Beyond Oberon

Shawn Hoffman
symbrkrs

Outline

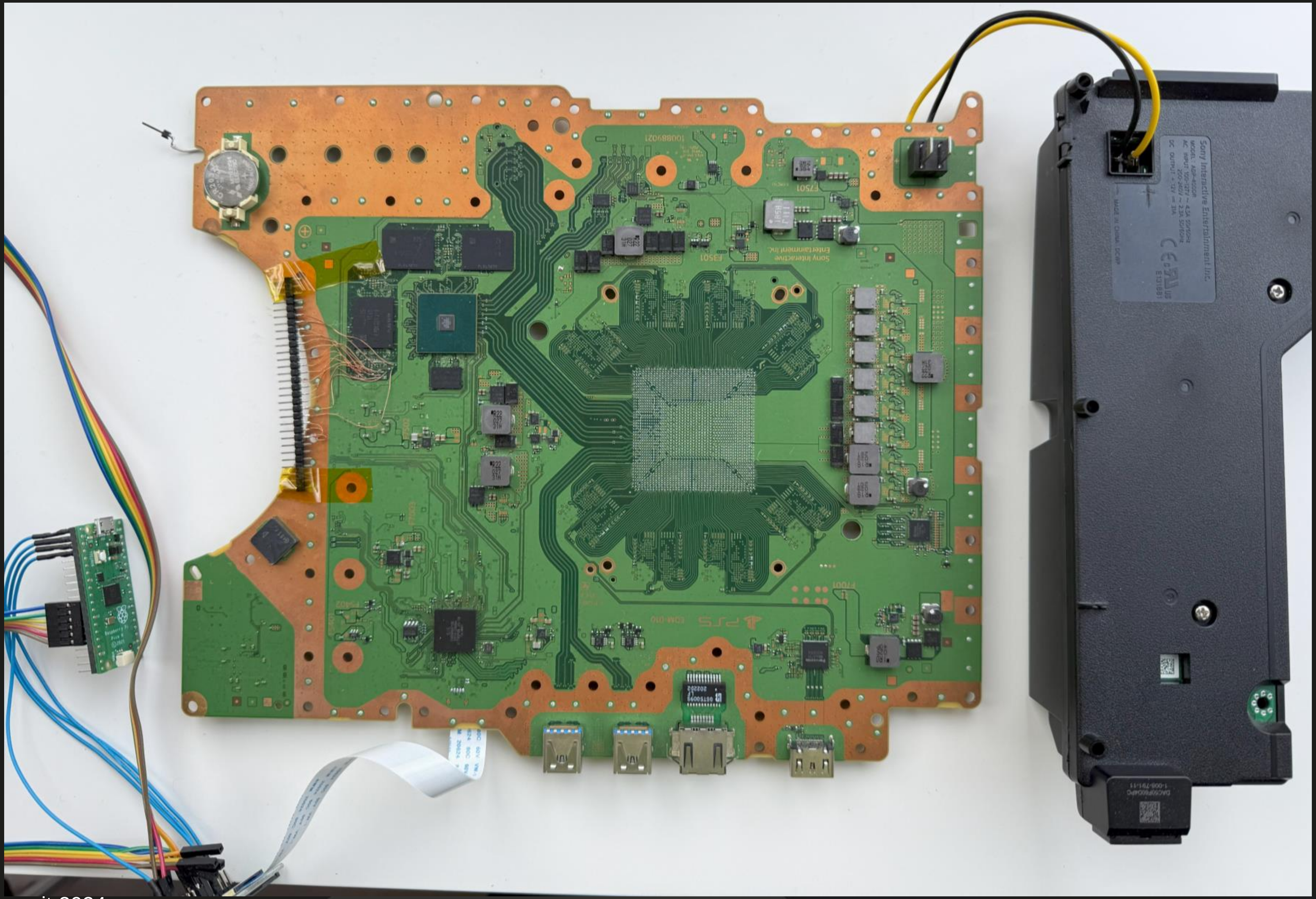
- High-Level Security Overview
- Boot Flow
- Salina
- Titania

Security Overview

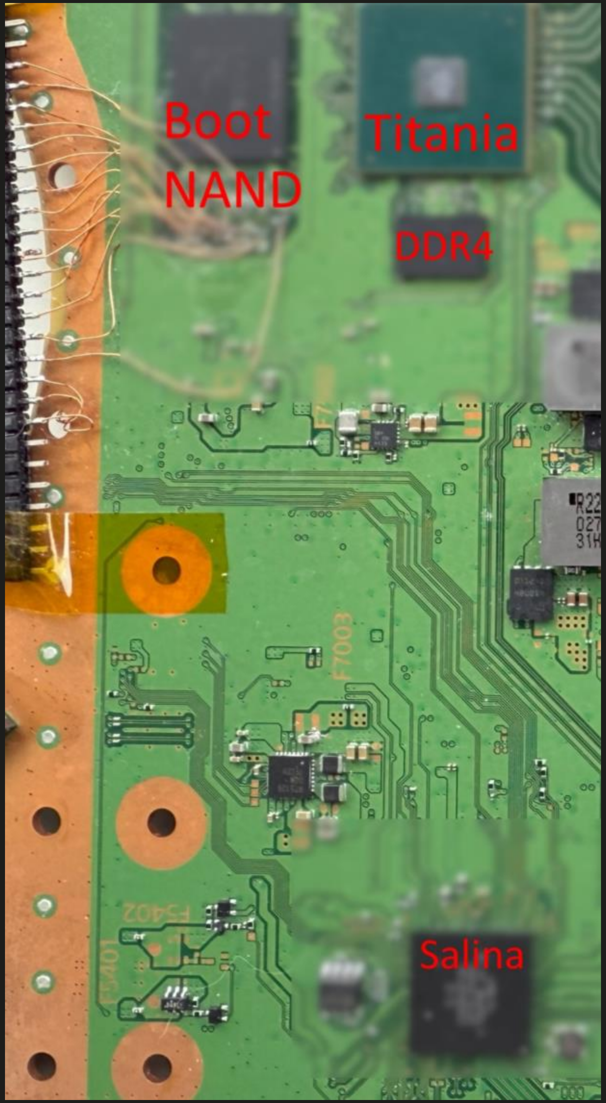
- Focus on preventing break of DRM
- Require proper license to access games/videos/etc.
  - License mainly on-disc (Blu-ray) or via PSN (tied to user account)
  - All license types for a given title contain same key
  - Title data encrypted at rest and integrity checked every load
- Protect against end-user-initiated attacks
  - Mudkips should not be possible
- Partially-compromised system should not enable piracy

