Crashdump 调试工具
使用指南

版本号: 1.7
发布日期: 2023.3.6

版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.4.14	AWA1831	初始版本
1.1	2021.4.21	XAA0193	对文件结构进行了调整
1.2	2022.1.6	XAA0193	1. 对 tigerdumpV1.1 及以上版本 dump 的数据无法解析的问题进行说明；2. 对文档的章节进行标记。
1.3	2022.8.1	XAA0193	1. 增加了 3.1.2 章节；2. 将其他文件中的案例转移到这里。
1.4	2022.9.20	AWA1881	更新 5.15 内核 crash 工具使用方法、编译调试方法以及常见 FAQ。
1.5	2022.11.3	AWA1881	1. 新增 5.10 内核相关说明；2. 为文档中的图片添加名称；3. 更正部分描述错误或不准确的问题。
1.6	2022.12.5	AWA2099	增加了 4.4.10 章节。
1.7	2023.3.6	AWA1881	整理合并 lichee/aw-software-develop-basis-doc 仓库《Crashdump_ 使用指南》文档。

目　录

1 概述 1
1.1 编写目的 1
1.2 适用范围 1
1.3 适用人员 1

2 系统介绍 2
2.1 系统简介 2
2.2 使用配置 2
2.2.1 menuconfig 配置 2
2.2.2 dts 配置 2

3 crash dump 数据步骤介绍 3
3.1 工具配置 3
3.1.1 使用 TigerDump 进行 dump 3
3.1.2 decrypt 工具使用介绍 6
3.1.3 使用 PhoenixSuit 进行 dump 6
3.1.4 使用 csat 进行 dump 9
3.2 内存转存说明 10
3.2.1 TigerDump dump 现象 10
3.2.2 PhoenixSuit dump 现象 10

4 Crash 工具使用方法和分析 13
4.1 CrashDump 分析所需资源 13
4.2 解析内存镜像 13
4.3 常见命令使用 14
4.4 应用场景案例 15
4.4.1 访问非法地址 15
4.4.2 OOM 16
4.4.3 内核链表信息被破坏 17
4.4.4 内核指针被破坏 24
4.4.5 指针访问权限非法 29
4.4.6 访问内核地址报错 31
4.4.7 栈指针出错 36
4.4.8 Workqueue 野指针 42
4.4.9 分支跳转错误 49
4.4.10 CPU 调频过程中死机 52

5 注意事项 FAQ 60
5.1 WARNING: cannot access vmalloc’d module memory 60

5.2 crash: cannot determine page size 60
5.3 crash_arm64: read error: kernel virtual address: ffffffc0087cf580 type: “kernel_config_data” 60
5.4 WARNING: could not find MAGIC_START! 61
5.5 crash 工具编译方法 63
5.6 crash 工具调试方法 63
5.7 crash 工具 GKI 固件支持 64

插 图

图 3-1 TigerDump 安装 . . . 3
图 3-2 TigerDump 配置 . . . 4
图 3-3 TigerDump 配置 . . . 5
图 3-4 decrypt-user-guide . . . 6
图 3-5 PhonenixSuit config dump.jpg . . . 6
图 3-6 PhoenixSuit v1.12 版本界面 . . . 7
图 3-7 PhoenixSuit v1.17 版本界面 . . . 8
图 3-8 PhoenixSuit 调试界面 . . . 9
图 3-9 TigerDump 过程 . . . 10
图 3-10 PhoenixSuit dump 过程 . . . 11
图 3-11 PhoenixSuit 调试窗口开始 dump . . . 12
图 3-12 PhoenixSuit 调试窗口 dump 完成 . . . 12
图 4-1 访问非法地址死机现场 . . . 15
图 4-2 访问非法地址定位异常方法 . . . 15
图 4-3 OOM crash 现场 . . . 16
图 4-4 报错详细信息 . . . 16
图 4-5 进程的内存占用信息 . . . 17
图 4-6 内核链表信息被破坏现场 . . . 18
图 4-7 crash PC 指针定位源码位置 . . . 18
图 4-8 异常发生时 PC 指针源码 . . . 19
图 4-9 crash LR 指针定位源码位置 . . . 19
图 4-10 异常发生时 LR 指针源码 . . . 20
图 4-11 查看 lock_class 中的 name . . . 20
图 4-12 all_lock_classes 文件 . . . 21
图 4-13 找到数据结构对应 name 的地址 . . . 21
图 4-14 查看链表上一个数据成员指针 . . . 22
图 4-15 正常情况下的结构体 . . . 22
图 4-16 获取正常偏移量时的结构体 . . . 23
图 4-17 取 prev/next 指针和 name 字符串成功 . . . 23
图 4-18 内核指针破坏 crash 现场 . . . 24
图 4-19 内核指针破坏 PC 指针反汇编 . . . 24
图 4-20 内核指针破坏 PC 指针对应源码 . . . 25
图 4-21 内核寻址到非法地址汇编代码 . . . 26
图 4-22 内核启动后 memory mapping . . . 26
图 4-23 地址翻转引起非法寻址 . . . 28
图 4-24 指针访问权限非法 crash 现场 . . . 29
图 4-25 地址访问非法 pc 指针 . . . 29
图 4-26 系统启动时内核地址范围 . . . 30
图 4-27 读取死机时 SP 指针 . . . 30

图 4-28 查看 Log 报错对应的内核源码 31
图 4-29 访问内核地址 crash 现场 32
图 4-30 访问内核地址 crash 时 pc 指针反汇编 32
图 4-31 访问内核地址 crash 内核 memory layout 33
图 4-32 读取引发 fault 的地址 33
图 4-33 打印 init_mm 结构体 34
图 4-34 kenel_memory_table 文件 35
图 4-35 根据 fault 地址找到一级页目录 35
图 4-36 kenel_memory_pmd_c_table 文件 35
图 4-37 根据 fault 地址找到二级页目录 35
图 4-38 根据 fault 地址找到三级页目录 36
图 4-39 kenel_memory_pte_139_table 文件 36
图 4-40 栈指针出错现场 37
图 4-41 栈指针出错 PC 指针反汇编 37
图 4-42 栈指针出错 PC 指针对应源码 38
图 4-43 kernel_entry 宏展开 39
图 4-44 栈指针出错 LP 指针反汇编 39
图 4-45 栈指针出错 LP 指针对应源码 40
图 4-46 栈指针出错死机指令 40
图 4-47 cpu_switch_to 源码 41
图 4-48 task2 文件 41
图 4-49 task2 中的 SP 指针 42
图 4-50 Workqueue 野指针 crash 现场 43
图 4-51 Workqueue 野指针 PC 指针反汇编 43
图 4-52 Workqueue 野指针 LR 指针反汇编 43
图 4-53 Workqueue 野指针 LR 指针源码 44
图 4-54 process_one_work 反汇编 44
图 4-55 get_work_pwq 函数 45
图 4-56 process_one_work 反汇编查看 x0/x1 45
图 4-57 查看 struct work_struct 结构体的值 46
图 4-58 INIT_WORK 定义 47
图 4-59 WORK_DATA_INIT 定义 47
图 4-60 WORK_STRUCT_NO_POOL 定义 47
图 4-61 异步调用 sunxi_vbus_det_work 48
图 4-62 insert_work 函数 48
图 4-63 分支跳转错误 crash 现场 49
图 4-64 分支跳转错误 PC 指针反汇编 49
图 4-65 分支跳转错误 LR 指针反汇编 50
图 4-66 分支跳转错误 LR 指针对应源码 50
图 4-67 对照 LR 指针函数反汇编 50
图 4-68 查看 page 结构体的值 1 51
图 4-69 查看 page 结构体的值 2 52
图 4-70 CPSR 寄存器分布 52

ALLWINNER®

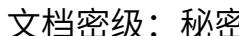


图 4-71 当前 CPU 正在运行的进程 53
图 4-72 cpufreq_policy 结构体成员的偏移 56
图 4-73 cpufreq_policy 结构体数据 57
图 4-74 sugov_policy 结构体数据 58

版权所有 © 珠海全志科技股份有限公司。保留一切权利

1 概述

1.1 编写目的

介绍全志平台上 CrashDump 的使用以及调试方法，为 CrashDump 系统的使用和开发提供参考。

1.2 适用范围

适用于 CrashDump 配套的 linux-4.9、linux-5.4、linux-5.10、linux-5.15 内核平台。目前支持的平台有：

内核版本	IC 平台
linux-4.9	A100
linux-4.9	A133
linux-4.9	T509
linux-5.4	A100
linux-5.4	A133
linux-5.10	A40i/A40i-H/A40i-C
linux-5.10	T3/T3-C/T3Pro
linux-5.15	A133
linux-5.15	A523

1.3 适用人员

CrashDump 的开发/使用/维护人员。

2 系统介绍

2.1 系统简介

CrashDump 是全志平台的一个分析内核崩溃的工具。当 Linux 系统**内核发生崩溃的时候**，可以通 KEXEC+KDUMP 等方式收集内核崩溃之前的内存，生成一个转储文件 vmcore。内核开发者通过分析该 vmcore 文件就可以诊断出内核崩溃的原因，从而进行操作系统的代码改进。主要用于分析以下问题场景：

- Android 系统内存泄露导致的卡顿、ANR 重启问题
- Linux 内核软件 Panic Oops 崩溃问题、死锁问题
- Linux 内核态 Memory overflow 内存溢出、OOM 内存分配失败问题

2.2 使用配置

2.2.1 menuconfig 配置

在 linux-4.9 或者中 linux-5.4 中，需要进入到内核目录里面，执行 make ARCH=arm64（32 位系统选择 arm）menuconfig，配置以下两个选项：

```
CONFIG_SUNXI_DUMP=y
CONFIG_PANIC_TIMEOUT=0
```

在 linux-5.10 或者 linux-5.15 中，需要在内核 menuconfig 配置以下选项：

```
CONFIG_AW_CRASHDUMP=y
CONFIG_PANIC_TIMEOUT=0
```

2.2.2 dts 配置

crashdump 功能不需要通过 dts 进行配置。

3 crash dump 数据步骤介绍

3.1 工具配置

CrashDump 需要配合全志的 dump 工具进行使用，现在**我们推荐使用 TigerDump 工具**，一些版本很老的 PhoenixSuit 软件（1.12 版本以前）也可以实现这个数据转储的功能，下面对这两种方式的使用进行介绍：

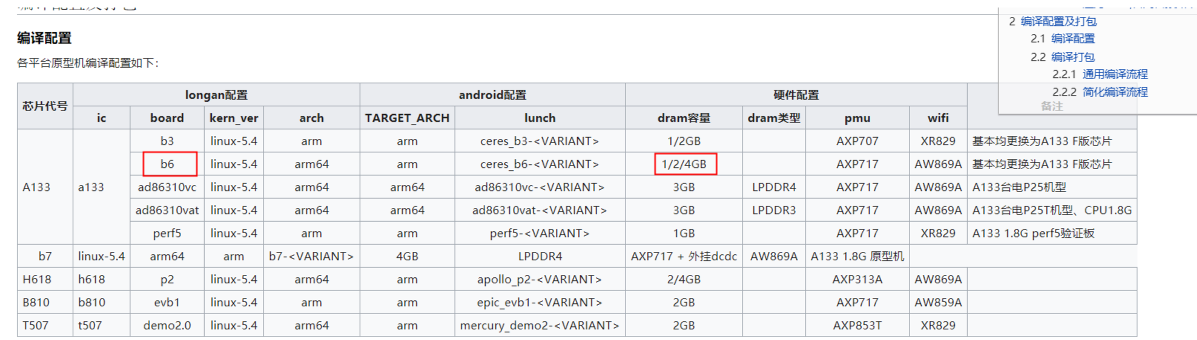
3.1.1 使用 TigerDump 进行 dump

1、TigerDump 的安装可以从 APST 软件中进行安装：

图 3-1: TigerDump 安装

2、安装之后按打开软件，按下图选择 DUMP DDR（大小选项由方案内存大小决定）。**注意这点很重要，很多获取的 crash 数据无法正确解析都是此原因导致的，你必须清楚地知道你当前使用的**

版型的 DDR 信息，并正确的选择如下配置：



2 编译配置及打包
2.1 编译配置
2.2 编译打包
2.2.1 通用编译流程
2.2.2 简化编译流程

编译配置

各平台原型机编译配置如下：

芯片代号	longan配置				android配置		硬件配置				备注
	ic	board	kern_ver	arch	TARGET_ARCH	lunch	dram容量	dram类型	pmu	wifi	
A133	a133	b3	linux-5.4	arm	arm	ceres_b3-<VARIANT>	1/2GB		AXP707	XR829	基本均更换为A133 F版芯片
		b6	linux-5.4	arm64	arm	ceres_b6-<VARIANT>	1/2/4GB		AXP717	AW869A	基本均更换为A133 F版芯片
		ad86310vc	linux-5.4	arm64	arm64	ad86310vc-<VARIANT>	3GB	LPDDR4	AXP717	AW869A	A133台电P25机型
		ad86310vat	linux-5.4	arm64	arm64	ad86310vat-<VARIANT>	3GB	LPDDR3	AXP717	AW869A	A133台电P25T机型、CPU1.8G
		perf5	linux-5.4	arm	arm	perf5-<VARIANT>	1GB		AXP717	XR829	A133 1.8G perf5验证板
b7	linux-5.4	arm64	arm	b7-<VARIANT>	4GB	LPDDR4	AXP717 + 外挂dcdc	AW869A	A133 1.8G 原型机		
H618	h618	p2	linux-5.4	arm64	arm	apollo_p2-<VARIANT>	2/4GB		AXP313A	AW869A	
B810	b810	evb1	linux-5.4	arm	arm	epic_evb1-<VARIANT>	2GB		AXP717	AW859A	
T507	t507	demo2.0	linux-5.4	arm64	arm	mercury_demo2-<VARIANT>	2GB		AXP853T	XR829	

