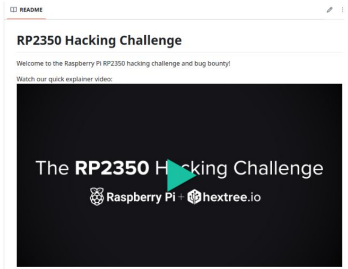
Security through Transparency: Tales from the RP2350 Hacking Challenge

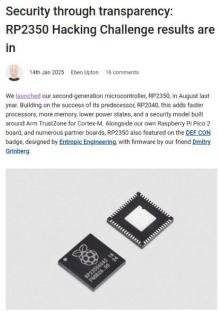
Marius Muench[1], Aedan Cullen[2], Kévin Courdesses[2], Thomas 'stacksmashing' Roth[3], Andrew Zonenberg[4],

[1]University of Birmingham [2]Independent [3]Hextree [4]IOActive

This Presentation

Context





RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

The attacks


Attacking the OTP PSM


Forcing a Vector Boot


Laser Fault Injection


OTP Read Double Fault


FIB/PVC Antifuse Data Extraction

Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

This Presentation

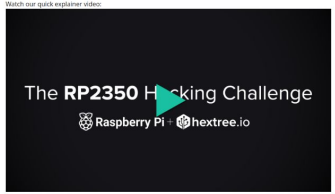
Context

README

RP2350 Hacking Challenge

Welcome to the Raspberry Pi RP2350 hacking challenge and bug bounty!

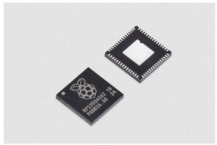
Watch our quick explainer video:



Security through transparency: RP2350 Hacking Challenge results are in

14th Jan 2025 Eben Upton 16 comments

We launched our second-generation microcontroller, RP2350, in August last year. Building on the success of its predecessor, RP2040, this adds faster processors, more memory, lower power states, and a security model built around Arm TrustZone for Cortex-M. Alongside our own Raspberry Pi Pico 2 board, and numerous partner boards, RP2350 also featured on the DEF CON badge, designed by Entropic Engineering, with firmware by our friend Dmitry Grinberg.



RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

The attacks

Attacking the OTP PSM

Forcing a Vector Boot

Laser Fault Injection

OTP Read Double Fault

FIB/PVC Antifuse Data Extraction

Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

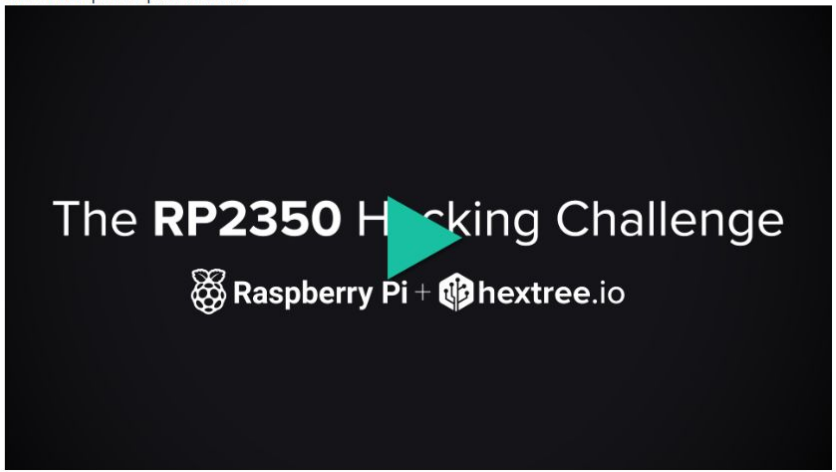
Context

README

RP2350 Hacking Challenge

Welcome to the Raspberry Pi RP2350 hacking challenge and bug bounty!

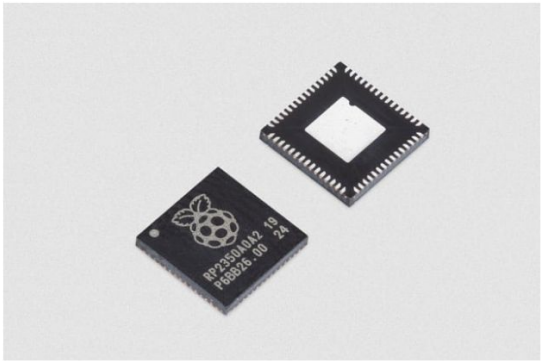
Watch our quick explainer video:



Security through transparency: RP2350 Hacking Challenge results are in

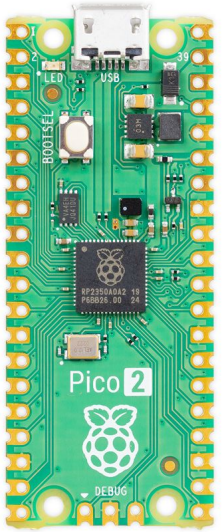
14th Jan 2025 Eben Upton 16 comments

We launched our second-generation microcontroller, RP2350, in August last year. Building on the success of its predecessor, RP2040, this adds faster processors, more memory, lower power states, and a security model built around Arm TrustZone for Cortex-M. Alongside our own Raspberry Pi Pico 2 board, and numerous partner boards, RP2350 also featured on the DEF CON badge, designed by Entropic Engineering, with firmware by our friend Dmitry Grinberg.