Disclaimer: This talk outlines methods to start research on the platform – no DRM bypasses here, sorry!

DUT



DUT



- Boot NAND: Chip containing second-stage firmware for Titania.
  - “Boot” in the sense Titania boots from it.
- DDR4: smaller chip below Titania. 512Mb on retail consoles.
  - Main SoC boots from DDR4.
- Chip to left of Salina: sflash for Salina firmware and unprotected storage.

Boot Flow - Simplified

Salina boots when console plugged in – always on

Salina boots Titania in EFC mode

- Titania secure boots into EFC firmware loaded from NAND

Salina has Titania load data for main SoC boot into DDR4

Salina boots main SoC, indicating location in DDR4

Main SoC performs secure boot

- ROM -> Secure Loader -> ABL / Secure Kernel -> Secure Modules -> ...

Salina – Reasons to Hack

Controls power, clocks, resets on the board

Makes automation of the board easy

- Think of it like a Baseboard Management Controller (BMC)

Serves as entry point to hack anything else on the board

Salina – External Interfaces

sflash

UART (UCMD)

SPI (to Floyd)

SPI (ICC)

USB/PCIe (to Wi-Fi module)

i2c (misc. peripherals)

...

Salina – UART Bug

UCMD UART protocol is ASCII line-buffer based.

RX byte handler parses bytes as they come to handle low-level events (backspace, NAK, end of line, etc.), and output "sanitized" lines of text to upper level.

When encountering an invalid char, the parser advances the output pointer without increasing the `num_received` variable, allowing constrained write off the end of destination buffer.

- Invalid = `[0,0x1f] or [0x80,0xff] and not (0xa or 0xd)`

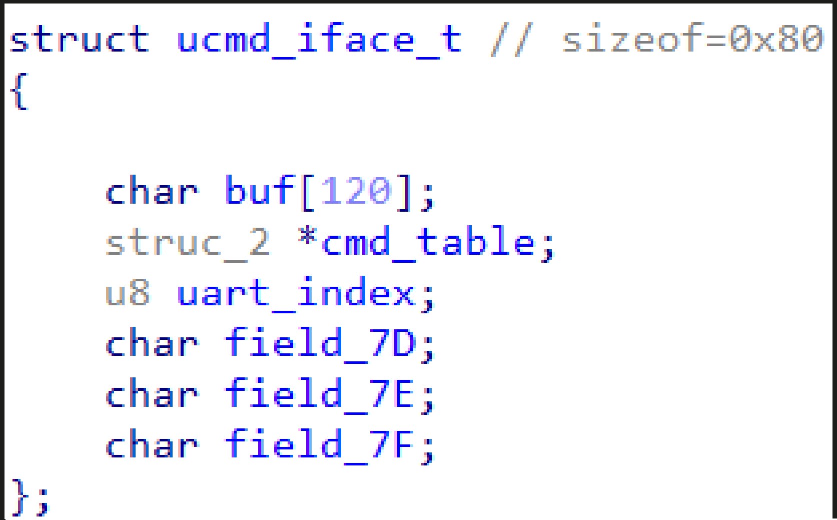
Salina – UART Bug

Struct layout is lucky!

Exploit method:

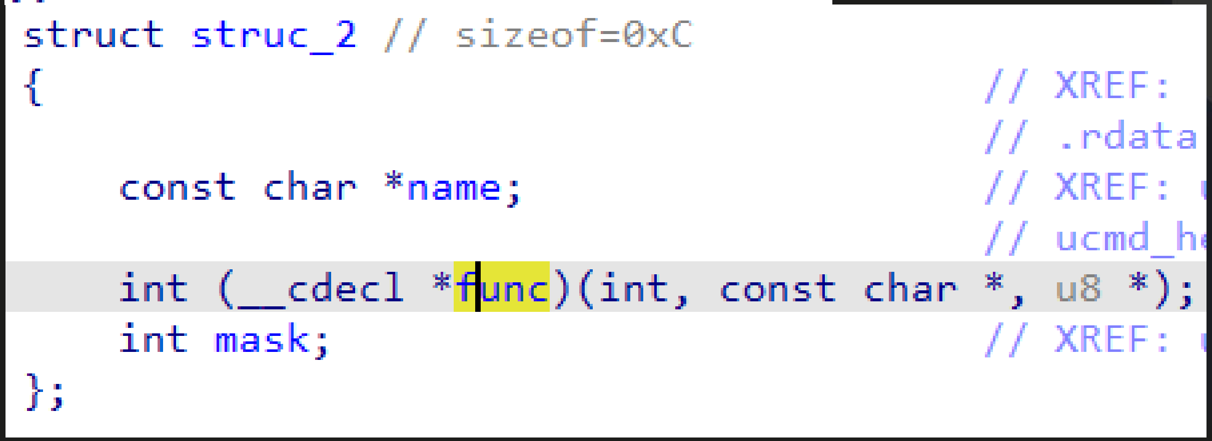
- Place own "struc_2" at known addr
- Overwrite ctx->cmd_table to point there
    - (Some constraints of values)
- Invoke UCMD with "name" given by us
- -> code exec 🥳

