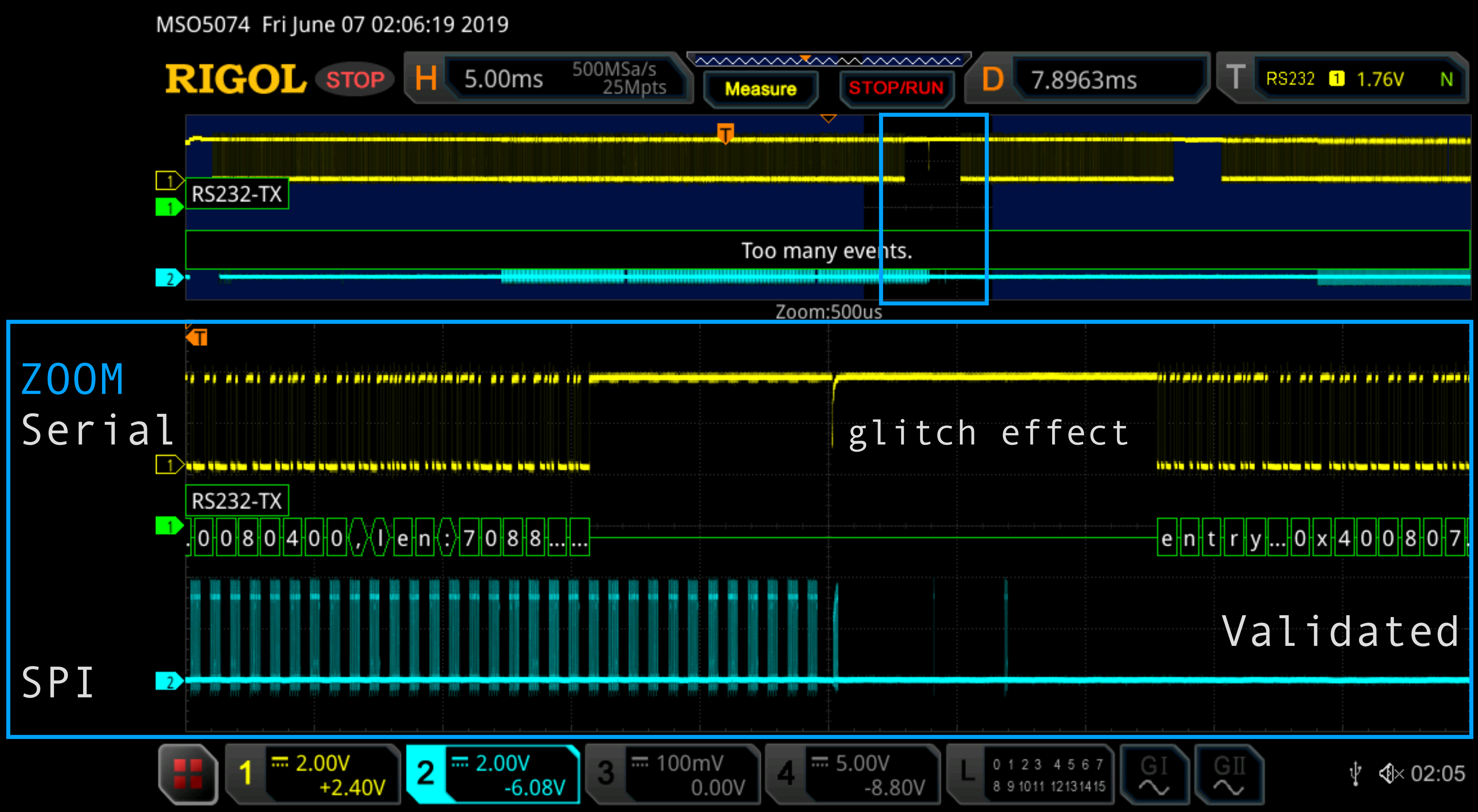
FI attempt during Boot

- Previous BootROM Reverse is helpful
  - to determine Fault injection Timing



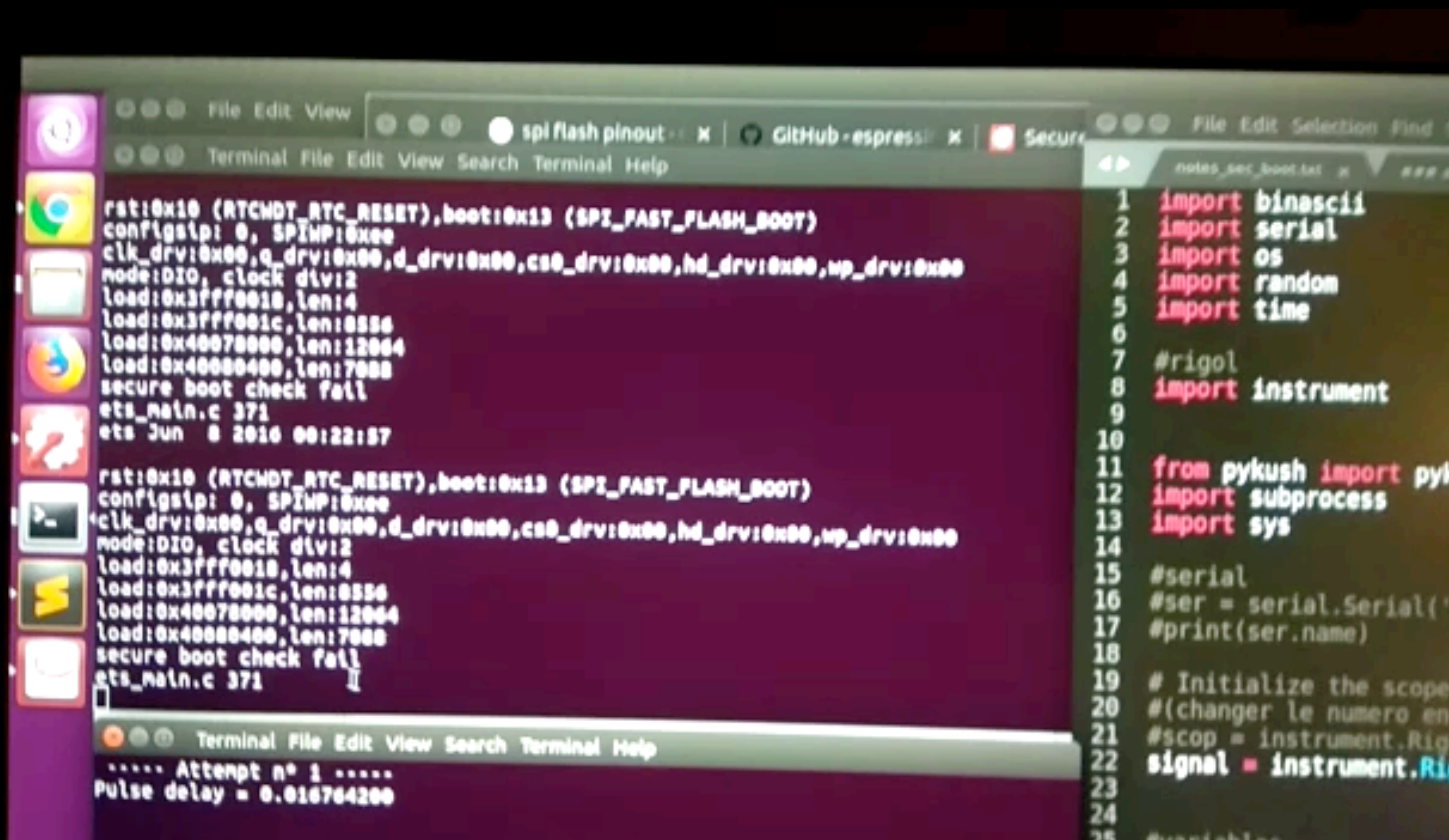
Successful Sec.Boot Bypass

- CPU is jumping to the entry point, Bootloader is executed



PoC Secure Boot

- Sorry for the tilt



Secure Boot Conclusion

- Secure Boot Bypass exploit
  - bootROM Vuln triggered by Fault Injection
  - Not persistent if Reset occurs
  - No way to Fix this without ROM revision
- Resp. disclosure
  - PoC sent on June 4, Posted on September 1
  - Security Advisory on Sept. 2
    - CVE-2019-15894 (requested by Vendor)
    - Patched by Flash Encryption always enabled
    - A security lab, called Riscure, found the same vuln
- Job done