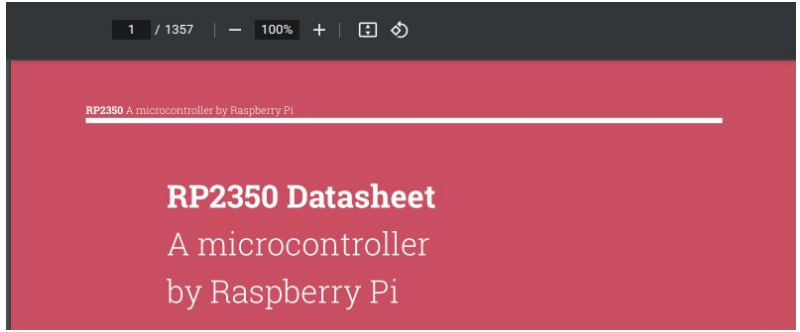


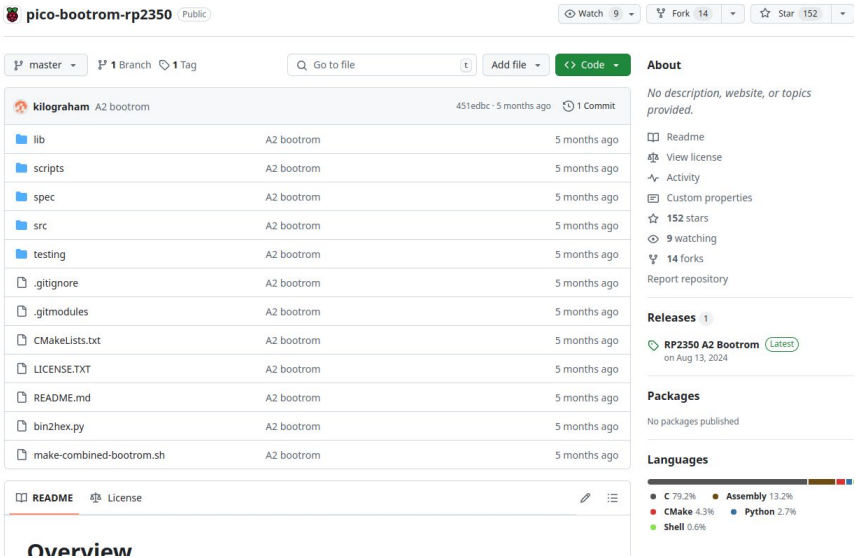
The RP2350

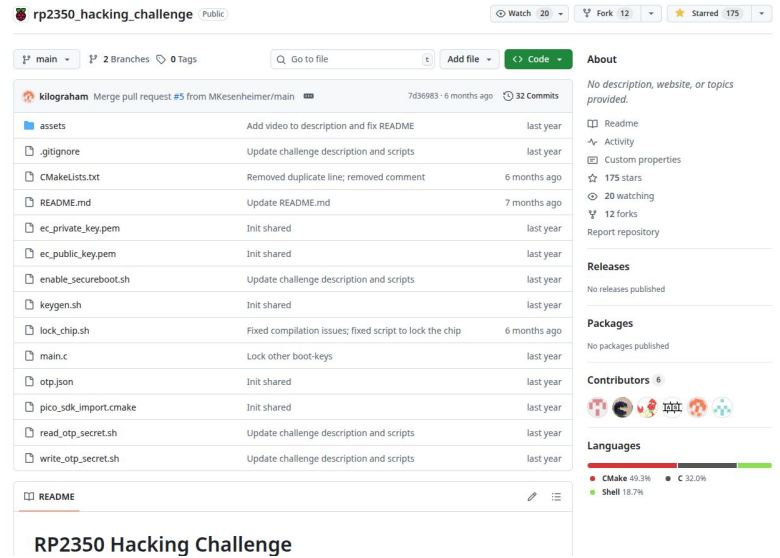
- Released in August 2024
- Dual core MCU @ 150 MHz
- Each core can use a different ISA
  - ARM Cortex-M33 (ARMv8-M)
  - Hazard3 RISC-V (RV32IMAC+)
- Secure Bootloader
  - With fault injection hardening
  - Designed for Arm



Transparency

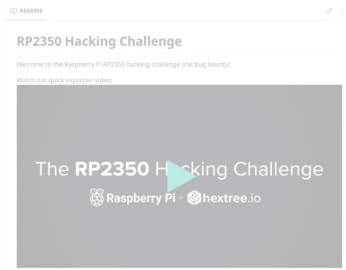


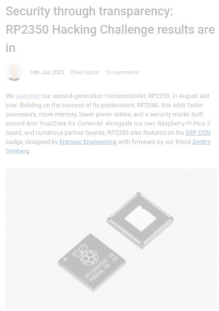




This Presentation

Context





RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

The attacks

Attacking the OTP PSM

Forcing a Vector Boot

Laser Fault Injection

OTP Read Double Fault

FIB/PVC Antifuse Data Extraction

Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

One-Time Programmable (OTP)Memory

- Antifuse Memory
- Configures critical components
  - Enabled cores
  - Debug access
  - Glitch detectors
- Relied on during secure boot
  - Key material
- Allows “locking” of OTP pages