...but how does data reach this parser from UART?



```
struct ucmd_iface_t // sizeof=0x80
{

    char buf[120];
    struc_2 *cmd_table;
    u8 uart_index;
    char field_7D;
    char field_7E;
    char field_7F;
};
```



```
struct struc_2 // sizeof=0xC
{                                                     // XREF:
                                                      // .rdata
    const char *name;                                 // XREF:
                                                      // ucmd_h
    int (__cdecl *func)(int, const char *, u8 *);
    int mask;                                         // XREF:
};
```

Salina – Journey of UART RX byte

uart_irq is configured to handle rx + tx UART events.

On RX, uart_irq enters critsec and pushes bytes into 160-byte ring buffer until uart RX FIFO is empty. If ring buffer gets full, bytes are dropped. An evtflg is set upon each byte recv'd.

uart_rx_task does approximately the following:

- `len = 0;`
- `while (!eol) { evtflg_wait(); eol = uart_recv(ctx->buf, &len, sizeof(ctx->buf)); }`
- `ucmd_dispatch(ctx->buf);`

uart_recv is where the bug lives (ptr into ctx->buf incremented without incrementing `len`, so `len < sizeof(ctx->buf)` remains true).

Salina – UART Observations

RX ring buffer and UCMD ctx->buf are different sizes (160 vs 120).

RX ring buffer pushes many bytes under critsec, but pops 1 at a time.

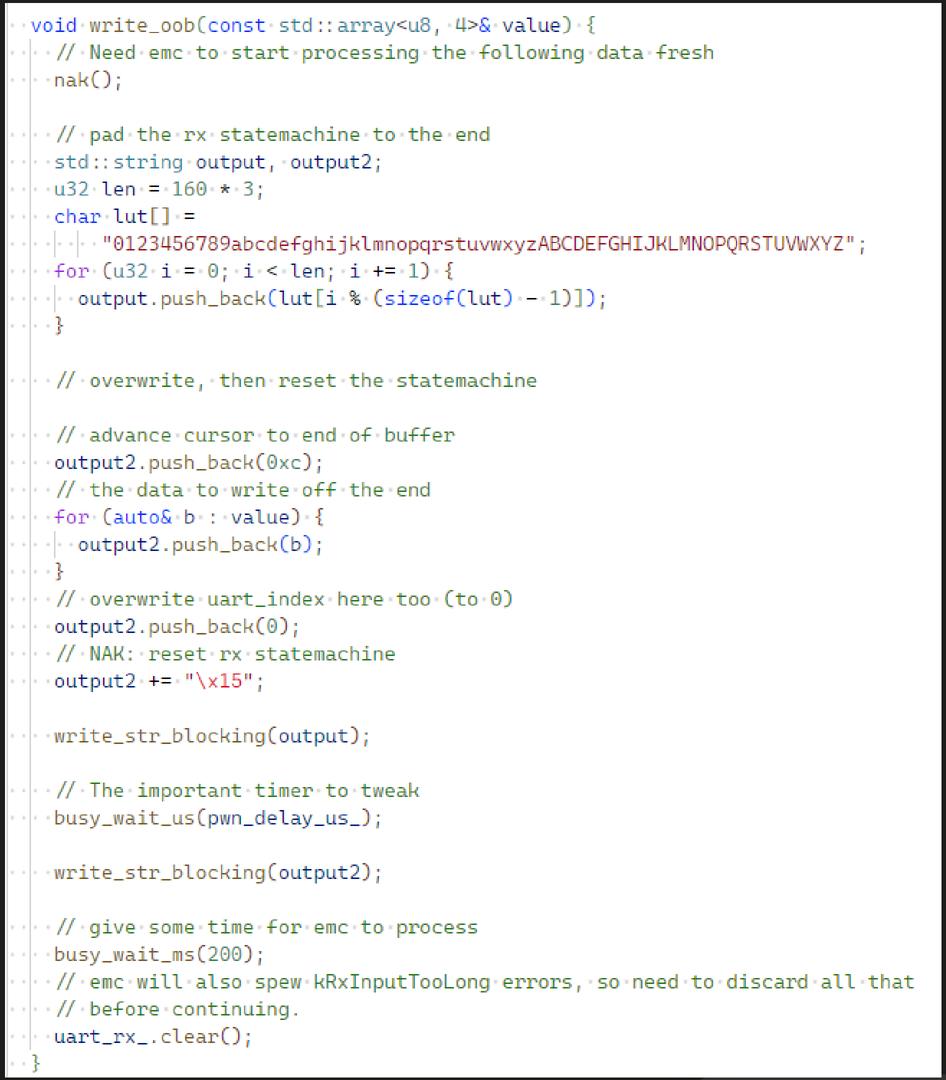
If line rate is fast enough / RX FIFO deep enough, can fill entire RX ring buffer atomically.