LEVEL 3

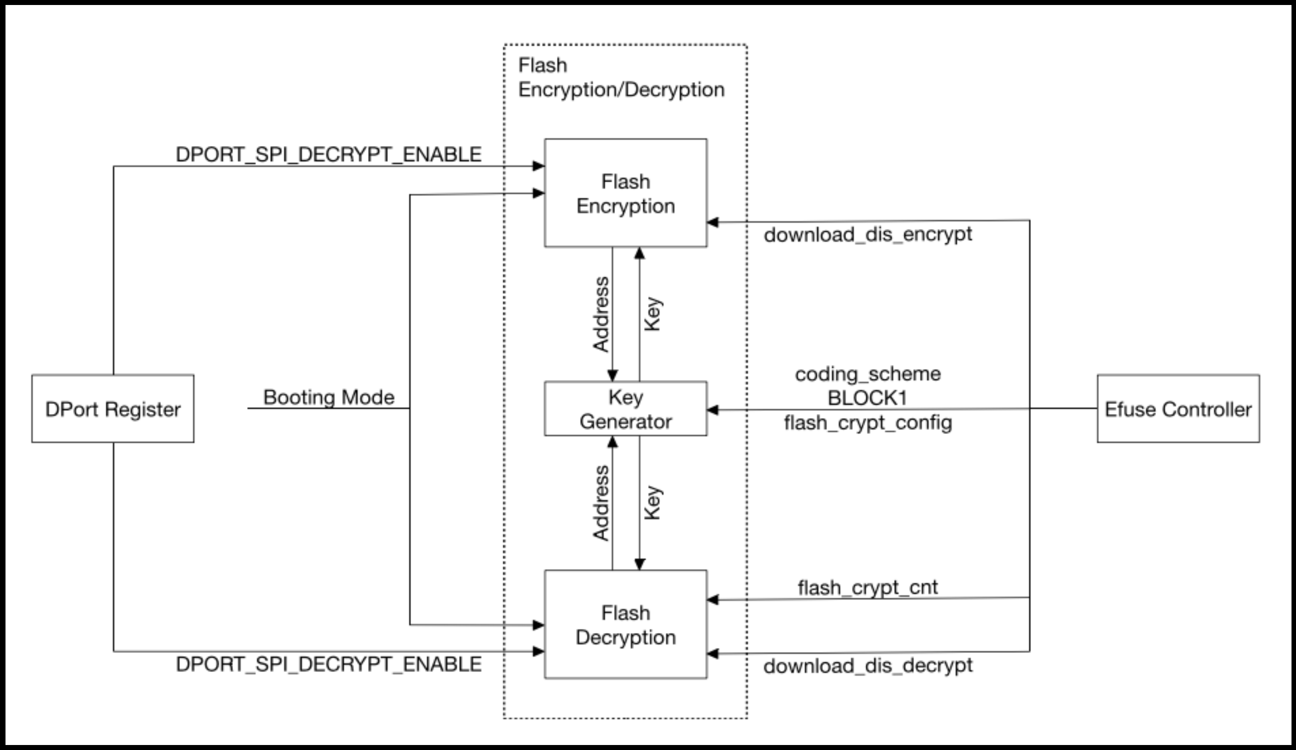
FLASH ENCRYPTION

Role of Flash Encryption

- Protector of FW Confidentiality
  - Protect against binary Reverse
- Without FE, it is easy to extract sensitive data
  - LIFX Wi-Fi lightbulb [6]
- Firmware Encryption more and more present Today
- Espressif recommends Secure Boot + Flash Encryption for maximum Security