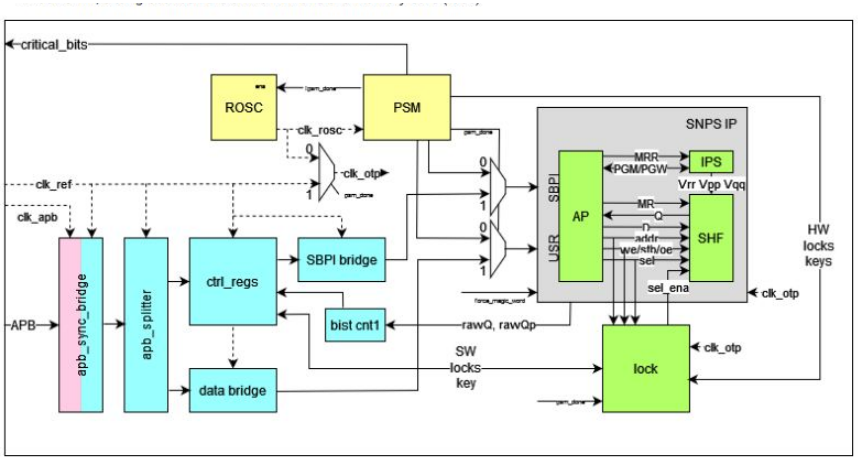
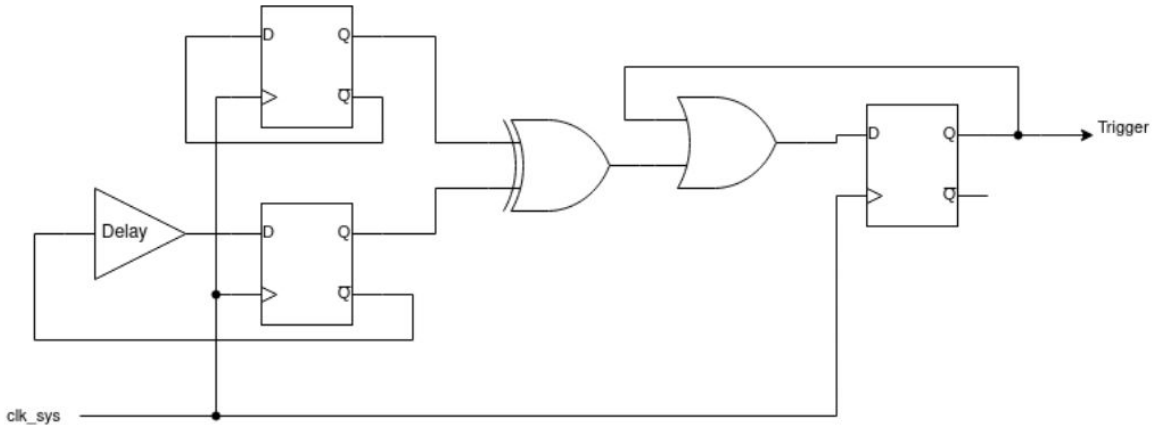


Image Source: RP2350 datasheet, Section 13.3.

The Glitch Detector

The glitch detector is comprised of four identical detector circuits, based on a pair of D flip-flops. These detector circuits are each placed in different, physically distant locations within the core voltage domain.

Figure 43. Glitch detector trigger circuit. Two flops each toggle on every system clock cycle. One has a programmable delay line in its feedback path, the other does not. Loss of setup or hold margin causes one of the flops to fail to toggle, so the flops values differ, setting the trigger output.



The detector triggers when the two D-flops take on different values, which is impossible under normal circumstances. The delay line is programmable from 75% to 120% of the minimum system clock period in increments of 15%. Higher delays make the circuit more sensitive to loss of setup margin. To configure initial sensitivity, use the GLITCH_DETECTOR_SENS OTP flags. You can fine-tune sensitivity for each detector using the SENSITIVITY register.

Image Source: RP2350 datasheet, Section 10.9.1.

Redundancy Co-Processor

- Stack Canaries
- Instruction Delays
- Sequence Count Checking
- Integer & Boolean Validation
- Panic

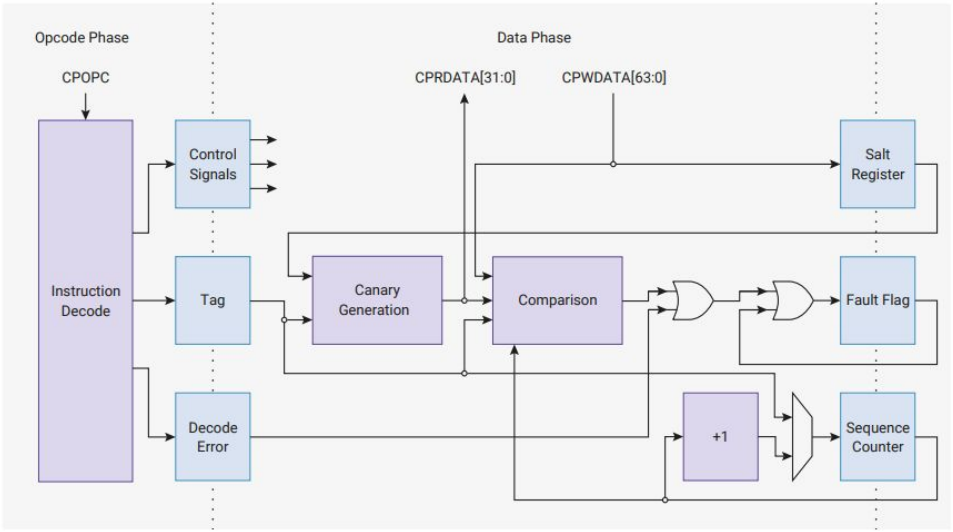
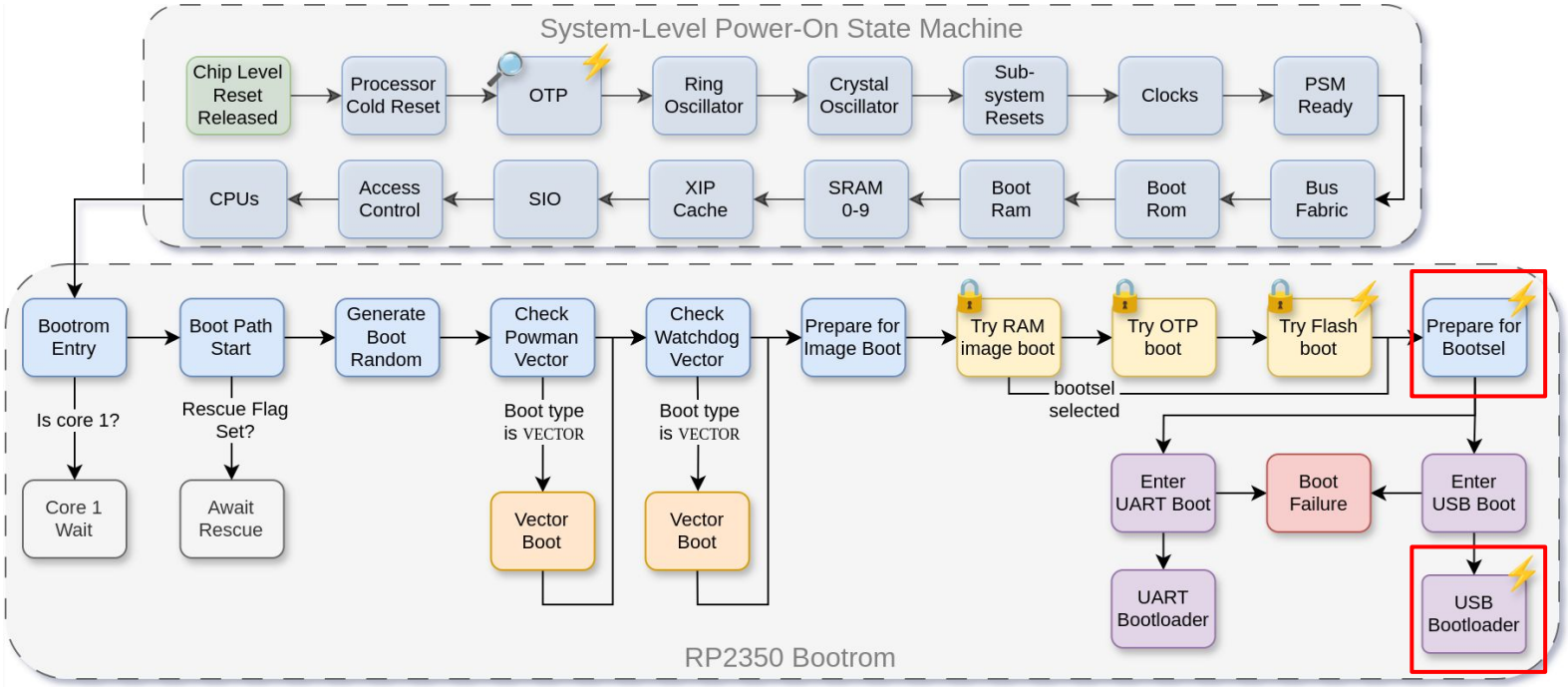