- Difficult in practice

Modified exploit strategy:

- Use NAK and filler bytes to manipulate filled size of ctx->buf, so last write (off the end) can be done atomically.
- Trigger OOB write multiple times if data to write contains valid characters.

Salina – Exploit Strategy



```
void write_oob(const std::array<u8, 4>& value) {
  // Need emc to start processing the following data fresh
  nak();

  // pad the rx statemachine to the end
  std::string output, output2;
  u32 len = 160 * 3;
  char lut[] =
      "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
  for (u32 i = 0; i < len; i += 1) {
    output.push_back(lut[i % (sizeof(lut) - 1)]);
  }

  // overwrite, then reset the statemachine

  // advance cursor to end of buffer
  output2.push_back(0xc);
  // the data to write off the end
  for (auto& b : value) {
    output2.push_back(b);
  }
  // overwrite uart_index here too (to 0)
  output2.push_back(0);
  // NAK: reset rx statemachine
  output2 += "\x15";

  write_str_blocking(output);

  // The important timer to tweak
  busy_wait_us(pwn_delay_us_);

  write_str_blocking(output2);

  // give some time for emc to process
  busy_wait_ms(200);
  // emc will also spew kRxInputTooLong errors, so need to discard all that
  // before continuing.
  uart_rx_.clear();
}
```

Care is taken to ensure target UART RX state machine reaches correct state before trying to trigger OOB write.

Need to accommodate errors generated on UART so UART can be used later.

Titania - Overview

Contains:

- 4 ARM Cortex-R5 cores (EFC)
- 1 ARM Cortex-A7 core (EAP)
- 1 ARM Cortex-M core (BCM)

Boots EFC or EAP based on external strap (gpio set by EMC).

EAP: low-power activities during console sleep.

EFC: high-throughput NAND SSD interface.

First hop from Main SoC to (almost?) all other devices.

Titania - Background

Based on a Marvell NVMe controller.

- Found https://github.com/johnnyplds123/ep3-1
- Sources for related Marvell device.
- Fully exploited older version of similar NVMe controller (side quest ✅).

Standard NVMe controller, but (at time of PS5 release) bleeding-edge.

Works in tandem with MP4 on main SoC to provide game storage.

Stores early-boot firmware and blobs in special region of NAND.

Titania – Reasons to Hack

Controls content and interfaces exposed to main SoC during boot.

Access raw NAND content (old bootloader versions, manufacturing information, etc.)

Note: On PS4, accessing “previous version” of firmware allowed access to manufacturing firmware, which contained interesting stuff. 🧐

Titania – External Interfaces

NAND

DDR4

UART

SPI

PCIe

...