图 3-2: TigerDump 配置

图 3-3: TigerDump 配置

3、然后在使用 usb 线连接小机，在小机发生 oops 问题系统崩溃后时，软件就可以自动 dump 数据到你指定的位置。

注意：对于使用 TigerDump V1.1 及以上版本 dump 出来的数据，还需要专门脚本 decrypt 进行解密，再通过 crash 工具解析。具体咨询 TigerDump 工具负责人。

版权所有 © 珠海全志科技股份有限公司。保留一切权利

3.1.2 decrypt 工具使用介绍

关于 crashdump 工具 TigerDump，dump ddr 加密数据对应的解密工具使用说明如下。

- 加密原因：保护 ddr 内容。
- 加密实现：TigerDump 工具 V1.1 及以上版本，dump ddr 数据时会自动完成加密。
- 解密工具介绍
- 版本：V1，环境：Linux
- 命令有 3 条，如下图所示。因工具较简单，故无使用指南。

```
fengdongxiu@AwExdroid66:~/linux-de$ ./decrypt -h
TigerDump-DecryptTool, VERSION:1
Usage: ./decrypt [options] command arguments... [command...]
        -h, --help                      Print this usage summary and exit
        en datafile endatafile          encrypt datafile to endatafile
        de endatafile dedatafile        decrypt endatafile to dedatafile
```

图 3-4: decrypt-user-guide

- ./decrypt -h：显示提示信息
- ./decrypt en 原数据文件完整路径加密数据文件完整路径
- ./decrypt de 加密数据文件完整路径解密数据文件完整路径（主要使用此命令）

3.1.3 使用 PhoenixSuit 进行 dump

- **修改 PhoenixSuit.cfg 文件**：在收集数据之前，需要先配置好 Crashdump 的采集配置（只需在执行 dump 前进行配置，不会影响固件烧录），在 PhoenixSuit 的安装目录，找到 Phoenix-Suit.cfg 文件，dump 相关的配置如下：

```
[dump]
enable =1
size = 2G  //这里表示dump 2G内存大小，配置大小看方案的内存大小决定。
```

配置项	默认值(未配置时)	值的含义	备注
path	d:\dump_dram	用户自定义 dump 数据存储路径	该值只能包含目录
size	1024M	用户自定义 dump 数据大小	值的形式：数字+g/G/M/m
start_addr	0x40000000	用户自定义要 dump 数据的 dram 起始地址	据 dram 地址有效范围配置

图 3-5: PhonenixSuit config dump.jpg

文件路径;

a.默认安装在C盘 C:\Program Files (x86)\AllwinnerTech\PhoenixSuit)
b.通过ATPS安装在D盘 D:\AllwinnerTech\APST\tools\ADFEF74D-6D16-463d-8A5B-9B1C09F96541

- **开始 dump**：在系统崩溃之后，使用 USB 连接 PhoenixSuit，注意需要在 phonixsuit 工具上选中一个固件（与小机正运行的固件相同），然后把红框里面的 Dump 打开。

图 3-6: PhoenixSuit v1.12 版本界面

- 如果是使用 1.17 版 PhoenixSuit，点击一键刷机页面的调试按钮，跳出调试小框。

版权所有 © 珠海全志科技股份有限公司。保留一切权利

图 3-7: PhoenixSuit v1.17 版本界面

- 成功后会跳出一下界面：

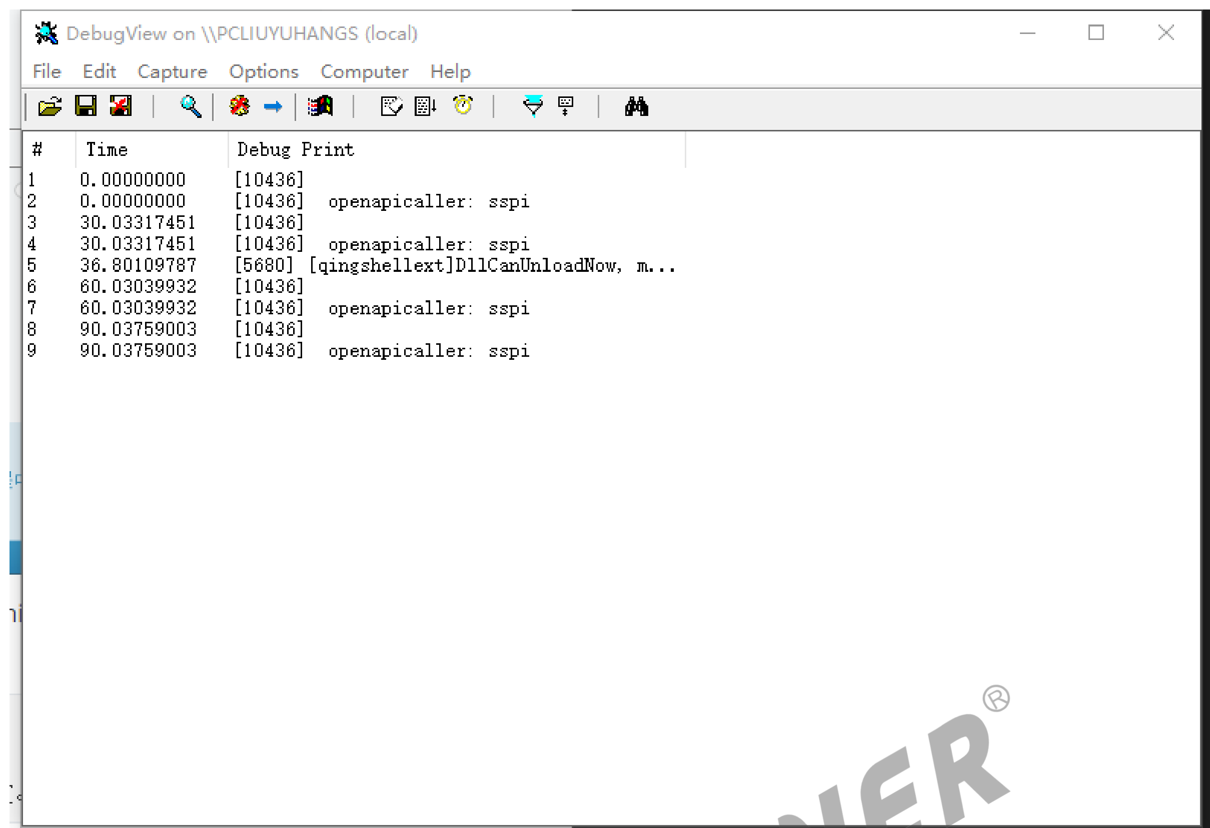


图 3-8: PhoenixSuit 调试界面

3.1.4 使用 csat 进行 dump

如果小机没有进入 crashdump 模式，就只能通过 csat 进行数据 dump 了。csat 具体使用可以参考 ARM CoreSight 文档。

- 运行 DS-5 Command Prompt 程序，输入以下命令：

```
csat
con usb
chain dev=auto clk=A
dvo 0

// 以AW1855平台dump 4GB数据为例，csat一次最大支持dump 1GB，dump数据存放在D盘
dfs 0 0x40000000 0x10000000 d:log_01
dfs 0 0x80000000 0x10000000 d:log_02
dfs 0 0xc0000000 0x10000000 d:log_03

// dump最后1GB数据，需要先修改DCU映射，具体咨询cpu硬件负责人
dmw 0 0x03010020 0x1
dfs 0 0x40000000 0x10000000 d:log_04
```

- 合并所有 dump 数据，得到完整内存镜像：

```
cat log_01 log_02 log_03 log_04 > log_whole
```

3.2 内存转存说明

按照**使用配置**完成配置后，如果内核崩溃就会进入内存转存模式。也可以**人工模拟内核崩溃**，只需执行以下指令即可：

```
echo "c" > /proc/sysrq-trigger
```

内核出现 crash 后，系统会进入 crashdump 模式，并在 log 中附带如下打印：

```
[ 103.647080] crashdump enter
```

3.2.1 TigerDump dump 现象

使用 TigerDump 则会出现如下的界面：

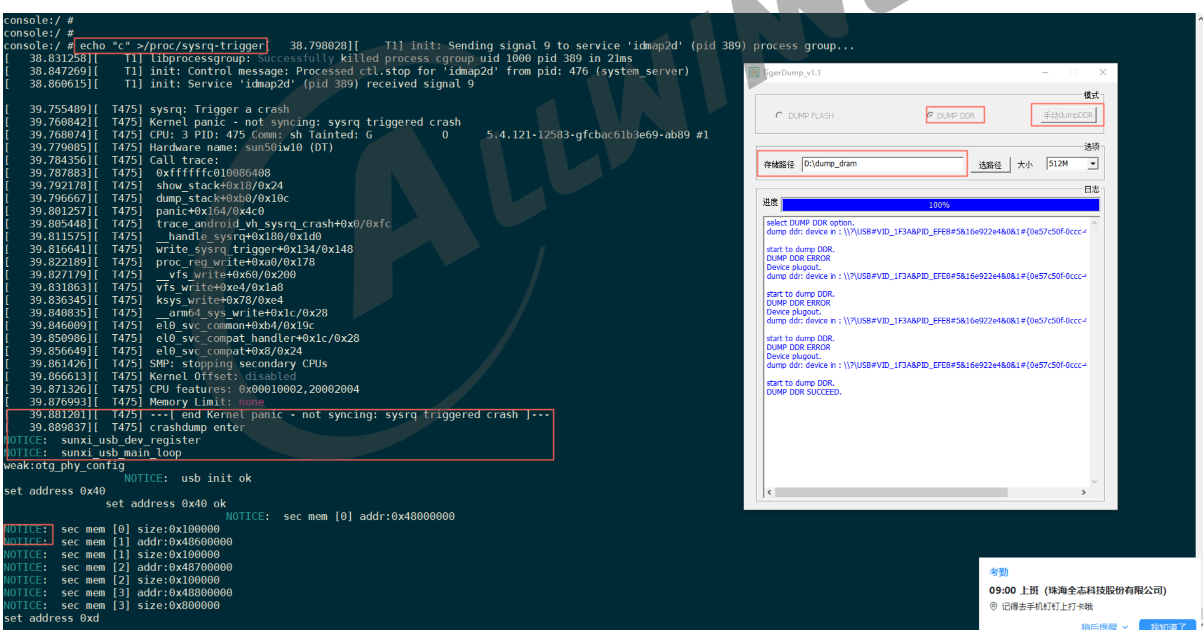


图 3-9: TigerDump 过程

3.2.2 PhoenixSuit dump 现象

用 USB 连接 PC 端和小机端，然后 PhoenixSuit 工具会开始 dump 内存。最终 dump 出来的内存镜像默认放置在：D:\dump_dram 目录。