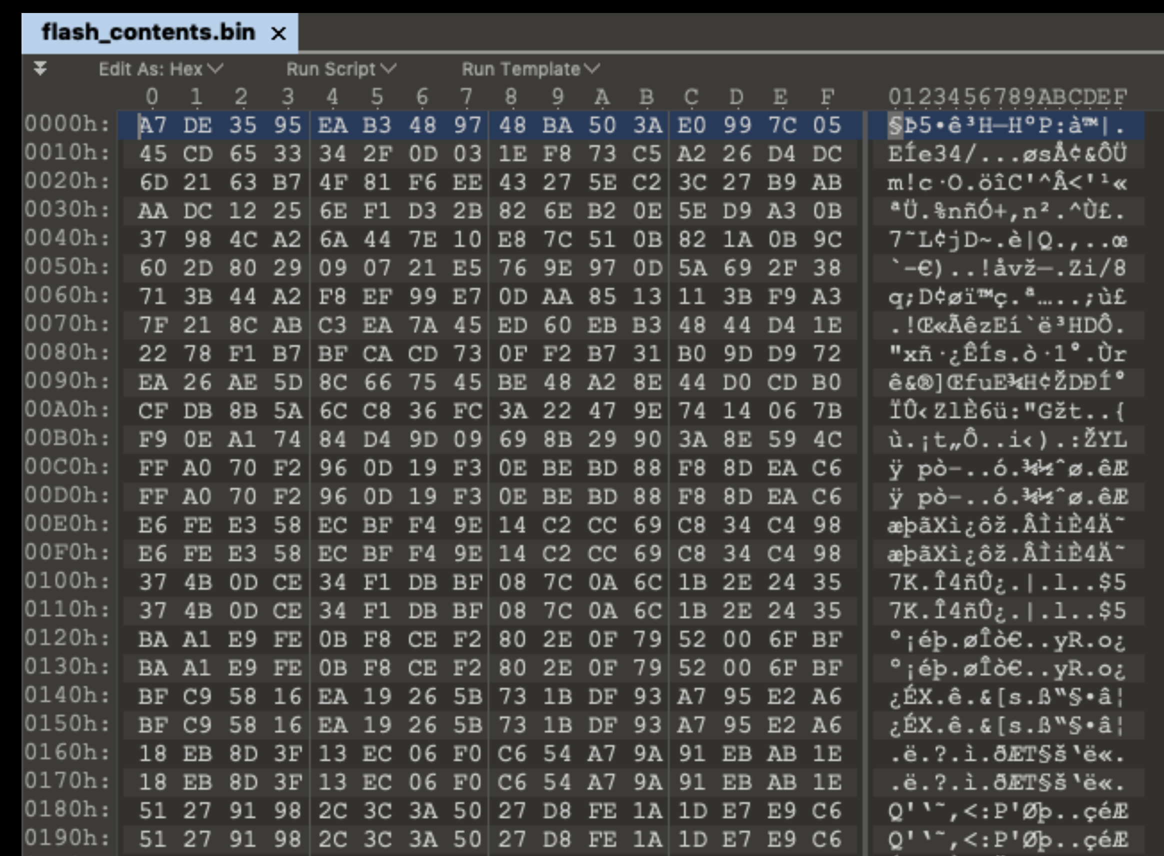
Flash Encryption Review

- HW Enc./Dec. Block in Flash Memory Controller
  - Fetch Key from E-Fuses and other parameters
  - Decrypt/Encrypt I/D into a Cache
  - SW cannot access
- Flash Encryption Key (FEK)
  - AES-Key used to decrypt the FW
    - Stored in E-Fuses BLK1 (R/W protected)
    - CRITICAL ASSET (of course)



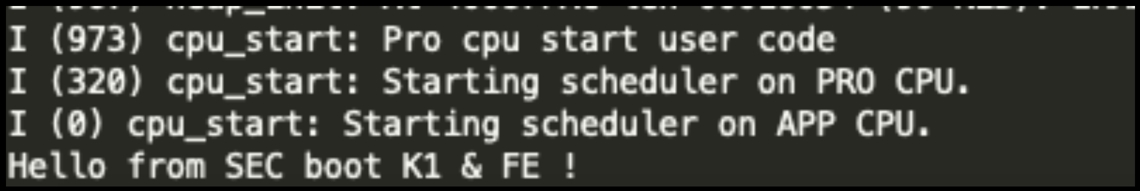
Set the Flash Encryption

- Burn the FEK into BLK1
  - $ espefuse.py --port /dev/ttyUSB0 burn_key flash_encryption my_flash_encryption_key.bin
- Activate the Flash Encryption
  - $ $ espefuse.py burn_efuse FLASH_CRYPT_CONFIG 0xf
  - $ espefuse.py burn_efuse FLASH_CRYPT_CNT
- Flash encrypted FW into ESP32
- E-Fuses Map
- Fw is encrypted



How to break Flash Encryption?

- I did some tests (believe me…)



  - Did not find particular Weakness to access the Key by SW
  - Did not find a way to Attack by DFA
- My Last Hope was Side Channel Analysis to target the Bootloader Decryption
- But my setup was too ‘limited’
  - SPI bus producing a lot of Noise
  - Cannot control the SPI frames properly
  - I tried DPA, CPA… but not enough leakage
- One week later, no result…

Flash Encryption Conclusion

- I lost…

EXTRA-COIN

OTP/E-FUSES: THE MOTHER OF VULNS

Role of OTP/E-Fuses

- One-Time-Programmable (OTP) Memory based on E-Fuses in ESP32
    - An e-Fuse can be ‘programmed’ just ‘One-Time’ from 0 to 1
    - Once burned, no possibility to rewrite it or to wipe it
- Organisation
    - EFUSE_BLK0 = ESP32 configuration
    - EFUSE_BLK1 = Flash Encryption Key (FEK)
    - EFUSE_BLK2 = Secure Boot Key (SBK)
    - EFUSE_BLK3 = reserved for User Application
- According to Espressif, these E-Fuses are R/W protected and cannot be readout/modified once protection bits set
- E-Fuses are managed by the E-Fuses Controller, a dedicated piece of HW inside the ESP32

ESP32 E-Fuses Reverse

- Only two identified functions
  - efuse_read and efuse_program
- Used during a ‘Special Boot mode’
  - interesting…
- BootROM never touch OTP values
- It means only the E-Fuses Controller has access to OTP
  - Pure HW Process
  - Has to be set before BootROM execution

```
ROM:40008600 ; =============== S U B R O U T I N E =======================================