Titania – EAP

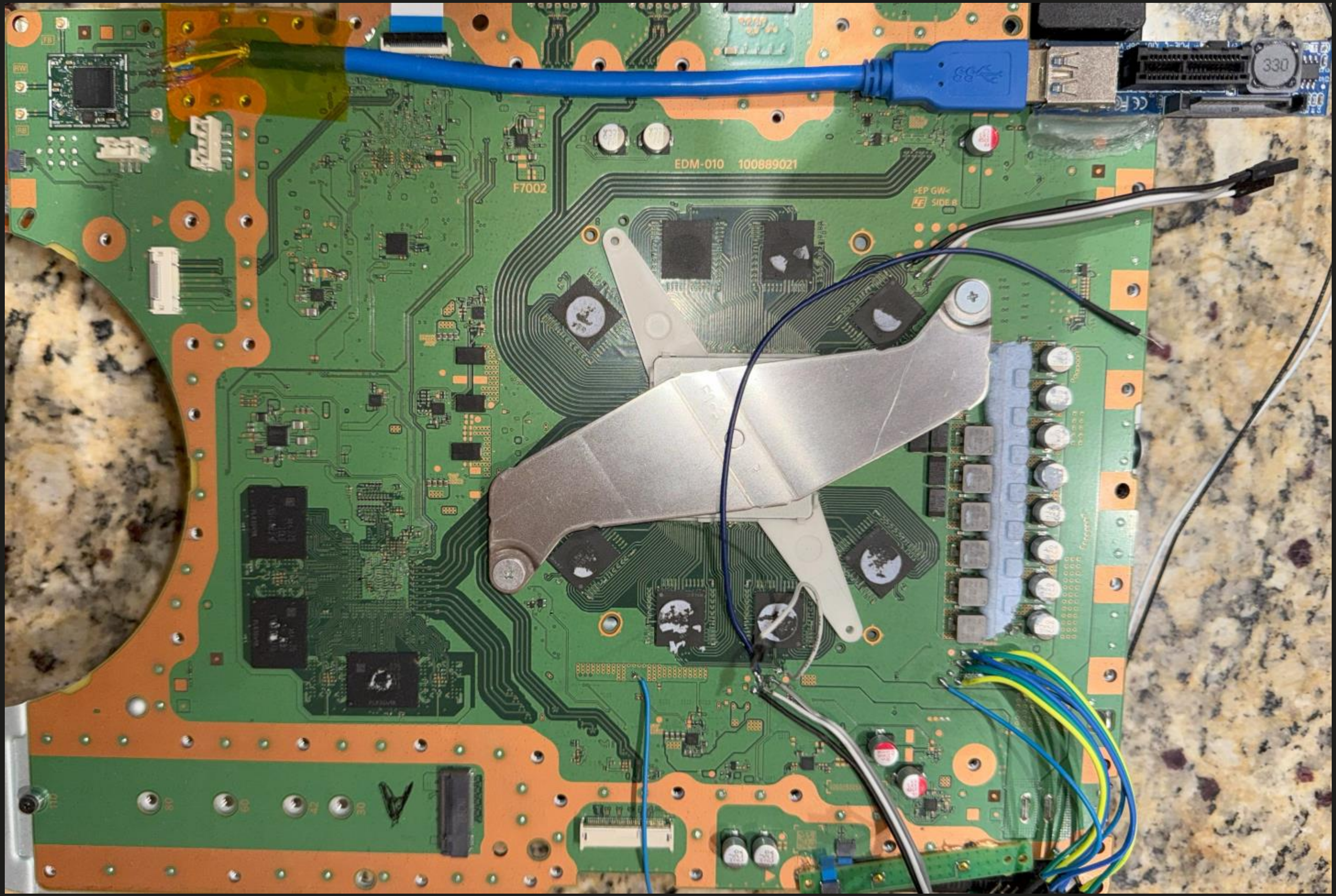
Initial method

- Replace Bluetooth/Wi-Fi module on board
- Use PCIe DMA to overwrite EAP FreeBSD kernel code in DDR4
- Perks: Doesn't require any existing foothold on board
- Issues: Requires fancy hardware, timing and cache complications

Titania – EAP

Initial method

Different DUT (this one boots 😅)

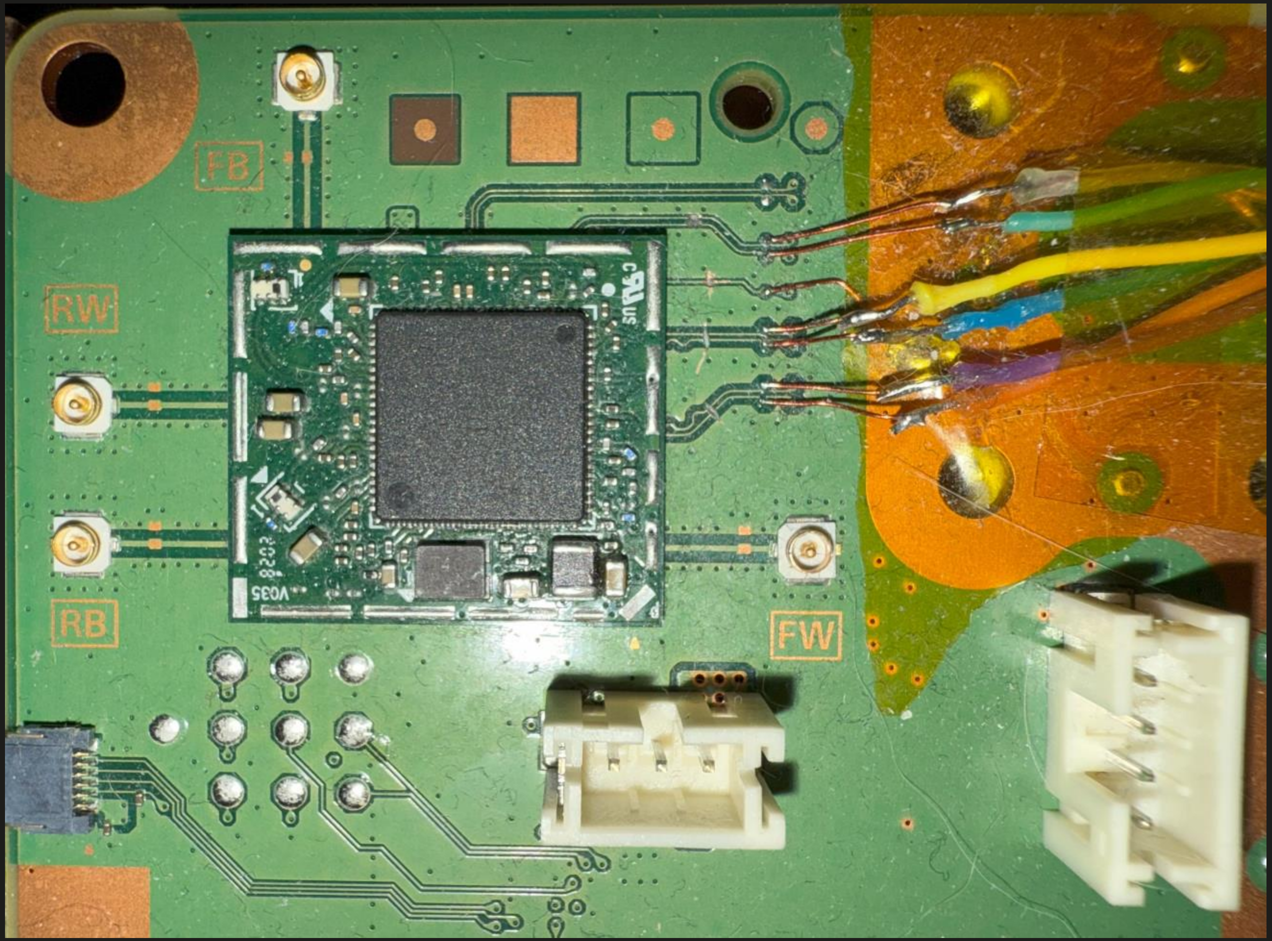


Titania – EAP

Initial method

Different DUT (this one boots 😅)