版权所有 © 珠海全志科技股份有限公司。保留一切权利

图 3-10: PhoenixSuit dump 过程

1.17 版，调试窗口的显示如下：

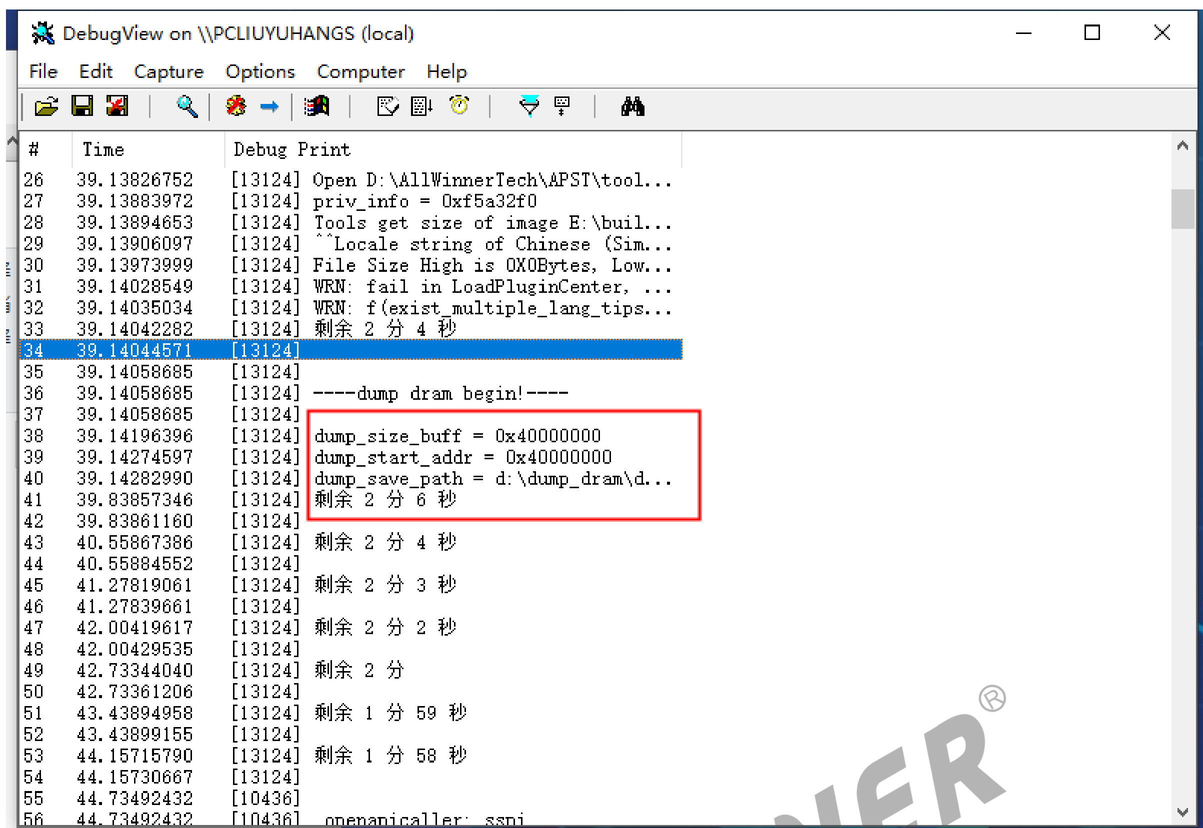


图 3-11: PhoenixSuit 调试窗口开始 dump

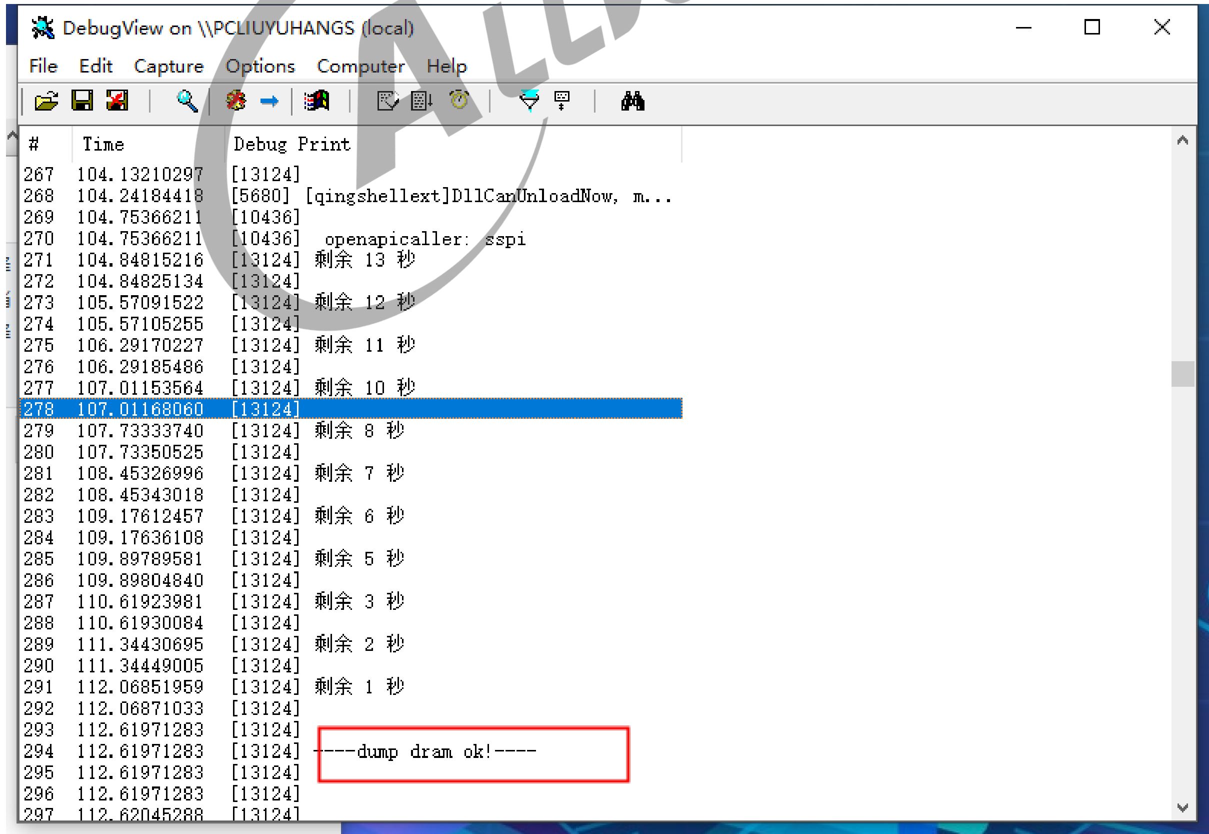


图 3-12: PhoenixSuit 调试窗口 dump 完成

4 Crash 工具使用方法和分析

4.1 CrashDump 分析所需资源

Crashdump 需要的资源如下：

1. 需要小机固件对应的内核符号表 vmlinux。

- linux-5.4 及以后放在 longan 目录下的 out/kernel/build；
- linux-4.9 放在内核目录下；

  注：linux-5.4 可以到 longon/build 目录下，执行 getvmlinux.sh 固件路径，会在该目录生成一个 output 目录，里面包含了固件的 vmlinux。

2. crash 解析程序 crash_arm64（32 位为 crash_arm），跟版本相关。

4.2 解析内存镜像

将 **crash_arm(64)，vmlinux 以及 dump 出来的内存镜像放到同一个目录下**，然后执行以下指令解析内存镜像：

linux-4.9 下使用这条指令：

```
./crash_arm64 vmlinux dram_data_20191210113324@0x40000000
```

linux-5.4 下使用这条指令：

```
./crash_arm64 vmlinux dram_data_2020514192634@0x40000000 --machdep vabits_actual=39 --machdep
        kimage_voffset=0xffffffbfd0000000
```

linux-5.10 及 linux-5.15 下使用这条指令：（注意，需要使用 8.0.1++ 版本的工具）

```
./crash_arm64 vmlinux dram_data_202292093322@0x40000000 --machdep vabits_actual=39 --machdep kimage_voffset
        =0xffffffbfc8000000 --kaslr 0x80000
```

参数的解释：

- vmlinux：当前固件对应的 vmlinux

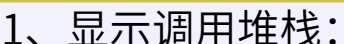
- dram_data_2020514192634： dump 出来的数据文件名，或者通过 decrypt 工具进行解密后的文件名；
- @0x40000000： 自定义的 dram 的起始地址（由 Memory Map Spec 指定）；
- vabits_actual=39：设置访问的位宽，通过 CONFIG_ARM64_VA_BITS_39 进行配置；
- machdep kimage_voffset=0xffffffbfd0000000：指定内核镜像的偏移地址，不便于计算，可在内核中添加打印获取其值；
- -kaslr 0x80000：kaslr 特性会对内核加载地址做 relocation，使能该特性后，内核实际映射的运行时地址和链接地址是不一样的，中间差距 kaslr offset 值（crash 时 dump_kernel_offset 函数会打印具体的值）。

若编译的固件为 ARM32，则使用 **crash_arm32** 工具和以下指令进行解析：（建议使用高版本的 crash_arm32 工具进行解析）

```
./crash_arm32 vmlinux dram_data_20191210113324@0x40000000
```

4.3 常见命令使用

解析完内存镜像后会进去控制台，常见的指令如下：



```
1、显示调用堆栈：
   :crash> bt -f
    :PID: 763    TASK: ef3e2640  CPU: 2   COMMAND: "sh"
   :bt: WARNING: cannot determine starting stack frame for task ef3e2640
2、log
  显示内核dmesg信息：
3、ps
  可以看到一个任务列表，就像在一个实时系统上一样。还可以按照任务名称查看该任务的所有线程。
4、files
5、fuser
  可以看到谁正在使用的文件路径或模式
6、list/tree
  可以从内核结构见列表或基数/ rbtree
7、IRQ
  可以看到IRQ信息，一个可以看到IRQ中断亲和力
8、dis
  可以看到反汇编代码，这是检查比特翻转问题有用
9、rd/wr
  读写存储器地址
10、task
  显示任务结构
11、struct
  显示内核结构及其大小，还可以显示成员的偏移量,可以将内存地址显示为结构的指针。
12、waitq
  可以看到任务在waitq待定
13、search
  在内存范围内搜索值/字符串
14、vm
  可以看到任务的虚拟内存映射，虚拟机可以看到物理内存映射。
15、kmem
  显示内核内存页信息。
16、p
```

ALLWINNER®

打印一个表达式或变量，struct的成员

4.4 应用场景案例

4.4.1 访问非法地址

使用 dmesg 命令查看死机现场的 log。看到最后的死机现场如下所示：

```
[   44.606457] Internal error: Accessing user space memory outside uaccess.h routines: 96000045 [#1] PREEMPT SMP
[   44.617547] Modules linked in: rtl_btlpm uvcvideo videobuf2_v4l2 videobuf2_vmalloc videobuf2_memops videobuf2_core mali_kbase(O)
[   44.630527] CPU: 1 PID: 2791 Comm: cat Tainted: G           O    4.9.170 #44
[   44.638390] Hardware name: sun50iw9 (DT)
[   44.642766] task: ffffffc053ec5600 task.stack: ffffffc03a1cc000
[   44.649380] PC is at sunxi_uart_dev_info_show+0x28/0x8c
[   44.655207] LR is at dev_attr_show+0x40/0x70
[   44.659960] pc : [<ffffff80085cd4dc>] lr : [<ffffff80085e9f28>] pstate: 40400145
[   44.668210] sp : ffffffc03a1cfbd0
[   44.671899] x29: ffffffc03a1cfbf0 x28: ffffffc043e20cc0
[   44.677818] x27: ffffffc0550e34c0 x26: ffffffc03a1cfd18
[   44.683752] x25: ffffffc03a1cfeb0 x24: ffffffc077ac9ea0
[   44.689688] x23: ffffffc043d26080 x22: ffffff8008c15738
[   44.695625] x21: ffffffc070de6c80 x20: ffffffc070de6c80
[   44.701559] x19: ffffffc077ac9e90 x18: 0000000000000000
[   44.707492] x17: 0000000000000000 x16: ffffff8008252130
[   44.713418] x15: 0000000000000000 x14: 00000000b994bb73
[   44.719350] x13: 00000000ff9b25e0 x12: 00000000b9912f79
[   44.725275] x11: 0000000080080000 x10: 0000000000001000
[   44.731216] x9 : 0000000000000000 x8 : ffffffc070de7c80
[   44.737160] x7 : 0000000000000000 x6 : 000000000000003f
[   44.743091] x5 : 0000000000000040 x4 : ffffff800a543230
[   44.749021] x3 : ffffff80085cd4b4 x2 : ffffffc070de6c80
[   44.754948] x1 : 0000000000000000 x0 : ffffff800a543dc0
```

图 4-1: 访问非法地址死机现场

PC is at sunxi_uart_dev_info_show+0x28/0x8c

再用反汇编命令 dis 解析出代码对应的源码及汇编：

```
dis -l sunxi_uart_dev_info_show+0x28
```

可以看出跑飞的代码为：/home/lidaxin/AndroidQ/longan/kernel/linux-4.9/drivers/tty/serial/sunxi-uart.c：1501。对应汇编为：str wzr,[x1]，这里是要往 x1 地址内存写 0，log 信息可以看到 x1=0，往 0 地址写 0，那就是访问非法地址了。

```
crash_arm64> dis -l sunxi_uart_dev_info_show+0x28
/home/lidaxin/AndroidQ/longan/kernel/linux-4.9/drivers/tty/serial/sunxi-uart.c: 1501
0xffffff80085cd4dc <sunxi_uart_dev_info_show+40>:       str     wzr, [x1]
crash_arm64>
```

图 4-2: 访问非法地址定位异常方法

可以看出跑飞的代码为：

> /home/lidaxin/AndroidQ/longan/kernel/linux-4.9/drivers/tty/serial/sunxi-uart.c: 1501。

对应汇编为：str wzr,[x1]，这里是要往 x1 地址内存写 0，log 信息可以看到 x1=0, 往 0 地址写 0，那就是访问非法地址了。

4.4.2 OOM

通过 dmesg 查看 OOM 的报错详细信息。解析后的 panic 原因是：Out of memory.

```
      KERNEL: vmlinux
   DUMPFILES: /var/tmp/ramdump_elf_FPc4gF [temporary ELF header]
              dram_data_oom
        CPUS: 4 [OFFLINE: 3]
        DATE: Tue Dec 10 11:29:04 2019
      UPTIME: 00:01:44
LOAD AVERAGE: 10.96, 3.35, 1.17
       TASKS: 393
    NODENAME: localhost
     RELEASE: 4.9.170
     VERSION: #43 SMP PREEMPT Tue Dec 10 16:05:02 CST 2019
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 2 GB
       PANIC: "Kernel panic - not syncing: Out of memory and no killable processes..."
         PID: 3395
     COMMAND: "memtester"
        TASK: ffffffc035440080  [THREAD_INFO: ffffffc035440080]
         CPU: 1
       STATE: TASK_RUNNING (PANIC)

crash_arm64>
```

图 4-3: OOM crash 现场

再通过 dmesg 查看 OOM 的报错详细信息。内存使用情况：free 只有 11252kB,low 水位是 18024kB，此时报出 oom 为正常。anon 页面和 file 页面都很少，说明进程占用的内存已经很少。mlock 页面为 1609904kB，unevictable 页面 1609904kB 。mlock 的页面即为 unevictable 的。所以这里要看是哪个进程把内存 mlock 了。

```
[  104.266042] DMA free:11252kB min:5552kB low:18024kB high:20024kB active_anon:0kB inactive_anon:56kB active_file:1116kB inactive_file:1584kB unevictable:160
9904kB writepending:80kB present:2097152kB managed:2001696kB mlocked:1609904kB slab_reclaimable:58804kB slab_unreclaimable:162604kB kernel_stack:7056kB pageta
bles:10964kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:6860kB
[  104.266055] lowmem_reserve[]: 0 0 0
[  104.266117] DMA: 799*4kB (UMEHC) 41*8kB (UMEHC) 19*16kB (UMEHC) 6*32kB (UHC) 1*64kB (U) 2*128kB (HC) 2*256kB (H) 1*512kB (C) 0*1024kB 1*2048kB (C) 1*4096kB
 (C) = 11508kB
[  104.266120] 1343 total pagecache pages
[  104.266126] 16 pages in swap cache
[  104.266130] Swap cache stats: add 90224, delete 90208, find 1063/20043
[  104.266133] Free swap  = 1373472kB
[  104.266136] Total swap = 1501268kB
[  104.266139] 524288 pages RAM
[  104.266141] 0 pages HighMem/MovableOnly
[  104.266144] 23864 pages reserved
[  104.266147] 16384 pages cma reserved
```