ROM:40008600
ROM:40008600
ROM:40008600 ets_efuse_read_op:
ROM:40008600                 entry           , a1, 0x20
ROM:40008603                 l32r            , a9, dword_400085F8
ROM:40008606                 l32r            , a8, dword_400085F4
ROM:40008609                 memw
ROM:4000860C                 s32i.n          , a9, a8, 0
ROM:4000860E                 l32r            , a8, dword_400085FC
ROM:40008611                 movi.n          , a9, 1
ROM:40008613                 memw
ROM:40008616                 s32i.n          , a9, a8, 0
ROM:40008618
ROM:40008618 loc_40008618:                            ; CODE XREF: ets_efuse_read_op+1D↓j
ROM:40008618                 memw
ROM:4000861B                 l32i.n          , a9, a8, 0
ROM:4000861D                 bnez            , a9, loc_40008618
ROM:40008620                 retw.n
ROM:40008620 ; End of function ets_efuse_read_op
ROM:40008620
ROM:40008620 ; ---------------------------------------------------------------------------
ROM:40008622                 .byte    0
ROM:40008623                 .byte    0
ROM:40008624 dword_40008624  .int 0x5A5A              ; DATA XREF: ets_efuse_program_op+3↓r
ROM:40008628
ROM:40008628 ; =============== S U B R O U T I N E =======================================
ROM:40008628
ROM:40008628
ROM:40008628 ets_efuse_program_op:
ROM:40008628                 entry           , a1, 0x20
ROM:4000862B                 l32r            , a9, dword_40008624
ROM:4000862E                 l32r            , a8, dword_400085F4
ROM:40008631                 memw
ROM:40008634                 s32i.n          , a9, a8, 0
ROM:40008636                 l32r            , a8, dword_400085FC
ROM:40008639                 movi.n          , a9, 2
ROM:4000863B                 memw
ROM:4000863E                 s32i.n          , a9, a8, 0
ROM:40008640
ROM:40008640 loc_40008640:                            ; CODE XREF: ets_efuse_program_op+1D↓j
ROM:40008640                 memw
ROM:40008643                 l32i.n          , a9, a8, 0
ROM:40008645                 bnez            , a9, loc_40008640
ROM:40008648                 retw.n
ROM:40008648 ; End of function ets_efuse_program_op
```