Image Source: RP2350 datasheet, Section 3.6.3

RP2350 Boot Process

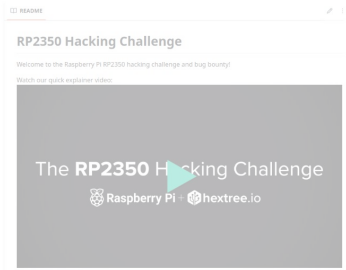


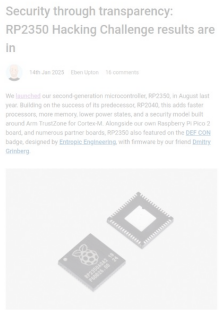
Challenge Setup

1) Generate a public/private key pair
2) Write a 128-bit secret value to OTP row 0xc08
3) Write the sha256 value of the public key to the bootkey0
4) Enabling secure boot
5) Lock down the chip
6) Run custom firmware removing NS access to secret value

This Presentation

Context





RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

The attacks


Attacking the OTP PSM


Forcing a Vector Boot


Laser Fault Injection

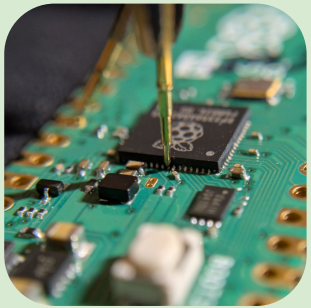

OTP Read Double Fault


FIB/PVC Antifuse Data Extraction

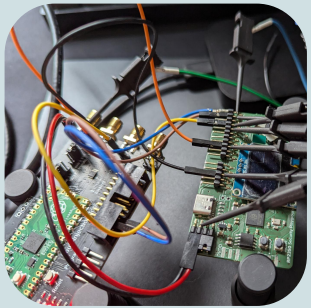
Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