图 4-4: 报错详细信息

再查看进程的内存占用信息：可以看出内存占用最高的进程是 memtester 进程。

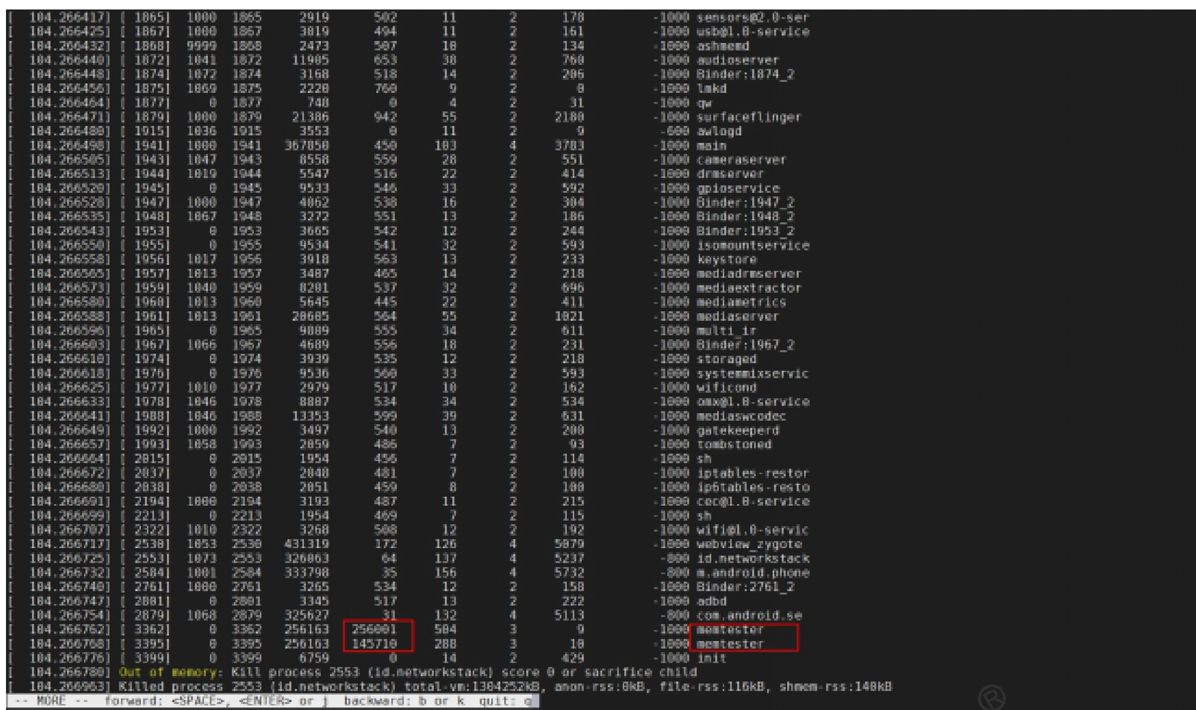


图 4-5: 进程的内存占用信息

所以这里造成 oom 的原因是 memtester 进程内存占用太高。

4.4.3 内核链表信息被破坏

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

版权所有 © 珠海全志科技股份有限公司。保留一切权利

```
[    2.546817] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[    2.549268] pgd = ffffff800a4d8000
[    2.552647] [00000000] *pgd=000000007f7fe003, *pud=000000007f7fe003, *pmd=0000000000000000
[    2.560884] Internal error: Oops: 96000005 [#1] PREEMPT SMP
[    2.566429] Modules linked in:
[    2.569465] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 4.9.170 #1
[    1.614146] Bluetooth: HCI socket layer initialized
[    1.618938] Bluetooth: L2CAP socket layer initialized
[    1.624183] Bluetooth: SCO socket layer initialized
[    1.633450] clocksource: Switched to clocksource arch_sys_counter
[    1.935199] VFS: Disk quotas dquot_6.6.0
[    1.935471] VFS: Dquot-cache hash table entries: 512 (order 0, 4096 bytes)
[    1.951774] udc_init,0
[    1.955482] thermal thermal_zone2: power_allocator: sustainable_power will be estimated
[    1.958550] NET: Registered protocol family 2
[    2.546817] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[    2.549268] pgd = ffffff800a4d8000
[    2.552647] [00000000] *pgd=000000007f7fe003, *pud=000000007f7fe003, *pmd=0000000000000000
[    2.560884] Internal error: Oops: 96000005 [#1] PREEMPT SMP
[    2.566429] Modules linked in:
[    2.569465] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 4.9.170 #1
[    2.575439] Hardware name: sun50iw10 (DT)
[    2.579427] task: ffffff8009546e00 task.stack: ffffff8009510000
[    2.585333] PC is at strcmp+0x88/0x160
[    2.589052] LR is at register_lock_class+0x40c/0x4b8
[    2.593987] pc : [<ffffff800848a9b8>] lr : [<ffffff8008115f8c>] pstate: 204001c5
[    2.601353] sp : ffffffc03daa7b90
[    2.604646] x29: ffffffc03daa7b90 x28: ffffff800999abb8
[    2.609932] x27: ffffff8009767d68 x26: ffffff8009767d68
[    2.615219] x25: 000000000002ade0 x24: ffffff800a438000
[    2.620506] x23: ffffff8009d4fbe8 x22: 0000000000000000
[    2.625792] x21: 00000000000053a0 x20: ffffff800996fdd8
[    2.631079] x19: ffffffc03daa7e08 x18: 0000000000000004
[    2.636366] x17: 00000000000001c0 x16: 000000000000000e
[    2.641652] x15: 0000000000000001 x14: 0000000000013880
```

图 4-6: 内核链表信息被破坏现场

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l 0xffffff800848a9b8
```

得到：

```
crash_arm64> dis -l 0xffffff800848a9b8
/home/luoweijian/workspace/A100/longan/kernel/linux-4.9/arch/arm64/lib/strcmp.S: 124
0xffffff800848a9b8 <strcmp+136>:        ldrb    w2, [x0],#1
```

图 4-7: crash PC 指针定位源码位置

查看源码：

```
00110: .Lmisaligned8:
00111:        /*
00112:        * Get the align offset length to compare per byte first.
00113:        * After this process, one string's address will be aligned.
00114:        */
00115:        and tmp1, src1, #7
00116:        neg tmp1, tmp1
00117:        add tmp1, tmp1, #8
00118:        and tmp2, src2, #7
00119:        neg tmp2, tmp2
00120:        add tmp2, tmp2, #8
00121:        subs tmp3, tmp1, tmp2
00122:        csel pos, tmp1, tmp2, hi /*Choose the maximum. */
00123: .Ltinycmp:
00124:        ldrb data1w, [src1], #1
00125:        ldrb data2w, [src2], #1
00126:        subs pos, pos, #1
00127:        ccmp      data1w, #1, #0, ne  /* NZCV = 0b0000.  */
00128:        ccmp      data1w, data2w, #0, cs  /* NZCV = 0b0000.  */
00129:        b.eq .Ltinycmp
00130:        cbnz pos, 1f /*find the null or unequal...*/
00131:        cmp data1w, #1
00132:        ccmp      data1w, data2w, #0, cs
```

图 4-8: 异常发生时 PC 指针源码

可以看到 strcmp 取第 0 个参数时取到了非法指针 0x1f00000000000000.

进一步查看是哪里调用了 strcmp，查看 LR 指针所处源码位置：

```
crash_arm64> dis -l 0xffffff8008115f8c
```

得到：

```
crash_arm64> dis -l 0xffffff8008115f8c
/home/luoweijian/workspace/A100/longan/kernel/linux-4.9/kernel/locking/lockdep.c: 643
0xffffff8008115f8c <register_lock_class+1036>:  ldr     w4, [x29,#144]
```

图 4-9: crash LR 指针定位源码位置

查看源码，如下：

```
00628: /*
00629:  * To make lock name printouts unique, we calculate a unique
00630:  * class->name_version generation counter:
00631:  */
00632: static int count_matching_names(struct lock_class *new_class)
00633: {
00634:        struct lock_class *class;
00635:        int count = 0;
00636:
00637:        if (!new_class->name)
00638:                return 0;
00639:
00640:        list_for_each_entry_rcu(class, &all_lock_classes, lock_entry) {
00641:                if (new_class->key - new_class->subclass == class->key)
00642:                        return class->name_version;
00643:                if (class->name && !strcmp(class->name, new_class->name))
00644:                        count = max(count, class->name_version);
00645:        }
00646:
00647:        return count + 1;
00648: }
```

图 4-10: 异常发生时 LR 指针源码

得知是内核在操作以 all_lock_classes 为头的 lock_class 结构体链表时，取到了非法指针。这里的非法指针是 0x1f00000000000000，所以逃过了内核的指针为 NULL 的合法性检查，将错误传入到了下一层的 strcmp。

Dump 出 all_lock_classes 为头的链表上所有 lock_class 结构详细信息，并转储到 all_lock_classes.txt 文件。

```
crash_arm64> list -H all_lock_classes lock_class.lock_entry -s lock_class > all_lock_classes.txt
```

查看 all_lock_classes.txt 文件，搜索 name 关键字：

```
Line 22512:  name_version = 1,
Line 22606:  name = 0xffffff8008df7173 "regmap_debugfs_early_lock",
Line 22607:  name_version = 1,
Line 22701:  name = 0xffffff8008da2ff9 "&rq->lock",
Line 22702:  name_version = 1,
Line 22796:  name = 0xffffff8008e20ec8 "__i2c_board_lock",
Line 22797:  name_version = 1,
Line 22891:  name = 0x1f00000000000000 <Address 0x1f00000000000000 out of bounds>,
Line 22892:  name_version = 0,
```

图 4-11: 查看 lock_class 中的 name

ALLWINNER®

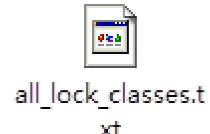


图 4-12: all_lock_classes 文件

找到 line22891 行，数据结构的 name 字符串指针非法。

再往上找到此结构的地址：

```
ffffff800998cae9
struct lock_class {
  hash_entry = {
    next = 0x0,
    pprev = 0x1000000000000000
  },
  lock_entry = {
    next = 0x3000000000000000,
    prev = 0xf8ffffff8009d56a
  },
  key = 0x18ffffff800998cc,
  subclass = 2148112585,
  dep_gen_id = 33554431,
  usage_mask = 14411518592899236O,
```

图 4-13: 找到数据结构对应 name 的地址

发现数据结构地址非法（指针末尾没有 4 对齐）

再查看链表的上一个数据成员的 next 指针：

```
ffffff800998cce8
struct lock_class {
  hash_entry = {
    next = 0x0,
    pprev = 0xffffff8009d516d0 <classhash_table+6888>
  },
  lock_entry = {
    next = 0xffffff800998caf9 <lock_classes+118049>,
    prev = 0xffffff800998cb08 <lock_classes+118064>
  },
```

图 4-14: 查看链表上一个数据成员指针

链表上一个 struct 的 next 指针已经被破坏，导致下一个数据结构地址非法，取到的 name 指针自然非法，strcmp 取到非法指针 0x1f00000000000000，内核崩溃。参见 dump 信息中 X0 寄存器（strcmp 函数的第 0 个参数）值为 0x1f00000000000000，非法 name 指针也为 0x1f00000000000000，严格匹配 strcmp dst1 地址，证实 strcmp() 函数进行字符串匹配时取到非法指针，内核崩溃。

为进一步证实推论（单个链表指针被破坏），参照正常 struct 结构规律：

```
ffffff800998caf8
struct lock_class {
  hash_entry = {
    next = 0x1,
    pprev = 0xffffff8009d56a30 <classhash_table+28232>
  },
  lock_entry = {
    next = 0xffffff800998ccf8 <lock_classes+118560>,
    prev = 0xffffff800998c918 <lock_classes+117568>
  },
```

图 4-15: 正常情况下的结构体

Next 指针值为 struct 指针值加 0x200（struct 指针值 0xffffff800998caf8+0x200=next 指针值 0xffffff800998ccf8），所以被破坏结构的 next 指针合法值应为：

0xffffff800998cce8+0x200=0xffffff800998cee8。

Struct 结构地址为：0xffffff800998cee8-0x10=0xffffff800998ced8。

获取 0xffffff800998ced8 结构体详细信息，看是否合法：

```
crash_arm64> struct lock_class 0xffffff800998ced8
```

显式结果合法，取到了合法的 prev/next 指针及合法的 name 字符串：

```
crash_arm64> struct lock_class 0xffffff800998ced8
struct lock_class {
  hash_entry = {
    next = 0x0,
    pprev = 0xffffff8009d51800 <classhash_table+7192>
  },
  lock_entry = {
    next = 0xffffff800998d0d8 <lock_classes+119552>,
    prev = 0xffffff800998ccf8 <lock_classes+118560>
  },
  key = 0xffffff80096a6418 <core_lock+112>,
  subclass = 0,
  dep_gen_id = 0,
  usage_mask = 5188,
  usage_traces = {{
      nr_entries = 0,
      max_entries = 0,
      entries = 0x0,
      skip = 0
    }, {
```

图 4-16: 获取正常偏移量时的结构体