Special Boot Mode

- ESP32 in Special Boot Mode (Download_Boot)
  - Management mode to Flash FW, and Set E-Fuses
  - IO0 connected to GND then Power-up

```
rst:0x10 (RTCWDT_RTC_RESET),boot:0x21 (DOWNLOAD_BOOT(UART0/UART1/SDIO_FEI_REO_V)
waiting for download
```

- Esptool is python utility to communicate with the ROM functions
  - Dedicated commands available from UART0 to deal with E-Fuses
    - dump, program,…

E-Fuses Protection

- Any attempt to read BLK1 or BLK2 returns 0x00
    - $ espefuse.py --port /dev/ttyUSB0 dump

```
espefuse.py v2.7-dev
Connecting....
EFUSE block 0:
00130180 bf4dbb34 00e43c71 0000a000 00000430 f0000000 00000054
EFUSE block 1:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 2:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 3:
00000000 00000000 00000000 00000000 00000000 40000000 00000000 00000000
```

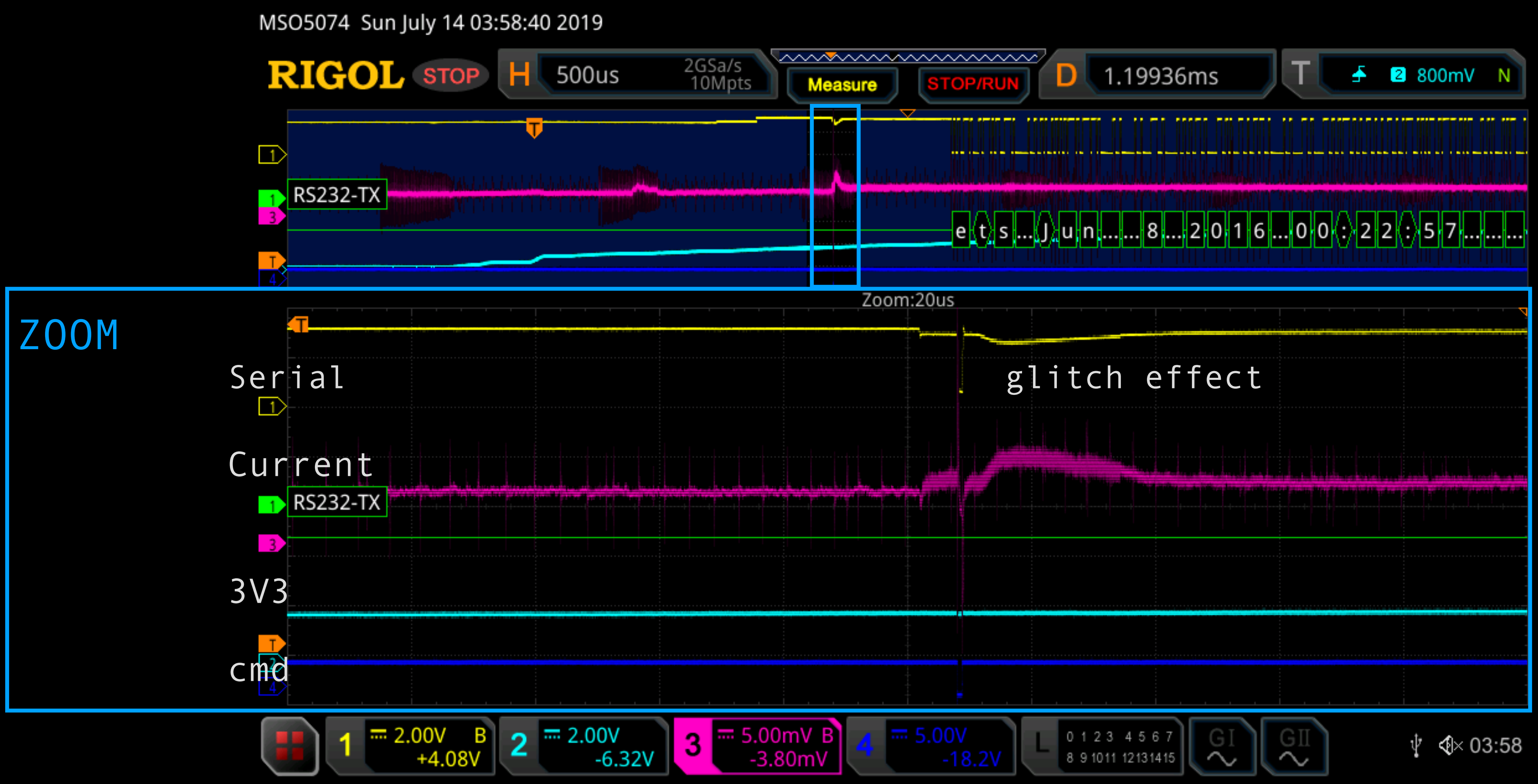
- Identification of R/W Protection bits in BLK0
- 00130180 = 00000000 00010011 00000001 1000 0000
    - These two bits are the Read protect bits