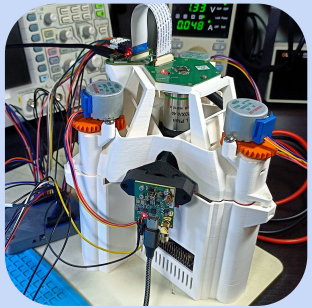
The attacks



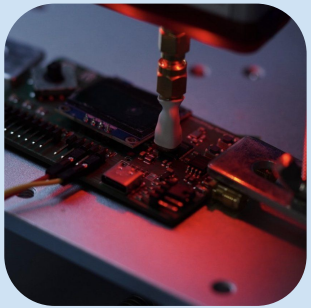
Attacking the OTP PSM



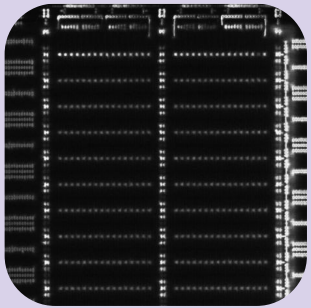
Forcing a Vector Boot



Laser Fault Injection

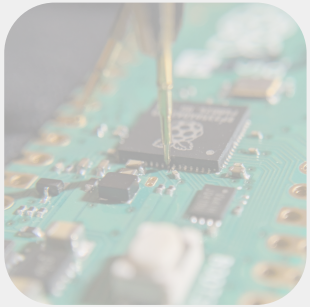


OTP Read Double Fault

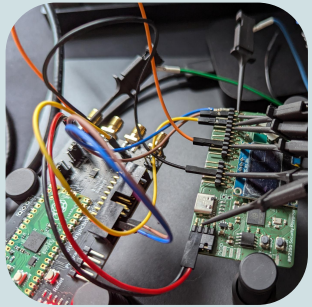


FIB/PVC Antifuse Data Extraction

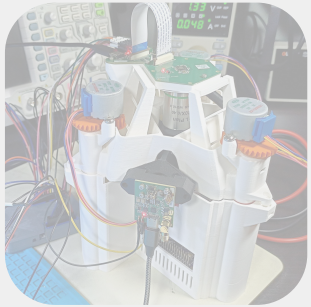
The attacks



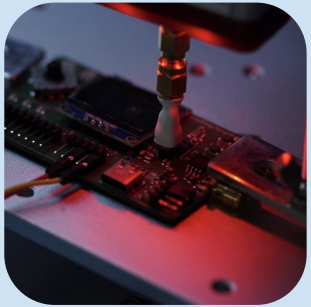
Attacking the OTP PSM



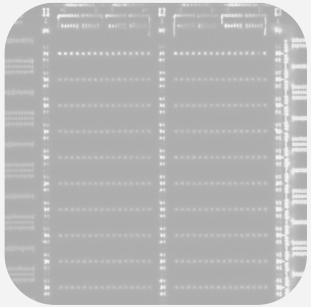
Forcing a Vector Boot



Laser Fault Injection

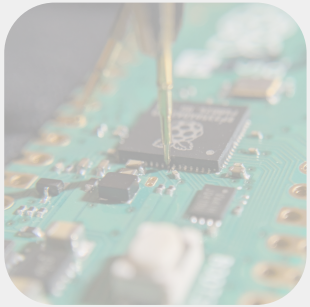


OTP Read Double Fault

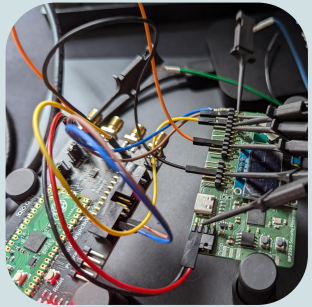


FIB/PVC Antifuse Data Extraction

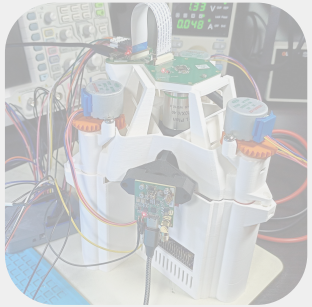
The attacks



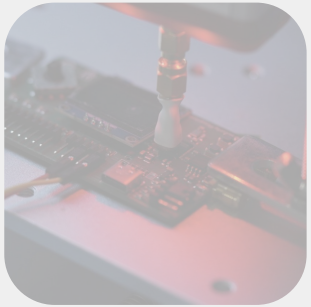
Attacking the OTP PSM



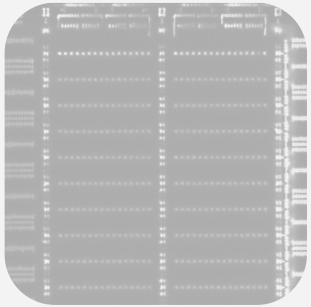
Forcing a Vector Boot



Laser Fault Injection



OTP Read Double Fault



FIB/PVC Antifuse Data Extraction

Context: USB Boot Loader

- Runs in Non-Secure execution state
  - Some functionality uses Bootrom APIs
  - These may also be accessed from firmware running in S

API Function	Short Description	NS	S
`flash_op`	Perform a flash read, erase, or program operation	✓	✓
`otp_access`	Writes or reads data to/from OTP	✓	✓
`reboot`	Resets the RP2350 and uses the watchdog facility to restart	✓	✓
`secure_call`	Call a Secure method from Non-Secure code	✓	

- Implements the interaction with Picotool
  - Custom command packets
  - One of them: Reset

Reboot Types

- RP2350 implements multiple reboot types
  - E.g., reboot normally, reboot from RAM

- One special boot type:

- `0x000d` : `REBOOT_TYPE_PC_SP` - reboot to a specific PC and SP. Note: this is not allowed in the `Arm-NS` variant.
  - `p0` - the initial program counter (PC) to start executing at. This must have the lowest bit set for Arm and clear for RISC-V
  - `p1` - the initial stack pointer (SP).

Reset Cmd Boot Flow

- NS: Receive command &
  jump to S entry with reboot-flag

- S: Sanitize parameters
  (including boot-type)
  - FI hardened!

- S: Jump to shared reset code
  - Verify boot type registers are matching
  - Execute requested boot

```
varm_misc.S:

// Redundant args are passed in r0/*sp, and we want them to mismatch when
   // bit 3 is set (set for all Secure-only reboot types).

   // first check that bit 3 isn't set (since that shouldn't be calling from NS)
   lsls r4, r0, #29
   bcs fail_reboot

   // now, try quite hard to clear bit 3 in r0, so that if bit3 were set in our
   // now-stacked 5th argument to s_varm_hx_reboot, the value in r0 won't match
   // and we'll get an rcp_ violation in the callee

   // 1. put r0 back together without bit 3
   lsrs r4, r4, #29
   lsrs r0, r0, #4
   lsls r0, r0, #4
   orrs r0, r4
   // 2. clear bit 3
   movs r4, #8
   bics r0, r4
   bl s_varm_hx_reboot
   b reboot_retur
```