```
  locks_after = {
    next = 0xffffff80097727b8 <list_entries+43520>,
    prev = 0xffffff80097727b8 <list_entries+43520>
  },
  locks_before = {
    next = 0xffffff800998d058 <lock_classes+119424>,
    prev = 0xffffff800998d058 <lock_classes+119424>
  },
  version = 0,
  ops = 12,
  name = 0xffffff8008e21a7c "core_lock",
  name_version = 1,
  contention_point = {0, 0, 0, 0},
  contending_point = {0, 0, 0, 0}
}
```

图 4-17: 取 prev/next 指针和 name 字符串成功

如上推测得到证实。如果 pc 指针能退回，并将此错误指针修改回去，内核就又可以欢快的 run 了。

4.4.4 内核指针被破坏

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
[ 3623.418619] Unable to handle kernel paging request at virtual address ffffffaf00d59828
[2020-02-11 11:38:28.396][ 3623.427700] pgd = ffffffc034887000
[2020-02-11 11:38:28.412][ 3623.431804] [ffffffaf00d59828] *pgd=0000000000000000|, *pud=000000000000000/
[2020-02-11 11:38:28.412][ 3623.440079] Internal error: Oops: 96000005 [#1] PREEMPT SMP
[2020-02-11 11:38:28.443][ 3623.446349] Modules linked in: xr829 gslX680new pvrsrvkm(O) xradio_btlpm vi
[2020-02-11 11:38:28.443][ 3623.466882] CPU: 0 PID: 2924 Comm: rotate-thread Tainted: G           O
[2020-02-11 11:38:28.443][ 3623.475688] Hardware name: sun50iw10 (DT)
[2020-02-11 11:38:28.459][ 3623.480194] task: ffffffc006d43500 task.stack: ffffffc035684000
[2020-02-11 11:38:28.459][ 3623.486862] PC is at put_cpu_partial+0x7c/0x1ec
[2020-02-11 11:38:28.459][ 3623.491955] LR is at put_cpu_partial+0x34/0x1ec
[2020-02-11 11:38:28.474][ 3623.497047] pc : [<ffffff8008232bcc>] lr : [<ffffff8008232b84>] pstate: 804
[2020-02-11 11:38:28.474][ 3623.505360] sp : ffffffc0356878c0
[2020-02-11 11:38:28.490][ 3623.509080] x29: ffffffc0356878c0 x28: ffffffc03ab93800
[2020-02-11 11:38:28.490][ 3623.515057] x27: ffffffc006d43500 x26: 000000000028fb6c
[2020-02-11 11:38:28.506][ 3623.521019] x25: ffffff8009193fa0 x24: 0000000000000001
[2020-02-11 11:38:28.506][ 3623.526983] x23: ffffff80095395b0 x22: ffffffc006d43500
[2020-02-11 11:38:28.506][ 3623.532953] x21: ffffffbf00eae400 x20: ffffffc03d401b00
[2020-02-11 11:38:28.506][ 3623.538915] x19: ffffffaf00d59800 x18: 0000000000000004
[2020-02-11 11:38:28.521][ 3623.544877] x17: 0000000000000000 x16: ffffff80082a44e0
[2020-02-11 11:38:28.521][ 3623.550835] x15: 0000000000000001 x14: 0000000000000000
[2020-02-11 11:38:28.537][ 3623.556784] x13: 0000000000000001 x12: 0000000000000001
[2020-02-11 11:38:28.537][ 3623.562736] x11: 0000000000000024 x10: 0000000040000000
[2020-02-11 11:38:28.552][ 3623.568695] x9 : 0000000000000000 x8 : ffffffc03cd69838
[2020-02-11 11:38:28.552][ 3623.574647] x7 : ffffff800857553c x6 : 0000000000000000
[2020-02-11 11:38:28.568][ 3623.580599] x5 : 0000000000000080 x4 : 0000000000000001
[2020-02-11 11:38:28.568][ 3623.586541] x3 : ffffffc03ab93800 x2 : 0000000000000000
[2020-02-11 11:38:28.568][ 3623.592498] x1 : 0000004034ae0000 x0 : 0000000000000001
```

图 4-18: 内核指针破坏 crash 现场

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l 0xffffff8008232bcc
```

得到：

```
crash_arm64> dis -l 0xffffff8008232bcc
/home/luoweijian/workspace/A100/longan/kernel/linux-4.9/mm/slub.c: 2225
0xffffff8008232bcc <put_cpu_partial+124>:       ldp     w1, w0, [x19,#40]
```

图 4-19: 内核指针破坏 PC 指针反汇编

查看源码：

```
02211: static void put_cpu_partial(struct kmem_cache *s, struct page *page, int drain)
02212: {
02213: #ifdef CONFIG_SLUB_CPU_PARTIAL
02214:         struct page *oldpage;
02215:         int pages;
02216:         int pobjects;
02217:
02218:         preempt_disable();
02219:         do {
02220:                 pages = 0;
02221:                 pobjects = 0;
02222:                 oldpage = this_cpu_read(s->cpu_slab->partial);
02223:
02224:                 if (oldpage) {
02225:                         pobjects = oldpage->pobjects;
02226:                         pages = oldpage->pages;
02227:                         if (drain && pobjects > s->cpu_partial) {
02228:                                 unsigned long flags;
02229:                                 /*
02230:                                  * partial array is full. Move the existing
02231:                                  * set to the per node partial list.
02232:                                  */
02233:                                 local_irq_save(flags);
02234:                                 unfreeze_partials(s, this_cpu_ptr(s->cpu_slab));
02235:                                 local_irq_restore(flags);
02236:                                 oldpage = NULL;
02237:                                 pobjects = 0;
02238:                                 pages = 0;
02239:                                 stat(s, CPU_PARTIAL_DRAIN);
02240:                         }
02241:                 }
02242:
02243:                 pages++;
02244:                 pobjects += page->objects - page->inuse;
02245:
02246:                 page->pages = pages;
02247:                 page->pobjects = pobjects;
02248:                 page->next = oldpage;
02249:
02250:         } ? end do ? while (this_cpu_cmpxchg(s->cpu_slab->partial, oldpage, page)
02251:                                                                 != oldpage);
```

图 4-20: 内核指针破坏 PC 指针对应源码

得知内核在执行寻址 oldpage 结构体成员时取到了非法指针：0xffffffaf00d59828。对应汇编如下：

```
457726 ffffff8008232b50 <put_cpu_partial>:
457727 ffffff8008232b50:   a9bb7bfd    stp x29, x30, [sp,#-80]!
457728 ffffff8008232b54:   910003fd    mov x29, sp
457729 ffffff8008232b58:   a90153f3    stp x19, x20, [sp,#16]
457730 ffffff8008232b5c:   a9025bf5    stp x21, x22, [sp,#32]
457731 ffffff8008232b60:   a90363f7    stp x23, x24, [sp,#48]
457732 ffffff8008232b64:   f90023f9    str x25, [sp,#64]
457733 ffffff8008232b68:   aa0003f4    mov x20, x0
457734 ffffff8008232b6c:   aa1e03e0    mov x0, x30
457735 ffffff8008232b70:   aa0103f5    mov x21, x1
457736 ffffff8008232b74:   2a0203f8    mov w24, w2
457737 ffffff8008232b78:   97f99a66    bl  ffffff8008099510 <_mcount>
457738 ffffff8008232b7c:   52800020    mov w0, #0x1                    // #1
457739 ffffff8008232b80:   97fab9dd    bl  ffffff80080e12f4 <preempt_count_add>
457740 ffffff8008232b84:   f0009837    adrp    x23, ffffff8009539000 <nop_trace+0x18>
457741 ffffff8008232b88:   9116c2f7    add x23, x23, #0x5b0
457742 ffffff8008232b8c:   d5384116    mrs x22, sp_el0
457743 ffffff8008232b90:   b9401ac0    ldr w0, [x22,#24]
457744 ffffff8008232b94:   11000400    add w0, w0, #0x1
457745 ffffff8008232b98:   b9001ac0    str w0, [x22,#24]
457746 ffffff8008232b9c:   f9400280    ldr x0, [x20]
457747 ffffff8008232ba0:   d538d081    mrs x1, tpidr_el1
457748 ffffff8008232ba4:   91006000    add x0, x0, #0x18
457749 ffffff8008232ba8:   f8616813    ldr x19, [x0,x1]
457750 ffffff8008232bac:   b9401ac0    ldr w0, [x22,#24]
457751 ffffff8008232bb0:   51000400    sub w0, w0, #0x1
457752 ffffff8008232bb4:   b9001ac0    str w0, [x22,#24]
457753 ffffff8008232bb8:   35000080    cbnz    w0, ffffff8008232bc8 <put_cpu_partial+0x78>
457754 ffffff8008232bbc:   f94002c0    ldr x0, [x22]
457755 ffffff8008232bc0:   36080040    tbz w0, #1, ffffff8008232bc8 <put_cpu_partial+0x78>
457756 ffffff8008232bc4:   94234b57    bl  ffffff8008b05920 <preempt_schedule_notrace>
457757 ffffff8008232bc8:   b4000353    cbz x19, ffffff8008232c30 <put_cpu_partial+0xe0>
457758 ffffff8008232bcc:   29450261    ldp w1, w0, [x19,#40]
```

图 4-21: 内核寻址到非法地址汇编代码

看最后一句指令功能。x19 寄存器中存放 page 结构体指针，偏移 40 寻找其 pobjects 结构成员。从 log 信息可以获知，x19 寄存器值：0xffffffaf00d59800 就已经不是一个合法指针了。

根据 linux 启动后的 memory mapping 图：

```
[    0.000000] CPU features: enabling workaround for ARM erratum 845719
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 258048
[    0.000000] Kernel command line: earlyprintk=sunxi-uart,0x05000000 initcall_debug=0 console=ttyS0,115200 loglevel=8 root=/dev/mmcblk0p4 init=/init partit
ns=bootloader@mmcblk0p1:env@mmcblk0p2:boot@mmcblk0p3:super@mmcblk0p4:misc@mmcblk0p5:recovery@mmcblk0p6:cache@mmcblk0p7:vbmeta@mmcblk0p8:vbmeta_system@mmcblk0
9:vbmeta_vendor@mmcblk0p10:metadata@mmcblk0p11:private@mmcblk0p12:frp@mmcblk0p13:empty@mmcblk0p14:dtbo@mmcblk0p15:media_data@mmcblk0p16:UDISK@mmcblk0p17 cma=
M snum=A100B3N109 mac_addr=10:14:15:15:9B:23 wifi_mac=10:A1:11:12:13:9D bt_mac=20:A1:11:12:13:9D specialstr= gpt=1 androidboot.vbmeta.avb_version=2.0 androi
oot.vbmeta.hash_alg=sha256 androidboot.vbmeta.size=7168 androidboot.vbmeta.digest=bc7aeaf03cab2104494808a16a47ebbfca12a28081db24facecd284e86fae14e androidboo
.vbmeta.device_state=locked androidboot.mode=normal androidboot.serialno=A100B3N109 androidboot.hardware=sun50iw10p1 boot_type=2 androidboot.boot_type=2 andr
idboot.secure_os_exist=1 gpt=1 androidboot.verifiedboots[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] Dentry cache hash table entries: 131072 (order: 8, 1048576 bytes)
[    0.000000] Inode-cache hash table entries: 65536 (order: 7, 524288 bytes)
[    0.000000] Memory: 953604K/1048576K available (10878K kernel code, 2360K rwdata, 3916K rodata, 6144K init, 13787K bss, 86780K reserved, 8192K cma-reserve
)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffffff8008000000 - 0xffffffbebfff0000   (   250 GB)
[    0.000000]       .text : 0xffffff8008080000 - 0xffffff8008b20000   ( 10880 KB)
[    0.000000]     .rodata : 0xffffff8008b20000 - 0xffffff8008f00000   (  3968 KB)
[    0.000000]       .init : 0xffffff8008f00000 - 0xffffff8009500000   (  6144 KB)
[    0.000000]       .data : 0xffffff8009500000 - 0xffffff800974e008   (  2361 KB)
[    0.000000]        .bss : 0xffffff800974e008 - 0xffffff800a4c4de8   ( 13788 KB)
[    0.000000]     fixed   : 0xffffffbefe7fb000 - 0xffffffbefec00000   (  4116 KB)
[    0.000000]     PCI I/O : 0xffffffbefee00000 - 0xffffffbeffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffffffbf00000000 - 0xffffffc000000000   (     4 GB maximum)
[    0.000000]               0xffffffbf00000000 - 0xffffffbf01000000   (    16 MB actual)
[    0.000000]     memory  : 0xffffffc000000000 - 0xffffffc040000000   (  1024 MB)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
```

图 4-22: 内核启动后 memory mapping

推测 x19 是从 0xffffffbf00d59800 翻转到 0xffffffaf00d59800 的。0xffffffbf00d59800 是一个合法的 page 指针。为了证实这个结论，进一步查看 0xffffffbf00d59800 指向的内容是否是一个合法的 page 结构，如下：

```
crash_arm64> struct page 0xffffffbf00d59800
```

得到了一个合法的 page 结构，如下，证实推测（**x19 是从 0xffffffbf00d59800 翻转到 0xffffffaf00d59800 的**）：