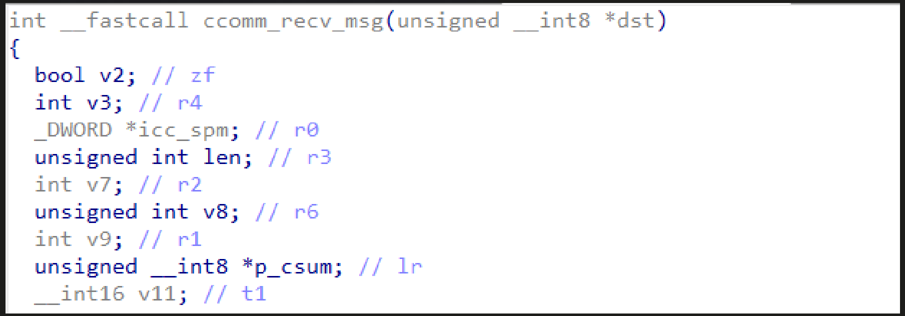
More proof PCIe is tolerant!



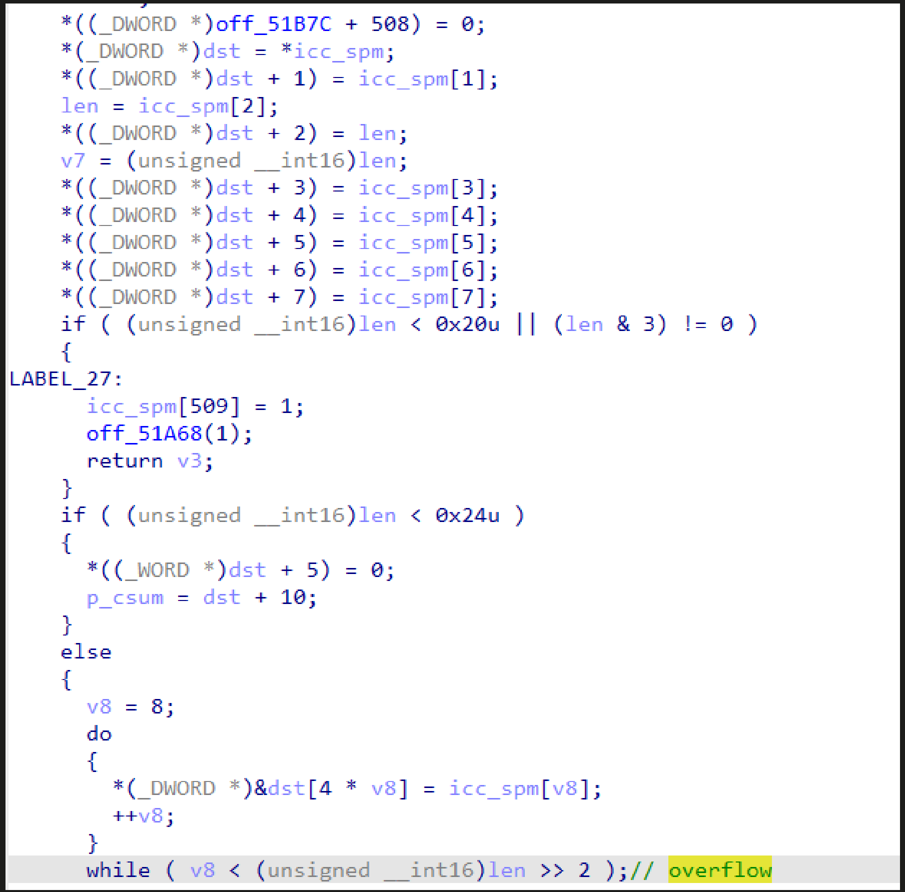
Titania – EAP

Refined method

- Exploit stack buffer overflow in eap_kbl ICC code
- There's a stack cookie...which is constant value 🧠
- Requires control of Salina or ICC mitm
- Perks: Doesn't require PCIe hardware, very reliable



```
int __fastcall ccomm_recv_msg(unsigned __int8 *dst)
{
  bool v2; // zf
  int v3; // r4
  _DWORD *icc_spm; // r0
  unsigned int len; // r3
  int v7; // r2
  unsigned int v8; // r6
  int v9; // r1
  unsigned __int8 *p_csum; // lr
  __int16 v11; // t1
```