Wait LR, where is the Vuln?

- I have no Vuln here sorry…
- But I know
  - BootROM does not manage the E-Fuses
  - Obviously, E-Fuses Controller does the job before
  - Special boot mode called ‘Download_Boot’
  - Read protection bits have been identified
- The idea
  - Glitch the E-Fuses Controller initialization to modify the R/W protections
  - Then send Dump command in Special Mode
  - Readout BLK1 (FEK) and BLK2 (SBK)

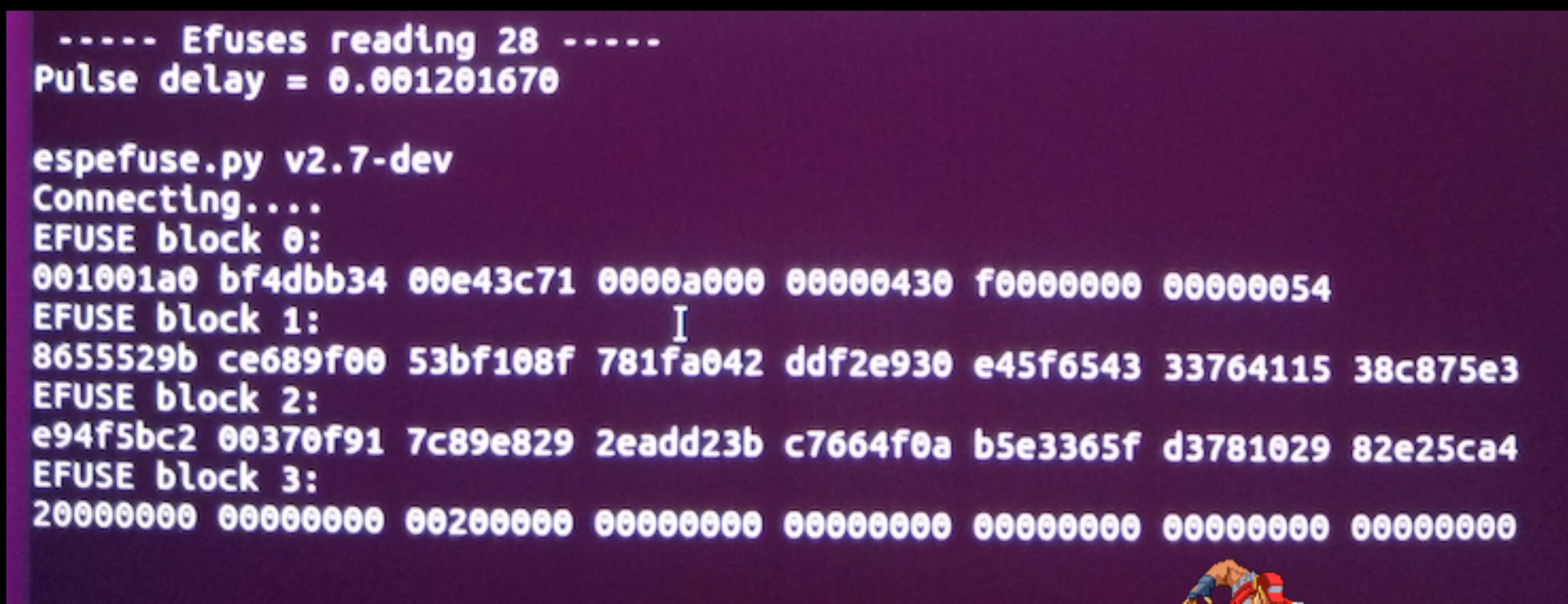
FATAL Glitch

- Simple Power Analysis to identify HW process
- Glitch during this identified activity



FATAL Results

- SBK and FEK extracted from eFuses



One more step

- Sadly, the dumped Keys are not exactly True values
    - Remember I burned the keys ☺
- Offline Statistical Analysis on 30-50 dumped key values
    - just Keep the most recurrent Bytes (here SBK analysis)
- 1 Byte still unknown and has to be Brute Forced (worst case)
    - Same for FEK