```
crash_arm64> struct page 0xffffffbf00d59800
struct page {
  flags = 66048,
  {
    mapping = 0x0,
    s_mem = 0x0,
    compound_mapcount = {
      counter = 0
    }
  },
  {
    index = 18446743799727534080,
    freelist = 0xffffffc035663000
  },
  {
    counters = 6443499534,
    {
      {
        _mapcount = {
          counter = -2146435058
        },
        active = 2148532238,
        {
          inuse = 14,
          objects = 16,
          frozen = 1
        },
        units = -2146435058
      },
      _refcount = {
        counter = 1
      }
    }
  },
  {
    lru = {
      next = 0xffffffbf00eabc00,
      prev = 0x300000003
    },
    pgmap = 0xffffffbf00eabc00,
    {
      next = 0xffffffbf00eabc00,
      pages = 3,
      pobjects = 3
    },
    callback_head = {
      next = 0xffffffbf00eabc00,
```

图 4-23: 地址翻转引起非法寻址

版权所有 © 珠海全志科技股份有限公司。保留一切权利

4.4.5 指针访问权限非法

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
[42559.577750] Internal error: Accessing user space memory outside uaccess.h routines: 96000005 [#1] PREEMPT SMP
[42559.588907] Modules linked in: xr829 gslX680new pvrsrvkm(O) xradio_btlpm vin_v4l2 gc0310_mipi gc2355_mipi gc030a_mipi gc2
385_mipi vin_io videobuf2_v4l2 videobuf2_dma_contig videobuf2_memops videobuf2_core [last unloaded: xr829]
[42559.611645] CPU: 2 PID: 2211 Comm: system_server Tainted: G        W  O    4.9.191 #1
[42559.620419] Hardware name: sun50iw10 (DT)
[42559.624911] task: ffffffc038661a80 task.stack: ffffffc038668000
[42559.631553] PC is at ktime_get_ts64+0x128/0x140
[42559.636627] LR is at ktime_get_ts64+0xbc/0x140
[42559.641602] pc : [<ffffff800814a1fc>] lr : [<ffffff800814a190>] pstate: 80400145
[42559.649904] sp : ffffffc03866bd40
[42559.653621] x29: ffffffc03866bd40 x28: ffffffc038661a80
[42559.659596] x27: 00000000ffd17f98 x26: ffffffc0382ff398
[42559.665559] x25: 0000000000000010 x24: 0000000000000010
[42559.671512] x23: ffffff8009560508 x22: 00000000013a5236
[42559.677463] x21: ffffff80082a5a78 x20: ffffffc03866be20
[42559.683425] x19: ffffff8009560500 x18: 0000000000000000
[42559.689381] x17: 0000000000000000 x16: ffffff80082a5e6c
[42559.695354] x15: 0000000000000000 x14: 00000000eb783741
[42559.701314] x13: 00000000ffd17f38 x12: 00000000ffd17f48
[42559.707274] x11: 000000000289a7d3 x10: 0000000000000000
[42559.713236] x9 : 0000000000000000 x8 : ffffffffa1bde2bb
[42559.719191] x7 : 000000eddea6370d x6 : 0027980b5fe117a6
[42559.725147] x5 : 0000000000000018 x4 : ffffffffc4653600
[42559.731096] x3 : 000000000000a63e x2 : 000000003b9ac9ff
[42559.737050] x1 : 000000000000a63f x0 : 000000000dc8ebef
[42559.743019]
               SP: 0xffffffc03866bcc0:
[42559.748577] bcc0  013a5236 00000000 09560508 ffffff80 00000010 00000000 00000010 00000000
```

图 4-24: 指针访问权限非法 crash 现场

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l 0xffffff800814a1fc
```

得到：

```
crash_arm64> dis -l ffffff800814a1fc
/home/huafenghuang/workspace/Q/longan/kernel/linux-4.9/kernel/time/timekeeping.c: 883
0xffffff800814a1fc <ktime_get_ts64+296>:        ldr     x23, [sp,#48]
```

图 4-25: 地址访问非法 pc 指针

死机时 CPU 正在执行指令：ldr x23, [sp, #48]，一个普通的存取堆栈操作。

对照死机 log sp 指针值：0xffffffc03866bd40 是个内核合法地址（如下图内核地址范围）：

```
[    0.000000] CPU features: enabling workaround for ARM erratum 845719
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 258048
[    0.000000] Kernel command line: earlyprintk=sunxi-uart,0x05000000 initcall_debug=0 console=ttyS0,115200 loglevel=8 root=/dev/mmcblk0p4 init=/init partit
ns=bootloader@mmcblk0p1:env@mmcblk0p2:boot@mmcblk0p3:super@mmcblk0p4:misc@mmcblk0p5:recovery@mmcblk0p6:cache@mmcblk0p7:vbmeta@mmcblk0p8:vbmeta_system@mmcblk
9:vbmeta_vendor@mmcblk0p10:metadata@mmcblk0p11:private@mmcblk0p12:frp@mmcblk0p13:empty@mmcblk0p14:dtbo@mmcblk0p15:media_data@mmcblk0p16:UDISK@mmcblk0p17 cma
M snum=A100B3N109 mac_addr=10:14:15:15:98:23 wifi_mac=10:A1:11:12:13:9D bt_mac=20:A1:11:12:13:9D specialstr= gpt=1 androidboot.vbmeta.avb_version=2.0 androi
oot.vbmeta.hash_alg=sha256 androidboot.vbmeta.size=7168 androidboot.vbmeta.digest=bc7aeaf03cab2104494808a16a47ebbfca12a28081db24facecd284e86fae14e androidboo
.vbmeta.device_state=locked androidboot.mode=normal androidboot.serialno=A100B3N109 androidboot.hardware=sun50iw10p1 boot_type=2 androidboot.boot_type=2 andr
idboot.secure_os_exist=1 gpt=1 androidboot.verifiedboots[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] Dentry cache hash table entries: 131072 (order: 8, 1048576 bytes)
[    0.000000] Inode-cache hash table entries: 65536 (order: 7, 524288 bytes)
[    0.000000] Memory: 953604K/1048576K available (10878K kernel code, 2360K rwdata, 3916K rodata, 6144K init, 13787K bss, 86780K reserved, 8192K cma-reserve
)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffffff8008000000 - 0xffffffbebfff0000   (   250 GB)
[    0.000000]       .text : 0xffffff8008080000 - 0xffffff8008b20000   ( 10880 KB)
[    0.000000]     .rodata : 0xffffff8008b20000 - 0xffffff8008f00000   (  3968 KB)
[    0.000000]       .init : 0xffffff8008f00000 - 0xffffff8009500000   (  6144 KB)
[    0.000000]       .data : 0xffffff8009500000 - 0xffffff800974e008   (  2361 KB)
[    0.000000]        .bss : 0xffffff800974e008 - 0xffffff800a4c4de8   ( 13788 KB)
[    0.000000]     fixed   : 0xffffffbefe7fb000 - 0xffffffbefec00000   (  4116 KB)
[    0.000000]     PCI I/O : 0xffffffbefee00000 - 0xffffffbeffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffffffbf00000000 - 0xffffffc000000000   (     4 GB maximum)
[    0.000000]               0xffffffbf00000000 - 0xffffffbf01000000   (    16 MB actual)
[    0.000000]     memory  : 0xffffffc000000000 - 0xffffffc040000000   (  1024 MB)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
[    0.000000] Running RCU self tests
```

图 4-26: 系统启动时内核地址范围

用 crash 工具尝试读取此地址：

```
crash_arm64> rd 0xffffffc03866bd40 0x100
```

得到：

```
crash_arm64> rd ffffffc03866bd40 0x100
ffffffc03866bd40:  ffffffc03866bd80 ffffff80082a5a78   ..f8....xZ*.....
ffffffc03866bd50:  ffffffc0382ff200 00000000314b04c0   ../8......K1....
ffffffc03866bd60:  000000000000a64f ffffffc0382fec01   O........./8....
ffffffc03866bd70:  ffffffc0382fec00 ffffff800818b42c   ../8....,.......
ffffffc03866bd80:  ffffffc03866be50 ffffff80082a5f38   P.f8....8_*.....
ffffffc03866bd90:  ffffff8009536000 0000000000000000   .`S.............
ffffffc03866bda0:  0000000000000008 000000000000002e   ................
ffffffc03866bdb0:  00000000ffd17f98 0000000000000010   ................
ffffffc03866bdc0:  000000000289a7d3 000000000000015a   ........Z.......
ffffffc03866bdd0:  ffffff8008b26000 ffffffc038661a80   .`........f8....
ffffffc03866bde0:  0000000000000000 0000004034e8f000   ...........4@...
ffffffc03866bdf0:  ffffff8009536000 00000000eb7b1160   .`S.....`.{.....
ffffffc03866be00:  00000000200d0010 0000000000000011   ... ............
ffffffc03866be10:  ffffffc03866be60 400000280000015a   `.f8....Z...(..@
ffffffc03866be20:  000000005e42c383 000000000000002e   ..B^............
```

图 4-27: 读取死机时 SP 指针

说明内存中内核页表是 ok 的。

查看 log 所处的内核源码，内核在 line：350 行报错：

```
00309: static int __kprobes do_page_fault(unsigned long addr, unsigned int esr,
00310:                                    struct pt_regs *regs)
00311: {
00312:         struct task_struct *tsk;
00313:         struct mm_struct *mm;
00314:         int fault, sig, code;
00315:         unsigned long vm_flags = VM_READ | VM_WRITE;
00316:         unsigned int mm_flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;
00317:
00318:         if (notify_page_fault(regs, esr))
00319:                 return 0;
00320:
00321:         tsk = current;
00322:         mm  = tsk->mm;
00323:
00324:         /*
00325:          * If we're in an interrupt or have no user context, we must not take
00326:          * the fault.
00327:          */
00328:         if (faulthandler_disabled() || !mm)
00329:                 goto ↓no_context;
00330:
00331:         if (user_mode(regs))
00332:                 mm_flags |= FAULT_FLAG_USER;
00333:
00334:         if (is_el0_instruction_abort(esr)) {
00335:                 vm_flags = VM_EXEC;
00336:         } else if ((esr & ESR_ELx_WNR) && !(esr & ESR_ELx_CM)) {
00337:                 vm_flags = VM_WRITE;
00338:                 mm_flags |= FAULT_FLAG_WRITE;
00339:         }
00340:
00341:         if (addr < TASK_SIZE && is_permission_fault(esr, regs)) {
00342:                 /* regs->orig_addr_limit may be 0 if we entered from EL0 */
00343:                 if (regs->orig_addr_limit == KERNEL_DS)
00344:                         die("Accessing user space memory with fs=KERNEL_DS", regs, esr);
00345:
00346:                 if (is_el1_instruction_abort(esr))
00347:                         die("Attempting to execute userspace memory", regs, esr);
00348:
00349:                 if (!search_exception_tables(regs->pc))
00350:                         die("Accessing user space memory outside uaccess.h routines", regs, esr);
00351:         }
```

图 4-28: 查看 Log 报错对应的内核源码

说明 fault 地址一定小于 4G。但此处 sp 远大于 4G，而且内核页表完好，CPU 不应该进入缺页异常分支。推测 CPU MMU 硬件出错。

4.4.6 访问内核地址报错

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
[   94.055365] Unable to handle kernel paging request at virtual address ffffffc03699f96c
[   94.055370] pgd = ffffffc012788000
[   94.055377] [ffffffc03699f96c] *pgd=0000000000000000, *pud=0000000000000000
[   94.055383] Internal error: Oops: 96000007 [#1] PREEMPT SMP
[   94.055415] Modules linked in: gslX680new xr829 pvrsrvkm(O) xradio_btlpm vin_v4l2 gc0310_mi
gc030a_mipi gc2385_mipi vin_io videobuf2_v4l2 videobuf2_dma_contig videobuf2_memops videobuf2_
[   94.055425] CPU: 0 PID: 3542 Comm: highpool[2] Tainted: G           O    4.9.170 #1
[   94.055427] Hardware name: sun50iw10 (DT)
[   94.055430] task: ffffffc0112bb500 task.stack: ffffffc013f08000
[   94.055447] PC is at sunxi_i2c_handler+0x12c/0xa6c
[   94.055451] LR is at sunxi_i2c_handler+0xfc/0xa6c
[   94.055455] pc : [<ffffff800873362c>] lr : [<ffffff80087335fc>] pstate: 804001c5
[   94.055457] sp : ffffffc03daa7ba0
[   94.055463] x29: ffffffc03daa7ba0 x28: ffffffc03699f968
[   94.055468] x27: ffffffc03daa7cb4 x26: 000000000003000d
[   94.055473] x25: ffffffc03cff0400 x24: 0000000000000000
[   94.055479] x23: ffffffc03d714700 x22: ffffff80096a7000
[   94.055484] x21: 000000000000000d x20: ffffff800807c400
[   94.055489] x19: ffffffc03d714000 x18: 000000000000000a
[   94.055494] x17: 0000000000000000 x16: ffffff80082c183c
[   94.055500] x15: 000000000000b7bc x14: ffffff808a43802f
[   94.055505] x13: ffffffffffffffff x12: 000000000000132a
[   94.055510] x11: 0000000000000000 x10: ffffff800955c388
[   94.055515] x9 : 0000000000000163 x8 : ffffff800955c380
[   94.055520] x7 : 0000000000000001 x6 : ffffff8008109814
[   94.055525] x5 : 0000000000000000 x4 : 0000000000000000
[   94.055530] x3 : 0000000000000000 x2 : 0000000000000000
[   94.055535] x1 : 0000000000010101 x0 : 0000000000000001
[   94.055538]
```

图 4-29: 访问内核地址 crash 现场

fault 地址 0xffffffc03699f96c，是一个合法的内核地址，在下图内核 mapping 空间内：

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l ffffff800873362c
```

得到：

```
crash_arm64> dis -l ffffff800873362c
/home/liyaoli/A100_Q/longan/kernel/linux-4.9/drivers/i2c/busses/i2c-sunxi.c: 992
0xffffff800873362c <sunxi_i2c_handler+300>:     ldrh    w4, [x28,#4]
```

图 4-30: 访问内核地址 crash 时 pc 指针反汇编

触发异常的指令是一条正常 load/store 指令，其中 x28 寄存器是合法内核指针，在如下地址空间内：