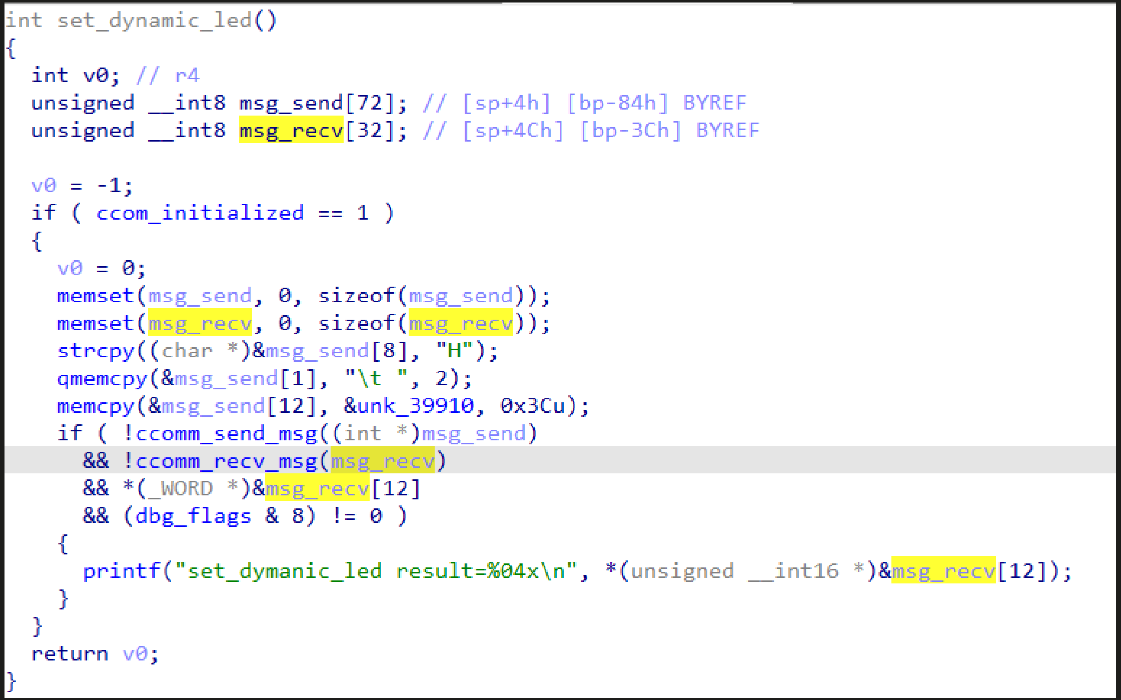


```
    *((_DWORD *)off_51B7C + 508) = 0;
    *(_DWORD *)dst = *icc_spm;
    *((_DWORD *)dst + 1) = icc_spm[1];
    len = icc_spm[2];
    *((_DWORD *)dst + 2) = len;
    v7 = (unsigned __int16)len;
    *((_DWORD *)dst + 3) = icc_spm[3];
    *((_DWORD *)dst + 4) = icc_spm[4];
    *((_DWORD *)dst + 5) = icc_spm[5];
    *((_DWORD *)dst + 6) = icc_spm[6];
    *((_DWORD *)dst + 7) = icc_spm[7];
    if ( (unsigned __int16)len < 0x20u || (len & 3) != 0 )
    {
LABEL_27:
      icc_spm[509] = 1;
      off_51A68(1);
      return v3;
    }
    if ( (unsigned __int16)len < 0x24u )
    {
      *((_WORD *)dst + 5) = 0;
      p_csum = dst + 10;
    }
    else
    {
      v8 = 8;
      do
      {
        *(_DWORD *)&dst[4 * v8] = icc_spm[v8];
        ++v8;
      }
      while ( v8 < (unsigned __int16)len >> 2 );// overflow
```

Titania – EAP

Refined method

- Exploit stack buffer overflow in eap_kbl ICC code
- There's a stack cookie...which is constant value 🧠
- Requires control of Salina or ICC mitm
- Perks: Doesn't require PCIe hardware, very reliable



```
int set_dynamic_led()
{
  int v0; // r4
  unsigned __int8 msg_send[72]; // [sp+4h] [bp-84h] BYREF
  unsigned __int8 msg_recv[32]; // [sp+4Ch] [bp-3Ch] BYREF

  v0 = -1;
  if ( ccom_initialized == 1 )
  {
    v0 = 0;
    memset(msg_send, 0, sizeof(msg_send));
    memset(msg_recv, 0, sizeof(msg_recv));
    strcpy((char *)&msg_send[8], "H");
    qmemcpy(&msg_send[1], "\t ", 2);
    memcpy(&msg_send[12], &unk_39910, 0x3Cu);
    if ( !ccomm_send_msg((int *)msg_send)
      && !ccomm_recv_msg(msg_recv)
      && *(_WORD *)&msg_recv[12]
      && (dbg_flags & 8) != 0 )
    {
      printf("set_dymanic_led result=%04x\n", *(unsigned __int16 *)&msg_recv[12]);
    }
  }
  return v0;
}
```

Titania – EFC

Command sent from Salina to EFC are handled in firmware, even commands during early boot (of EFC).

Initial command to EFC includes DDR4 geometry, which gets fed to DRAM controller hardware.

EFC uses special protection hardware to block external access to ranges of memories which could compromise integrity of itself.

By sending too-large DDR4 geometry, we effectively create larger aperture that gets decoded to DRAM but aliases the lower address range.

- -> Bypass the protection of DRAM ranges.
- -> Overwrite EFC firmware code in DRAM while it’s running.
- Cache issues need a workaround, but it’s doable!