0	1	2	3	4	5	6	7
e94f5bc2	00370f91	7c897429	2eadd23b	c7664f05	5ae3365f	d3781029	82e25c4c
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c98
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c98
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c9c
e94f5bc2	00370f91	7c89f029	2eadd23b	c7664f10	bfe3365f	d3781029	82e25c64
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25ce4
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f09	b7e3365f	d3781029	82e25cc8
e94f5bc2	00370f91	7c89e029	2eadd23b	c7664f04	bbe3365f	d3781029	82e25c64
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25ccc
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c1c
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c98
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f08	b6e3365f	d3781029	82e25c98
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c9a
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f08	b7e3365f	d3781029	82e25c62
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0b	b6e3365f	d3781029	82e25c8c
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f09	b7e3365f	d3781029	82e25cc8
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c64
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f09	bfe3365f	d3781029	82e25cc8
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c98
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c80
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c9a
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c9a
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25ce4
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f08	b7e3365f	d3781029	82e25c64
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f08	b7e3365f	d3781029	82e25c0c
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25ca4
e94f5bc2	00370f91	7c89e029	2eadd23b	c7664f01	bfe3365f	d3781029	82e25cc8
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c9c
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25c06
e94f5bc2	00370f91	7c89e829	2eadd23b	c7664f0a	b5e3365f	d3781029	82e25cef
e94f5bc2	00370f91	7c89f429	2eadd23b	c7664f09	fee3365f	d3781029	82e25c4c

Appearance Rate:

0	1	2	3	4	5	6	7
100%	100%	100%	100%	60%	60%	100%	0%(1 Byte by BF)

Real Secure Boot Key:
e94f5bc2 00370f91 7c89e829 2eadd23b c7664f0a b5e3365f d3781029 82e25c99

FATAL Exploit step 1: Decrypt FW

- Dump the encrypted FW
  - By Download Mode or by dumping Flash
- Perform FATAL Glitch to extract FEK and SBK
  - Run Statistical analysis
- Confirm the True FEK (by decrypting FW)

```
limited@linux:~/esp/bin_decrypt_dump$ espsecure.py decrypt_flash_data --keyfile my_dumped_
fek.bin --output decrypted.bin --address 0x0 flash_contents.bin
espsecure.py v2.7-dev
Using 256-bit key
limited@linux:~/esp/bin_decrypt_dump$ strings decrypted.bin | grep Hello
Hello from SEC boot K1 & FE !
```

- Respect the bytes order in binary file
  -

```
limited@linux:~/esp/bin_decrypt_dump$ hexdump -C my_dumped_fek.bin
00000000  38 c8 75 e3 33 76 41 15  f9 5f 65 43 dd f2 e9 2c  |8.u.3vA.._eC...,|
00000010  78 1f a0 42 53 bf 14 8f  ce 68 9f 00 86 55 52 9b  |x..BS....h...UR.|
```