```
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffffff8008000000 - 0xffffffbebfff0000   (   250 GB)
[    0.000000]       .text : 0xffffff8008080000 - 0xffffff8008b20000   ( 10880 KB)
[    0.000000]     .rodata : 0xffffff8008b20000 - 0xffffff8008f00000   (  3968 KB)
[    0.000000]       .init : 0xffffff8008f00000 - 0xffffff8009510000   (  6208 KB)
[    0.000000]       .data : 0xffffff8009510000 - 0xffffff800975e008   (  2361 KB)
[    0.000000]        .bss : 0xffffff800975e008 - 0xffffff800a4d4e68   ( 13788 KB)
[    0.000000]     fixed   : 0xffffffbefe7fb000 - 0xffffffbefec00000   (  4116 KB)
[    0.000000]     PCI I/O : 0xffffffbefee00000 - 0xffffffbeffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffffffbf00000000 - 0xffffffc000000000   (     4 GB maximum)
[    0.000000]               0xffffffbf00000000 - 0xffffffbf01000000   (    16 MB actual)
[    0.000000]     memory  : 0xffffffc000000000 - 0xffffffc040000000   (  1024 MB)
```

图 4-31: 访问内核地址 crash 内核 memory layout

尝试读取引发 fault 的地址：0xffffffc03699f96c，crash 工具可以正常读取和完成地址转换：

```
crash_arm64> rd ffffffc03699f96c 10
```

如下图：

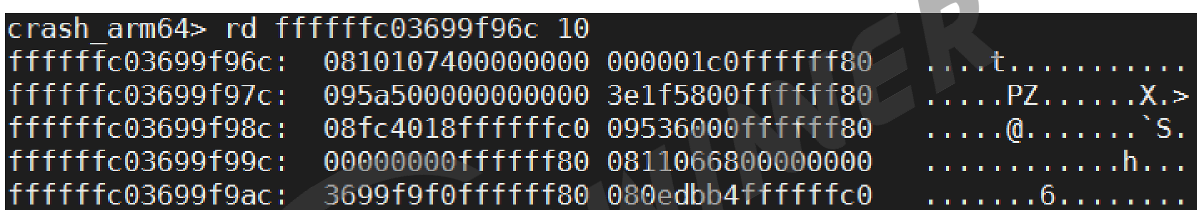


图 4-32: 读取引发 fault 的地址

手动读取内存中的 3 级内核页表。为了找到根页表，读取 init_mm 结构体，如下：

```
crash_arm64> p init_mm
```

结果如下：

```
    group_node = {
crash_arm64> task 3542 > task3542.txt
crash_arm64> p init_mm
init_mm = $1 = {
  mmap = 0x0,
  mm_rb = {
    rb_node = 0x0
  },
  vmacache_seqnum = 0,
  get_unmapped_area = 0x0,
  mmap_base = 0,
  mmap_legacy_base = 0,
  task_size = 0,
  highest_vm_end = 0,
  pgd = 0xffffff800a4d8000,
  mm_users = {
    counter = 2
  },
  mm_count = {
    counter = 1
  },
  nr_ptes = {
    counter = 0
  },
  nr_pmds = {
    counter = 1
  },
  map_count = 0,
  page_table_lock = {
    {
      rlock = {
        raw_lock = {
          owner = 90,
          next = 90
        },
        magic = 3735899821,
        owner_cpu = 4294967295,
        owner = 0xffffffffffffffff,
        dep_map = {
          key = 0xffffff80095743d0 <init_mm+128>,
          class_cache = {0xffffff80099a1648 <lock_classes+202864>, 0x0},
```

图 4-33: 打印 init_mm 结构体

得到内核页表基地址：pgd = 0xffffff800a4d8000

读取内核基页表并存到 kernel_memory_pgd_table.txt 中：

```
crash_arm64> rd 0xffffff800a4d8000 512 > kenel_memory_table.txt
```

读取的附件如下：

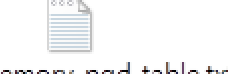


图 4-34: kenel_memory_table 文件

根据 fault 地址：0xffffffc03699f96c，找到一级页目录项，如下：

```
ffffff800a4d8800:  000000007f7ea003  0000000000000000   ..~.............
ffffff800a4d8810:  0000000000000000  0000000000000000   ................
ffffff800a4d8820:  0000000000000000  0000000000000000   ................
ffffff800a4d8830:  0000000000000000  0000000000000000   ................
```

图 4-35: 根据 fault 地址找到一级页目录

末尾 2bit 全 1，是一个合法页目录项。

读取二级页目录到 kenel_memory_pmd_c_table.txt 文件中:

```
crash_arm64> rd -p 000000007f7ea000 512 > kenel_memory_pmd_c_table.txt
```

读取附件如下：

kernel_memory_pmd_c_table.txt

图 4-36: kenel_memory_pmd_c_table 文件

根据 fault 地址：0xffffffc03699f96c，找到二级页目录项，如下：

```
7f7ea9f0:   000000007f6b1003 000000007f6b0003   ..k.......k.....
7f7eaa00:   000000007f6af003 000000007f6ae003   ..j.......j.....
7f7eaa10:   000000007f6ad003 000000007f6ac003   ..j.......j.....
```

图 4-37: 根据 fault 地址找到二级页目录

末尾 2bit 全 1，是一个合法的页目录项。

读取三级页目录到 kenel_memory_pte_139_table.txt 文件中

```
crash_arm64> rd -p 000000007f6b0000 512 > kernel_memory_pte_139_table.txt
```

截图如下，都是合法页表：

```
7f6b0000:  00e8000067e00713 00e8000067e01713   ...g.......g....
7f6b0010:  00e8000067e02713 00e8000067e03713   .'.g.....7.g....
7f6b0020:  00e8000067e04713 00e8000067e05713   .G.g.....W.g....
7f6b0030:  00e8000067e06713 00e8000067e07713   .g.g.....w.g....
7f6b0040:  00e8000067e08713 00e8000067e09713   ...g.......g....
7f6b0050:  00e8000067e0a713 00e8000067e0b713   ...g.......g....
7f6b0060:  00e8000067e0c713 00e8000067e0d713   ...g.......g....
7f6b0070:  00e8000067e0e713 00e8000067e0f713   ...g.......g....
7f6b0080:  00e8000067e10713 00e8000067e11713   ...g.......g....
7f6b0090:  00e8000067e12713 00e8000067e13713   .'.g.....7.g....
7f6b00a0:  00e8000067e14713 00e8000067e15713   .G.g.....W.g....
7f6b00b0:  00e8000067e16713 00e8000067e17713   .g.g.....w.g....
7f6b00c0:  00e8000067e18713 00e8000067e19713   ...g.......g....
7f6b00d0:  00e8000067e1a713 00e8000067e1b713   ...g.......g....
7f6b00e0:  00e8000067e1c713 00e8000067e1d713   ...g.......g....
7f6b00f0:  00e8000067e1e713 00e8000067e1f713   ...g.......g....
7f6b0100:  00e8000067e20713 00e8000067e21713   ...g.......g....
```

图 4-38: 根据 fault 地址找到三级页目录

得到附件如下：

kernel_memory_pte_139_table.txt

图 4-39: kenel_memory_pte_139_table 文件

以上过程再次说明内核页表没问题，问题处在 CPU MMU 硬件上。

4.4.7 栈指针出错

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
[2020-02-14 09:41:10.528][165958.557101] Unable to handle kernel paging request at virtual address ffffff8000f86000
[2020-02-14 09:41:10.544][165958.566083] pgd = ffffffc037136000
[2020-02-14 09:41:10.544][165958.569986] [ffffff8000f86000] *pgd=0000000000000000, *pud=0000000000000000
[2020-02-14 09:41:10.544][165958.577902] Internal error: Oops: 96000047 [#1] PREEMPT SMP
[2020-02-14 09:41:10.560][165958.584243] Modules linked in: gslX680new xr829 pvrsrvkm(O) xradio_btlpm vin_v412
gc0310_mipi gc2355_mipi gc030a_mipi gc2385_mipi vin_io videobuf2_v412 videobuf2_dma_contig videobuf2_memops
videobuf2_core
[2020-02-14 09:41:10.575][165958.604779] CPU: 1 PID: 23126 Comm: kworker/u8:2 Tainted: G           O    4.9.170 #1
[2020-02-14 09:41:10.575][165958.613643] Hardware name: sun50iw10 (DT)
[2020-02-14 09:41:10.591][165958.618825] Workqueue: pvr_misr MISRWrapper [pvrsrvkm]
[2020-02-14 09:41:10.591][165958.624685] task: ffffffc00302b500 task.stack: ffffffc004304000
[2020-02-14 09:41:10.607][165958.631424] PC is at el1_sync+0x30/0xe0
[2020-02-14 09:41:10.607][165958.635827] LR is at __switch_to+0xc0/0xdc
[2020-02-14 09:41:10.607][165958.640504] pc : [<ffffff8008083030>] lr : [<ffffff8008086998>] pstate: 604003c5
[2020-02-14 09:41:10.622][165958.648875] sp : ffffff8000f85f40
[2020-02-14 09:41:10.622][165958.652679] x29: ffffffc004307a80 x28: ffffff8008b05854
[2020-02-14 09:41:10.622][165958.658734] x27: 0000000000000000 x26: ffffff8009536dc8
[2020-02-14 09:41:10.638][165958.664785] x25: ffffff8009574350 x24: ffffffc03de47818
[2020-02-14 09:41:10.638][165958.670839] x23: ffffffc03aa65580 x22: ffffffc03aa61b00
[2020-02-14 09:41:10.653][165958.676881] x21: ffffffc002b94f80 x20: ffffffc00302b500
[2020-02-14 09:41:10.653][165958.682932] x19: ffffffc002b94f80 x18: 0000000000000004
[2020-02-14 09:41:10.653][165958.688971] x17: 0000000000000000 x16: ffffff800815a08c
[2020-02-14 09:41:10.669][165958.695025] x15: 0000000000000001 x14: ffffff8008da47f7
[2020-02-14 09:41:10.669][165958.701081] x13: 0000000000000003 x12: 0000000000000000
[2020-02-14 09:41:10.685][165958.707139] x11: ffffffc03dc7fe44 x10: 00000000000016a0
[2020-02-14 09:41:10.685][165958.713198] x9 : ffffff8004307a80 x8 : ffffffc00302cc00
[2020-02-14 09:41:10.685][165958.719246] x7 : ffffff8008105a94 x6 : 0000000000000002
[2020-02-14 09:41:10.685][165958.725294] x5 : 0000000000000001 x4 : ffffff80081a66b4
[2020-02-14 09:41:10.700][165958.731348] x3 : ffffff8008fc4018 x2 : 0000000000000002
[2020-02-14 09:41:10.700][165958.737410] x1 : ffffffc00302b500 x0 : ffffffc032608000
```

图 4-40: 栈指针出错现场

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l ffffff8008083030
```

得到：

```
crash_arm64> dis -l ffffff8008083030
/home/luoweijian/workspace/A100/longan/kernel/linux-4.9/arch/arm64/kernel/entry.S: 477
0xffffff8008083030 <el1_sync+48>:       stp     x24, x25, [sp,#192]
```

图 4-41: 栈指针出错 PC 指针反汇编

对应源码：

```
00472: /*
00473:  * EL1 mode handlers.
00474:  */
00475:         .align   6
00476: el1_sync:
00477:         kernel_entry 1
00478:         mrs x1, esr_el1                  // read the syndrome register
00479:         lsr   x24, x1, #ESR_ELx_EC_SHIFT      // exception class
00480:         cmp x24, #ESR_ELx_EC_DABT_CUR        // data abort in EL1
00481:         b.eq el1_da
00482:         cmp x24, #ESR_ELx_EC_IABT_CUR        // instruction abort in EL1
00483:         b.eq el1_ia
00484:         cmp x24, #ESR_ELx_EC_SYS64           // configurable trap
00485:         b.eq el1_undef
00486:         cmp x24, #ESR_ELx_EC_SP_ALIGN        // stack alignment exception
00487:         b.eq el1_sp_pc
00488:         cmp x24, #ESR_ELx_EC_PC_ALIGN        // pc alignment exception
00489:         b.eq el1_sp_pc
00490:         cmp x24, #ESR_ELx_EC_UNKNOWN         // unknown exception in EL1
00491:         b.eq el1_undef
00492:         cmp x24, #ESR_ELx_EC_BREAKPT_CUR         // debug exception in EL1
00493:         b.ge el1_dbg
00494:         b      el1_inv
```

图 4-42: 栈指针出错 PC 指针对应源码

477 行 kernel_entry 是一个宏，展开后如下：

```
00120:      .macro   kernel_entry, el, regsize = 64
00121:      .if    \regsize == 32
00122:      mov w0, w0                  // zero upper 32 bits of x0
00123:      .endif
00124:      stp  x0, x1, [sp, #16 * 0]
00125:      stp  x2, x3, [sp, #16 * 1]
00126:      stp  x4, x5, [sp, #16 * 2]
00127:      stp  x6, x7, [sp, #16 * 3]
00128:      stp  x8, x9, [sp, #16 * 4]
00129:      stp  x10, x11, [sp, #16 * 5]
00130:      stp  x12, x13, [sp, #16 * 6]
00131:      stp  x14, x15, [sp, #16 * 7]
00132:      stp  x16, x17, [sp, #16 * 8]
00133:      stp  x18, x19, [sp, #16 * 9]
00134:      stp  x20, x21, [sp, #16 * 10]
00135:      stp  x22, x23, [sp, #16 * 11]
00136:      stp  x24, x25, [sp, #16 * 12]
00137:      stp  x26, x27, [sp, #16 * 13]
00138:      stp  x28, x29, [sp, #16 * 14]
00139:
00140:      .if    \el == 0
00141:      mrs x21, sp_el0
00142:      ldr_this_cpu   tsk, __entry_task, x20  // Ensure MDSCR_EL1.SS is clear,
00143:      ldr   x19, [tsk, #TSK_TI_FLAGS]   // since we can unmask debug
00144:      disable_step_tsk x19, x20         // exceptions when scheduling.
00145:
00146:      apply_ssbd 1, 1f, x22, x23
```

图 4-43: kernel_entry 宏展开

根据汇编，可以得知 PC 在执行 line136 行时 panic。此时 sp 指针值：ffffff8000f85f40，加上 #192（0xc0）后调入下一个 4K 页面：0xffffff8000f86000 后，由于下一个页面还没有映射，所以 panic。

由 el1_sync 源码得知，其本身就已经是一个异常入口了，所以这个现场的 panic 信息 dump 出的是异常后再次死机的第二现场。需要继续寻找死机的第一现场：引发进入 el1_sync 的第一死机现场。

查看 LR 指针所处源码位置：