Vulnerable Instruction Sequence

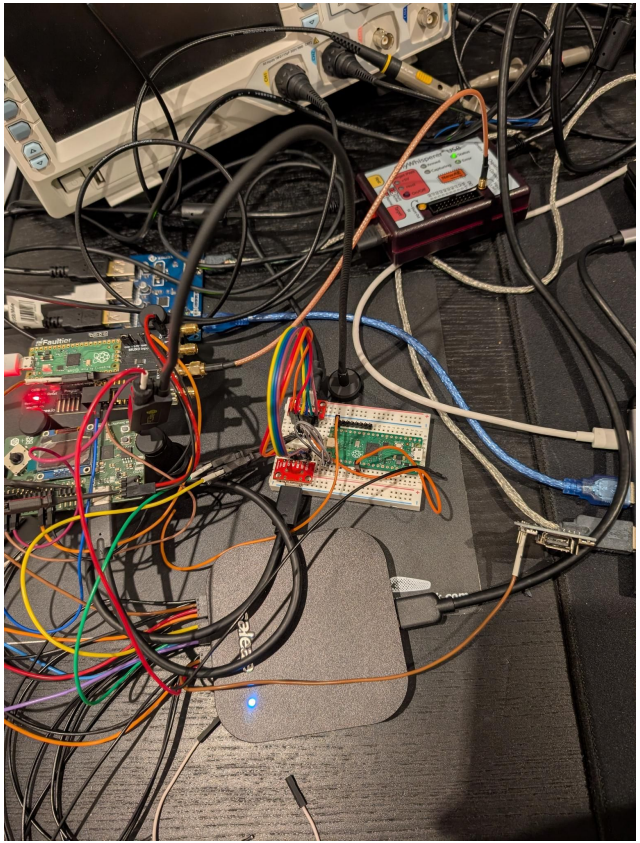
```
                        s_varm_api_reboot_end        XREF[4]:  Entry Point(*),
                        s_varm_hx_reboot                       s_from_nsboot_var…
                                                               s_varm_api_reboot…
                                                               .debug_frame::000…
00000578 f7 b5      push    {flags,delay_ms,p0,r4,r5,r6,r…
0000057a 14 fe      mrc2    p7,0x0,r4,cr4,cr4,0x1
         34 47
0000057e 01 94      str     r4,[sp,#__stack_canary_value]
00000580 08 9c      ldr     r4,[sp,#flags2]
00000582 44 ec      mcrr    p7,0x7,flags,r4,cr0
         70 07
00000586 00 25      movs    r5,#0x0
00000588 02 26      movs    r6,#0x2
0000058a 26 4c      ldr     r4,[DAT_00000624]                 = 400D8000h
0000058c 76 42      rsbs    r6,r6
0000058e 25 60      str     r5,[r4,#0x0]=>DAT_400d8000
00000590 25 4d      ldr     r5,[DAT_00000628]                 = 40018000h
00000592 ae 60      str     r6,[r5,#0x8]=>DAT_40018008
00000594 0f 25      movs    r5,#0xf
00000596 32 36      adds    r6,#0x32
00000598 05 40      ands    r5,flags
0000059a 06 40      ands    r6,flags
0000059c 0d 2d      cmp     r5,#0xd
0000059e 09 d1      bne     do_other_reboot_types
000005a0 6e b9      cbnz    r6,reboot_to_specific_arch

                        do_reboot_type_pc_sp         XREF[1]:  000005dc(j)
000005a2 22 4d      ldr     r5,[DAT_0000062c]                 = B007C0D3h
000005a4 e5 61      str     r5,[r4,#0x1c]=>DAT_400d801c
000005a6 e5 69      ldr     r5,[r4,#0x1c]=>DAT_400d801c
000005a8 6d 42      rsbs    r5,r5
```

Attack strategy

- Load unsigned firmware in RAM
- Send USB Reset Command with PC/SP to attack code
- Glitch critical instruction

Implementation Steps:

- Develop attack code
- Verify feasibility
- Bypass glitch detector
- Find & calibrate trigger
- Bypassing random instruction delay of RCP



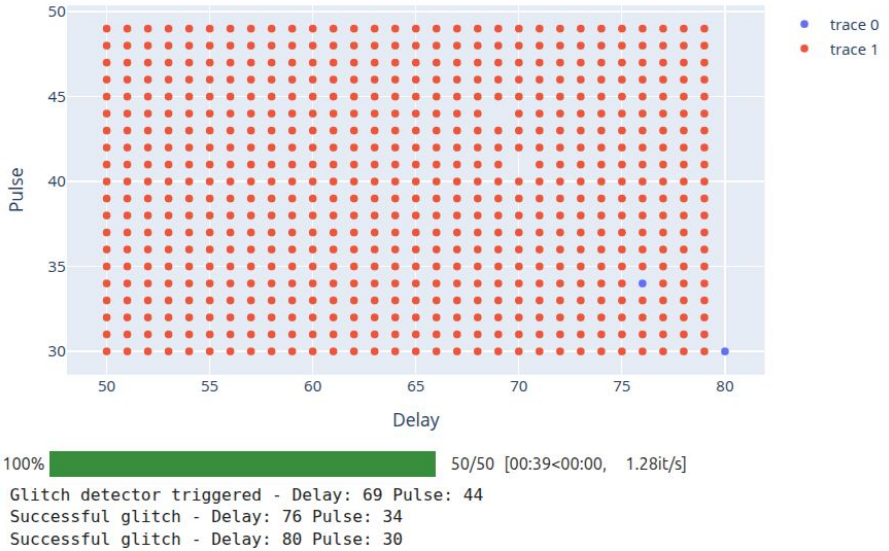
Bypassing the Glitch Detector (Voltage FI)

Experimental profiling:

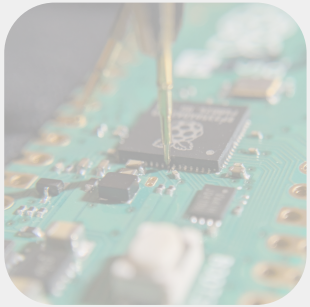
- Glitch detector calibrated to 150MHz clock

However:

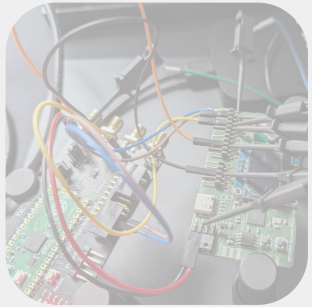
- USB Bootloader runs from different, slower clock!