FATAL Exploit step 2: Sign Your Code

- Firmware is now decrypted
- dd ivt.bin (the first 128 random bytes at 0x00 in decrypted.bin)
- dd Bootloader.bin at 0x1000
- Confirm the true SBK
    - digest computation command
- Write your Code
    - a little FW backdoor maybe? ☺
- Compile images
    - using FEK and SBK
- Flash new FW

```
limited@linux:~/esp/bin_decrypt_dump$ hexdump -C -n 192 decrypted.bin
00000000  bd 84 e7 f2 39 b8 8f 55  fb d9 48 9b 26 c8 c2 d3  |....9..U..H.&...|
00000010  9c 13 72 d9 5a 77 94 0d  67 ed 2d 48 fc 69 aa 5f  |..r.Zw..g.-H.i._|
00000020  0d 1c 4d ef 67 ec a1 43  d3 3a 67 86 9f e3 e3 58  |..M.g..C.:g....X|
00000030  9a 80 85 31 b7 9f cb 27  ad 35 e0 bb 2f 93 8d 79  |...1...'.5../..y|
00000040  22 5e e5 22 ca e1 eb 9c  2e 4d d8 93 fc 97 66 5a  |"^."......M....fZ|
00000050  4b 58 8c 24 a9 04 78 e4  45 99 94 37 3d b6 4b 7f  |KX.$..x.E..7=.K.|
00000060  70 d4 df 56 7f 1f b8 52  24 0c 0d 45 22 e1 d1 d5  |p..V...R$..E"...|
00000070  cf 2d 85 2b e9 f1 01 9d  04 88 5c bf 17 ab b6 2f  |.-.+......\..../|
00000080  b5 a5 82 70 5c 3e 1e 25  44 30 92 84 d0 13 a4 bc  |...p\>.%D0......|
00000090  b0 d4 ee 63 01 ee a0 d5  72 07 91 51 67 82 a8 8d  |...c....r..Qg...|
000000a0  6c a5 2a 1e 5e 39 29 d7  60 1b 9d 22 3e dc f4 64  |l.*.^9).`..">..d|
000000b0  6f c7 bf 2e ba a7 9a bf  24 4b dc d0 fc 87 ee bb  |o.......$K......|
000000c0
limited@linux:~/esp/bin_decrypt_dump$ espsecure.py digest_secure_bootloader --keyf
ile my_dumped_sbk.bin --iv ivt.bin bootloader.bin
espsecure.py v2.7-dev
WARNING: --iv argument is for TESTING PURPOSES ONLY
Using 256-bit key
digest+image written to bootloader-digest-0x0000.bin
limited@linux:~/esp/bin_decrypt_dump$ hexdump -C -n 192 bootloader-digest-0x0000.b
in
00000000  bd 84 e7 f2 39 b8 8f 55  fb d9 48 9b 26 c8 c2 d3  |....9..U..H.&...|
00000010  9c 13 72 d9 5a 77 94 0d  67 ed 2d 48 fc 69 aa 5f  |..r.Zw..g.-H.i._|
00000020  0d 1c 4d ef 67 ec a1 43  d3 3a 67 86 9f e3 e3 58  |..M.g..C.:g....X|
00000030  9a 80 85 31 b7 9f cb 27  ad 35 e0 bb 2f 93 8d 79  |...1...'.5../..y|
00000040  22 5e e5 22 ca e1 eb 9c  2e 4d d8 93 fc 97 66 5a  |"^."......M....fZ|
00000050  4b 58 8c 24 a9 04 78 e4  45 99 94 37 3d b6 4b 7f  |KX.$..x.E..7=.K.|
00000060  70 d4 df 56 7f 1f b8 52  24 0c 0d 45 22 e1 d1 d5  |p..V...R$..E"...|
00000070  cf 2d 85 2b e9 f1 01 9d  04 88 5c bf 17 ab b6 2f  |.-.+......\..../|
00000080  b5 a5 82 70 5c 3e 1e 25  44 30 92 84 d0 13 a4 bc  |...p\>.%D0......|
00000090  b0 d4 ee 63 01 ee a0 d5  72 07 91 51 67 82 a8 8d  |...c....r..Qg...|
000000a0  6c a5 2a 1e 5e 39 29 d7  60 1b 9d 22 3e dc f4 64  |l.*.^9).`..">..d|
000000b0  6f c7 bf 2e ba a7 9a bf  24 4b dc d0 fc 87 ee bb  |o.......$K......|
000000c0
limited@linux:~/esp/bin_decrypt_dump$ hexdump -C my_dumped_sbk.bin
00000000  82 e2 5c 99 d3 78 10 29  b5 e3 36 5f c7 66 4f 0a  |..\..x.)..6_.fO.|
00000010  2e ad d2 3b 7c 89 e8 29  00 37 0f 91 e9 4f 5b c2  |...;|..).7...O[.|
00000020
```

OTP/EFuses FATAL Conclusion

- FATAL exploit leading to SBK and FEK extraction
  - Breaking Secure Boot and Flash Encryption
- An attacker can decrypt the Firmware (and access sensitive data)
- An attacker can sign & run his own (encrypted) code PERSISTENTLY
- Low Cost, Low Complexity
- Easy to reproduce
- No Way to fix
- All ESP32 versions vulnerable

Vendor Reaction

- Resp. disclosure
  - PoC sent on July 24
  - CVE-2019-17391 (req. by Vendor)
  - Posted on November 13
- Security Advisory on November 1 [7]

> The ESP32-D0WD-V3 chip has checks in ROM which prevent fault injection attack. This chip and related modules will be available in Q4 2019. More information about ESP32-D0WD-V3 will be released soon.

- No way to Fix but…you can buy the next version ☺
- Millions of vulnerable Devices on the field for the coming years

The impact

- For Hobbyists
  - Don’t worry, your ‘connected DIY device’ is safe ☺
- For Developers
  - In case you are using the ESP32 security features to protect SECRETS, you should be worried…
  - FYI, I identified 3 companies using ESP32 Flash Enc. and Sec.Boot in their products to protect their ‘business model’
- For the vendor
  - Force to modify silicon to save his longevity commitment and his reputation
  - What about current devices offered for sales?

Final Conclusion

- Attacker with physical access can compromise ESP32 security badly
  - PERSISTENT Bypass of Secure Boot + Flash Encryption
- Fix?
  - No fix on current ESP32 version
- Platform is broken
  - A new chip version will be released
- General Message for Vendors
  - Don't patch silently
- New Results coming soon, stay tuned!

References & Credits

- Espressif
  - [1] Espressif 100-Millions chip shipments
- ESP32
  - [2] Datasheet, TRM
- Fault injection references
  - [3] Chris Gerlinsky (@akacastor)
  - [4] Colin O'Flynn (@colinoflynn)
- Xtensa
  - [5] ISA Manual
- LIFX Pwn
  - [6] LIFX Pwn
- Security Advisory
  - [7] CVE-2019-17391
- Fatal Fury Animations

Thank you!

@LimitedResults

www.limitedresults.com

Black Hat Europe 2019