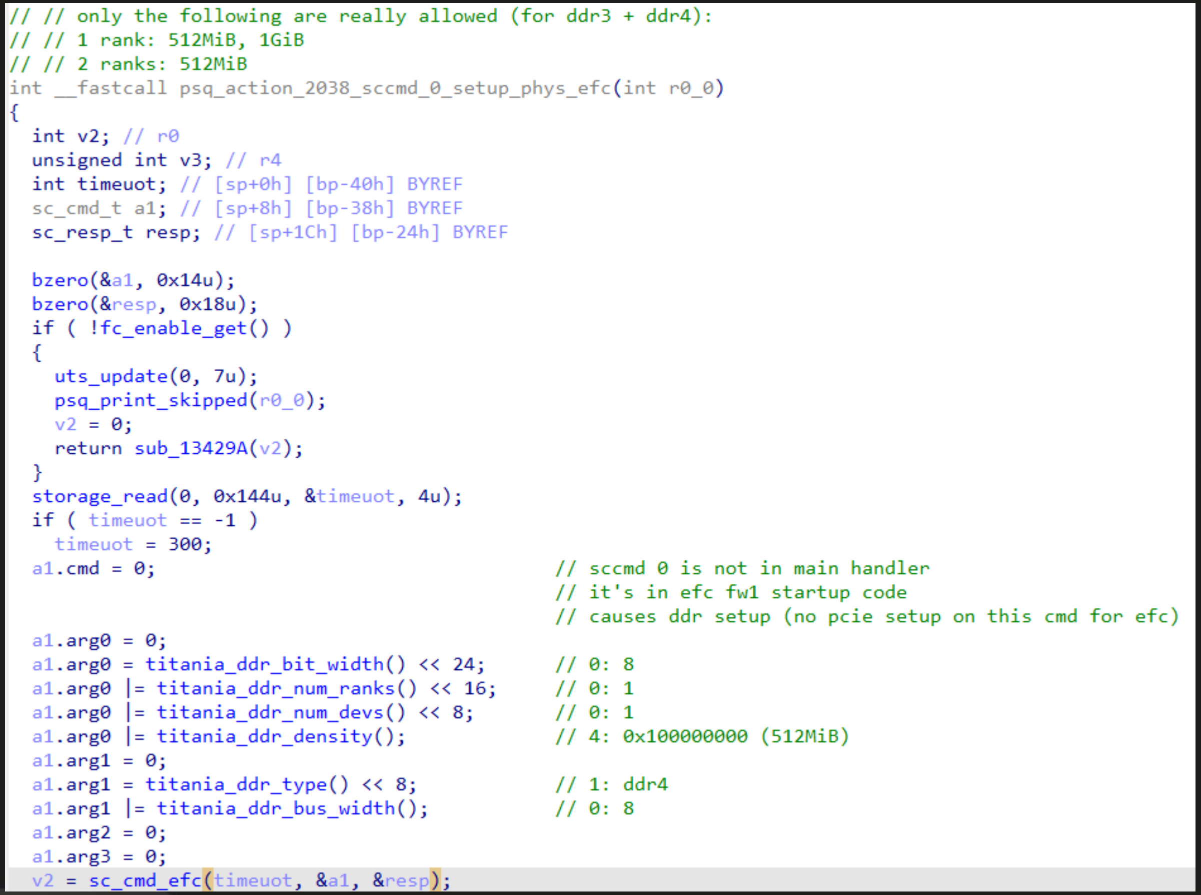
Titania – Exploit Implementations

Exploits can be implemented by injecting hooks into EMC code.

Could also be implemented by physical mitm / bus tampering, but having control of EMC makes it easy!

Exploit:

- Patches `titania_ddr_density`
- Coerces EFC to exec some code which won't be in cache and has been overwritten.
- Overwrite occurs over PCIe, but all transparent to us since we're on EMC 😎



```
// // only the following are really allowed (for ddr3 + ddr4):
// // 1 rank: 512MiB, 1GiB
// // 2 ranks: 512MiB
int __fastcall psq_action_2038_sccmd_0_setup_phys_efc(int r0_0)
{
  int v2; // r0
  unsigned int v3; // r4
  int timeuot; // [sp+0h] [bp-40h] BYREF
  sc_cmd_t a1; // [sp+8h] [bp-38h] BYREF
  sc_resp_t resp; // [sp+1Ch] [bp-24h] BYREF

  bzero(&a1, 0x14u);
  bzero(&resp, 0x18u);
  if ( !fc_enable_get() )
  {
    uts_update(0, 7u);
    psq_print_skipped(r0_0);
    v2 = 0;
    return sub_13429A(v2);
  }
  storage_read(0, 0x144u, &timeuot, 4u);
  if ( timeuot == -1 )
    timeuot = 300;
  a1.cmd = 0;                                    // sccmd 0 is not in main handler
                                                 // it's in efc fw1 startup code
                                                 // causes ddr setup (no pcie setup on this cmd for efc)

  a1.arg0 = 0;
  a1.arg0 = titania_ddr_bit_width() << 24;       // 0: 8
  a1.arg0 |= titania_ddr_num_ranks() << 16;      // 0: 1
  a1.arg0 |= titania_ddr_num_devs() << 8;        // 0: 1
  a1.arg0 |= titania_ddr_density();              // 4: 0x100000000 (512MiB)
  a1.arg1 = 0;
  a1.arg1 = titania_ddr_type() << 8;             // 1: ddr4
  a1.arg1 |= titania_ddr_bus_width();            // 0: 8
  a1.arg2 = 0;
  a1.arg3 = 0;
  v2 = sc_cmd_efc(timeuot, &a1, &resp);
```

Putting It All Together

Everything is wrapped into RPi Pico firmware and python scripts.

Interfaces with various PS5 UARTs and embeds the Salina exploit in Pico firmware (for better determinism).

Easily inject code into EMC, EFC, EAP and poke at things :) 🐍

https://github.com/symbrkrs/ps5-uart

Loose Ends

Titania

- Extract secure boot keys (offline decrypt of EFC/EAP KBL)
- Pwn BMC
- Reverse physical NAND layout of boot data

Reverse software-level structure of NAND

- Recover old versions of bootloaders, etc.