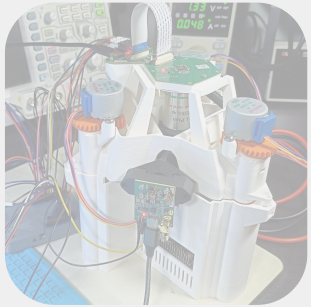
The attacks



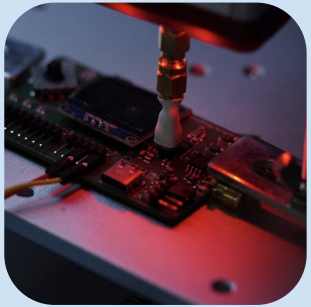
Attacking the OTP PSM



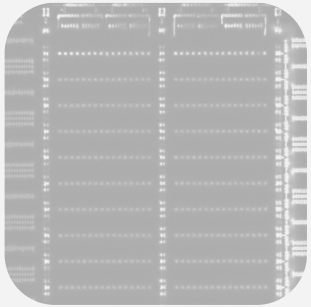
Forcing a Vector Boot



Laser Fault Injection



OTP Read Double Fault



FIB/PVC Antifuse Data Extraction

Context: OTP Locking & Entering Bootsel

- All OTP configuration available during early boot
- When preparing Bootsel mode, OTP locks are read and applied
- Picotool/user can read non-locked OTP values

```
nsr@xpad ~ % picotool otp get -e 0x40
ROW 0x0040: OTP_DATA_CRIT1
        "Page 1 critical boot flags (RBIT-8)"

    VALUE 0x2b0071
    RAW_VALUE=0x000071;0x000071;0x000071;0x000071;0x000071;0x000071
;0x000071;0x000071(WARNING - ECC IS INVALID)
    field GLITCH_DETECTOR_SENS (bits 5-6) = 3
        "Increase the sensitivity of the glitch detectors from
        their default."
    field GLITCH_DETECTOR_ENABLE (bit 4) = 1
        "Arm the glitch detectors to reset the system if an
        abnormal clock/power event is observed."
    field BOOT_ARCH (bit 3) = 0
        "Set the default boot architecture, 0=ARM 1=RISC-V.
        Ignored if ARM_DISABLE, RISCV_DISABLE or
        SECURE_BOOT_ENABLE is set."
    field DEBUG_DISABLE (bit 2) = 0
        "Disable all debug access"
    field SECURE_DEBUG_DISABLE (bit 1) = 0
        "Disable Secure debug access"
    field SECURE_BOOT_ENABLE (bit 0) = 1
        "Enable boot signature enforcement, and permanently
        disable the RISC-V cores."
```

Locking Down OTP Pages

```
for(; page < NUM_OTP_PAGES; page++) {
    uint32_t sw_lock = s_otp_advance_bl_to_s_value(0xff, page);
    uint32_t sw_lock2 = s_otp_advance_bl_to_s_value(0xff, page2);
    hx_assert_equal2i(sw_lock, sw_lock2);
    otp_hw->sw_lock[page] = sw_lock;
    uint32_t sw_lock_verify = otp_hw->sw_lock[page];
    page2 += (sw_lock_verify & sw_lock) == sw_lock;
}
hx_set_step(STEPTAG_NSBOOT_OTP_ADVANCE);
hx_assert_equal2i(page2, NUM_OTP_PAGES);
```

Vulnerable Instruction Sequence

sw_lock:

`0b0000 0000 0000 0000 0000 0000 0000 1111`

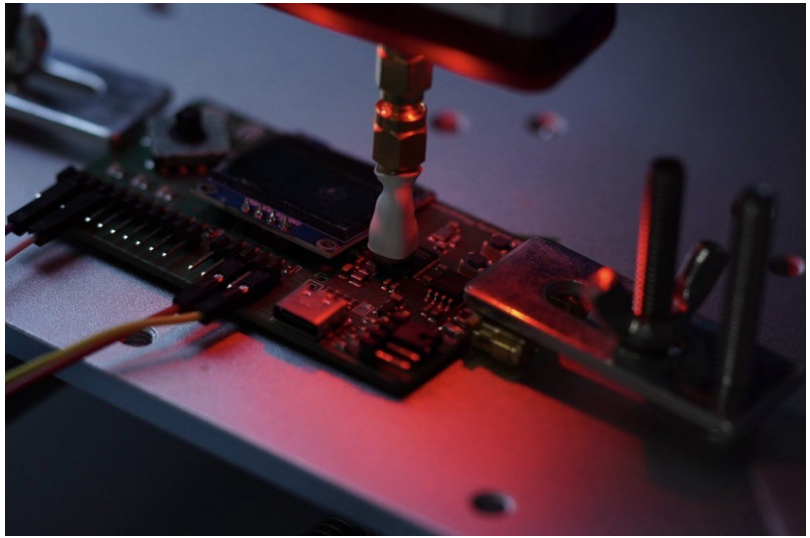
`0b1111 0000 0000 0000 0000 0000 0000 0000`

Ignored by OTP Locking Hardware!

```
                    s_otp_advance_bl_to_s_value  XREF[3]:  Entry Point(*),
                                                           00003f38(c),
                                                           00003f42(c)
00003380 89 ee      mcr     p7,0x4,r0,cr9,cr0,0x0
         10 07
00003384 08 4a      ldr     r2,[DAT_000033a8]                  = 4013FE04h
00003386 c9 00      lsls    r1,r1,#0x3
00003388 50 58      ldr     r0,[r2,r1]
0000338a 01 0b      lsrs    r1,r0,#0xc
0000338c 02 03      lsls    r2,r0,#0xc
0000338e 11 43      orrs    r1,r2
00003390 00 09      lsrs    r0,r0,#0x4
00003392 01 40      ands    r1,r0
00003394 0c 20      movs    r0,#0xc

                    LAB_00003396                       XREF[1]:  0000339a(j)
00003396 08 43      orrs    r0,r1
00003398 09 0a      lsrs    r1,r1,#0x8
0000339a fc d1      bne     LAB_00003396
0000339c 00 07      lsls    r0,r0,#0x1c
0000339e 00 0f      lsrs    r0,r0,#0x1c
000033a0 a9 ee      mcr     p7,0x5,r0,cr9,cr0,0x1
         30 07
000033a4 70 47      bx      lr
```

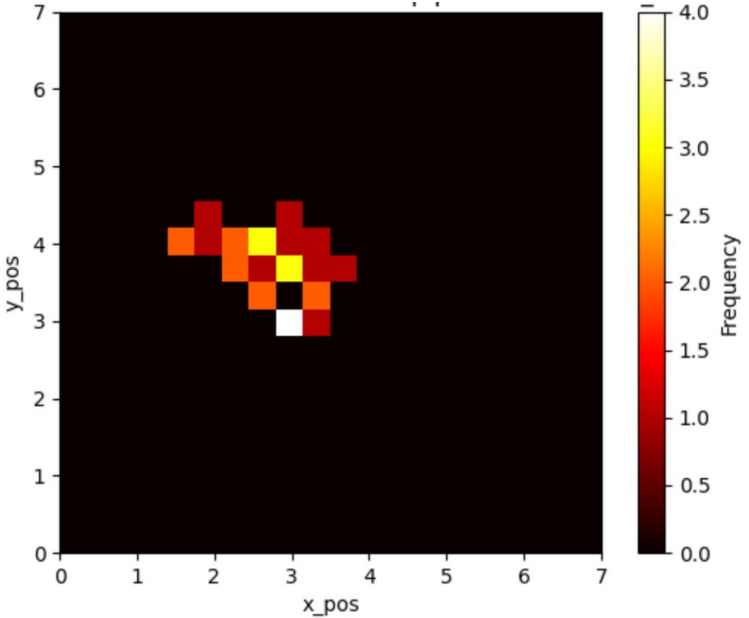
Attack strategy

- Setup boot to “bootsel” mode
- Bypass Glitch Detector
- Glitch critical instruction twice
- Readout secret via picotool



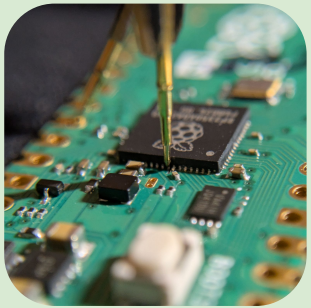
Bypassing the Glitch Detector (EMFI)

- Glitch detectors measure at specific physical locations of chip
- With EMFI, we can localize the fault area
- Glitch detector may not pick up on injected faults



Mitigations & Errata

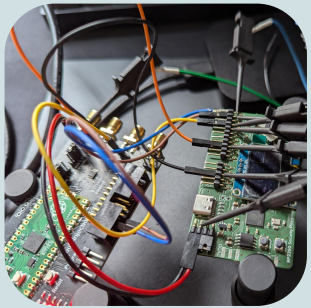
E16



Attacking the OTP PSM

Fixed by: A3

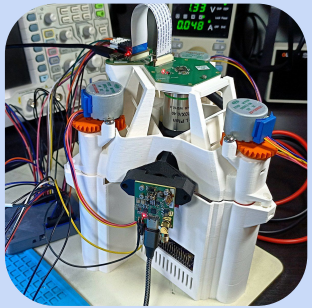
E20



Forcing a Vector Boot

Fixed by: A3

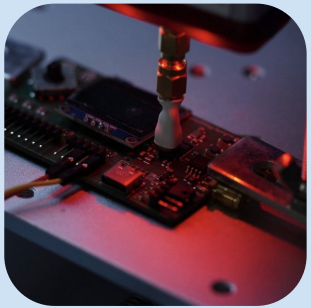
E24



Laser Fault Injection

Fixed by: A4

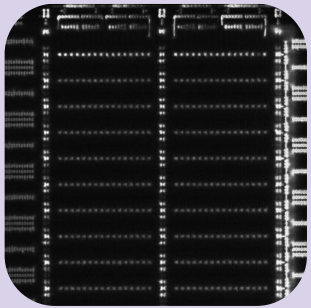
E21



OTP Read Double Fault

Fixed by: A3

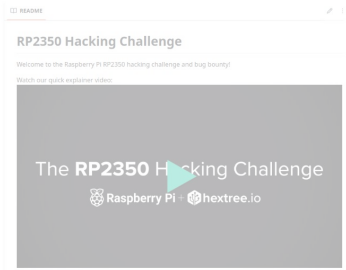
-

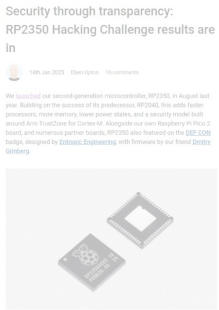


FIB/PVC Antifuse Data Extraction

This Presentation

Context





RP2350 Security Features

- Cortex-M33 Security Features
- One-Time Programmable Memory
- Glitch Detector
- Redundancy Coprocessor
- Secure Bootchain

The attacks

Attacking the OTP PSM

Forcing a Vector Boot

Laser Fault Injection

OTP Read Double Fault

FIB/PVC Antifuse Data Extraction

Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

Lessons Learned

- Importance of fault models for pre-boot attacks
- Effectiveness of Hardware Mitigations
- Dangers of complicated boot paths
- Costs of laser fault injection

Conclusion

Security through transparency is 😍

Artifact & Paper

https://github.com/bhamsec/woot25-rp2350-challenge