```
crash_arm64> dis -l ffffff8008086998
```

得到：

```
crash_arm64> dis -l ffffff8008086998
/home/luoweijian/workspace/A100/longan/kernel/linux-4.9/arch/arm64/kernel/process.c: 439
0xffffff8008086998 <__switch_to+192>:    ldp     x19, x20, [sp,#16]
```

图 4-44: 栈指针出错 LP 指针反汇编

对应源码：

```
00417: struct task_struct *__switch_to(struct task_struct *prev,
00418:                      struct task_struct *next)
00419: {
00420:        struct task_struct *last;
00421:
00422:        fpsimd_thread_switch(next);
00423:        tls_thread_switch(next);
00424:        hw_breakpoint_thread_switch(next);
00425:        contextidr_thread_switch(next);
00426:        entry_task_switch(next);
00427:        uao_thread_switch(next);
00428:
00429:        /*
00430:         * Complete any pending TLB or cache maintenance on this CPU in case
00431:         * the thread migrates to a different CPU.
00432:         */
00433:        dsb(ish);
00434:
00435:        /* the actual thread switch */
00436:        last = cpu_switch_to(prev, next);
00437:
00438:        return last;
00439: } ? end __switch_to ?
```

图 4-45: 栈指针出错 LP 指针对应源码

死机发生在进程切换（__switch_to()）执行完毕后的函数返回弹栈阶段。死机指令：

```
ffffff8008086994:   97fff42f    bl  ffffff8008083a50 <cpu_switch_to>
ffffff8008086998:   a94153f3    ldp x19, x20, [sp,#16]
ffffff800808699c:   a8c27bfd    ldp x29, x30, [sp],#32
ffffff80080869a0:   d65f03c0    ret
```

图 4-46: 栈指针出错死机指令

发生在 cpu_switch_to 函数执行完线程寄存器 context 切换后的第一条指令。说明 cpu_switch_to 执行完毕后 sp 的值就非法了。

根据软件逻辑，所有的合法 sp 负值操作都是在进程切换的底层操作都是在函数 cpu_switch_to 中完成的，参考下图源码 line818：

版权所有 © 珠海全志科技股份有限公司。保留一切权利

```
00799: ENTRY(cpu_switch_to)
00800:         mov x10, #THREAD_CPU_CONTEXT
00801:         add x8, x0, x10
00802:         mov x9, sp
00803:         stp  x19, x20, [x8], #16          // store callee-saved registers
00804:         stp  x21, x22, [x8], #16
00805:         stp  x23, x24, [x8], #16
00806:         stp  x25, x26, [x8], #16
00807:         stp  x27, x28, [x8], #16
00808:         stp  x29, x9, [x8], #16
00809:         str  lr, [x8]
00810:         add x8, x1, x10
00811:         ldp  x19, x20, [x8], #16          // restore callee-saved registers
00812:         ldp  x21, x22, [x8], #16
00813:         ldp  x23, x24, [x8], #16
00814:         ldp  x25, x26, [x8], #16
00815:         ldp  x27, x28, [x8], #16
00816:         ldp  x29, x9, [x8], #16
00817:         ldr  lr, [x8]
00818:         mov sp, x9
00819:         msr  sp_el0, x1
00820:         ret
00821: ENDPROC(cpu_switch_to)
```

图 4-47: cpu_switch_to 源码

系统线程的 CPU 寄存器都保存在其 task 结构体的 thread.cpu_context 成员中。查找系统所有 task 对应的 task.thread.cpu_context 成员，执行如下命令：

```
crash_arm64> foreach task -R thread.cpu_context -x > task2.txt
```

得到的系统所有 task.thread.cpu_context 值如附件：

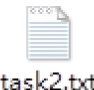


图 4-48: task2 文件

可见，所有 task 的 sp 值都是合法值。除 CPU0 的 0 号进程 sp 值为 sp = 0xffffff8xxxxxxxxx 外，其他所有的 task 内核栈指针都是：0x0xffffffcxxxxxxxxx。

截图如下：

```
Search "sp = " (758 hits in 1 file)
  T:\a100\longan\task2.txt (758 hits)
	Line 14: 	sp = 0xffffff8009513e20,
	Line 31: 	sp = 0xffffffc03d5e7e90,
	Line 48: 	sp = 0xffffffc03d5f3e90,
	Line 65: 	sp = 0xffffffc03d5f7e90,
	Line 82: 	sp = 0xffffffc03d4f7bc0,
	Line 99: 	sp = 0xffffffc03d587d80,
	Line 116: 	sp = 0xffffffc03d5a7c70,
	Line 133: 	sp = 0xffffffc03d5bfc60,
	Line 150: 	sp = 0xffffffc03d5c3b10,
	Line 167: 	sp = 0xffffffc03d5c7c10,
	Line 184: 	sp = 0xffffffc03d5d3c10,
	Line 201: 	sp = 0xffffffc03d5d7c60,
	Line 218: 	sp = 0xffffffc03d5dfc20,
	Line 235: 	sp = 0xffffffc03d5e3c60,
	Line 252: 	sp = 0xffffffc03d603c60,
	Line 269: 	sp = 0xffffffc03d607c60,
	Line 286: 	sp = 0xffffffc03d613c60,
	Line 303: 	sp = 0xffffffc03d617c60,
	Line 320: 	sp = 0xffffffc03d61bc60,
	Line 337: 	sp = 0xffffffc03d633c70,
	Line 354: 	sp = 0xffffffc03d657c60,
	Line 371: 	sp = 0xffffffc03d663c60,
	Line 388: 	sp = 0xffffffc03d667c60,
	Line 405: 	sp = 0xffffffc03d673c60,
	Line 422: 	sp = 0xffffffc03d683c70,
	Line 439: 	sp = 0xffffffc03d687c60,
	Line 456: 	sp = 0xffffffc03d69bc60,
	Line 473: 	sp = 0xffffffc03d69fc60,
	Line 490: 	sp = 0xffffffc03d6a3c60,
	Line 507: 	sp = 0xffffffc03d6b3c70,
	Line 524: 	sp = 0xffffffc03d6d3c60,
```

图 4-49: task2 中的 SP 指针

DDR 中 task 结构体都处于正常状态。此死机现场的死机线程：CPU: 1 PID: 23126 Comm: kworker。内存镜像中的 sp 指针断不可能是：ffffff8000f85f40。所以此处 sp 变为异常值最大可能是硬件状态发生了非法改变。

4.4.8 Workqueue 野指针

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
[ 1162.162657] Unable to handle kernel NULL pointer dereference at virtual address 00000008
[ 1162.171755] pgd = ffffff800a4b7000
[ 1162.175566] [00000008] *pgd=000000007f7fe003, *pud=000000007f7fe003, *pmd=0000000000000000
[ 1162.184867] Internal error: Oops: 96000005 [#1] PREEMPT SMP
[ 1162.191129] Modules linked in: xr829 gslX680new pvrsrvkm(O) xradio_btlpm vin_v4l2 gc0310_mipi
gc2355_mipi gc030a_mipi gc2385_mipi vin_io videobuf2_v4l2 videobuf2_dma_contig videobuf2_memops
videobuf2_core [last unloaded: xr829]
[ 1162.213854] CPU: 0 PID: 20811 Comm: kworker/0:3 Tainted: G           O    4.9.170 #1
[ 1162.222530] Hardware name: sun50iw10 (DT)
[ 1162.227035] task: ffffffc01cc30000 task.stack: ffffffc000fc4000
[ 1162.233695] PC is at process_one_work+0x50/0x6c4
[ 1162.238876] LR is at process_one_work+0x48/0x6c4
[ 1162.244055] pc : [<ffffff80080ce568>] lr : [<ffffff80080ce560>] pstate: 404001c5
[ 1162.252381] sp : ffffffc000fc7c80
[ 1162.256111] x29: ffffffc000fc7c80 x28: 0000000000000000
[ 1162.262092] x27: 0000000000000000 x26: ffffff80094fa000
[ 1162.268070] x25: ffffffc03dc6fd60 x24: ffffffc003da3c30
[ 1162.274045] x23: 0000000000000000 x22: ffffff8009516000
[ 1162.280024] x21: ffffffc03dc6fd00 x20: ffffff80096834c0
[ 1162.286003] x19: ffffffc003da3c00 x18: 0000000000000004
[ 1162.291981] x17: 0000000000000000 x16: ffffff800815a08c
[ 1162.297958] x15: 0000000000000001 x14: ffffff8008d93af7
[ 1162.303934] x13: 0000000000000003 x12: 0000000000000000
[ 1162.309914] x11: ffffffc03daa8e44 x10: 00000000000016a0
[ 1162.315890] x9 : 0000000000000000 x8 : ffffffc03dc6fd18
[ 1162.321859] x7 : ffffff80080cea24 x6 : 0000000000000000
[ 1162.327833] x5 : 0000000000000080 x4 : 0000000000000001
[ 1162.333812] x3 : 0000000000000000 x2 : ffffffc03dc74260
[ 1162.339791] x1 : ffffffc000fc7cf8 x0 : 0000000000000000
```

图 4-50: Workqueue 野指针 crash 现场

查看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l ffffff80080ce568
```

得到：

```
crash_arm64> dis -l ffffff80080ce568
/home/luoweijian/workspace/androidQ/longan/kernel/linux-4.9/kernel/workqueue.c: 2041
0xffffff80080ce568 <process_one_work+80>:        ldr     x0, [x0,#8]
```

图 4-51: Workqueue 野指针 PC 指针反汇编

查看 LR 指针所处源码位置：

```
crash_arm64> dis -l ffffff80080ce560
```

得到：

```
crash_arm64> dis -l ffffff80080ce560
/home/luoweijian/workspace/androidQ/longan/kernel/linux-4.9/kernel/workqueue.c: 2039
0xffffff80080ce560 <process_one_work+72>:        mov     x23, x0
```

图 4-52: Workqueue 野指针 LR 指针反汇编

查看源码如下：

```
2035  static void process_one_work(struct worker *worker, struct work_struct *work)
2036  __releases(&pool->lock)
2037  __acquires(&pool->lock)
2038  {
2039      struct pool_workqueue *pwq = get_work_pwq(work);
2040      struct worker_pool *pool = worker->pool;
2041      bool cpu_intensive = pwq->wq->flags & WQ_CPU_INTENSIVE;
2042      int work_color;
2043      struct worker *collision;
2044  #ifdef CONFIG_LOCKDEP
```

图 4-53: Workqueue 野指针 LR 指针源码

对照其反汇编：

```
ffffff80080ce518 <process_one_work>:
ffffff80080ce518:    a9b57bfd    stp x29, x30, [sp,#-176]!
ffffff80080ce51c:    910003fd    mov x29, sp
ffffff80080ce520:    a90153f3    stp x19, x20, [sp,#16]
ffffff80080ce524:    a9025bf5    stp x21, x22, [sp,#32]
ffffff80080ce528:    a90363f7    stp x23, x24, [sp,#48]
ffffff80080ce52c:    a9046bf9    stp x25, x26, [sp,#64]
ffffff80080ce530:    a90573fb    stp x27, x28, [sp,#80]
ffffff80080ce534:    aa0003f3    mov x19, x0
ffffff80080ce538:    aa1e03e0    mov x0, x30
ffffff80080ce53c:    aa0103f4    mov x20, x1
ffffff80080ce540:    9000a256    adrp    x22, ffffff8009516000 <nf_conntrac
ffffff80080ce544:    97ff2bf3    bl  ffffff8008099510 <_mcount>
ffffff80080ce548:    913642c0    add x0, x22, #0xd90
ffffff80080ce54c:    f9400001    ldr x1, [x0]
ffffff80080ce550:    f90057a1    str x1, [x29,#168]
ffffff80080ce554:    d2800001    mov x1, #0x0                    // #0
ffffff80080ce558:    aa1403e0    mov x0, x20
ffffff80080ce55c:    97fff196    bl  ffffff80080cabb4 <get_work_pwq>
ffffff80080ce560:    aa0003f7    mov x23, x0
ffffff80080ce564:    9101e3a1    add x1, x29, #0x78
ffffff80080ce568:    f9400400    ldr x0, [x0,#8]
```

图 4-54: process_one_work 反汇编

死机 PC 指针：ffffff80080ce568 处的死机原因是 x0 寄存器为 0，load/store 空指针出错。从反汇编可以看到，x0 寄存器其实是 get_work_pwq() 函数的返回值，说明 get_work_pwq() 函数的返回值为 0.

查看 get_work_pwq() 函数返回 NULL 的原因，如下图：

```
00682: static struct pool_workqueue *get_work_pwq(struct work_struct *work)
00683: {
00684:         unsigned long data = atomic_long_read(&work->data);
00685:
00686:         if (data & WORK_STRUCT_PWQ)
00687:                 return (void *)(data & WORK_STRUCT_WQ_DATA_MASK);
00688:         else
00689:                 return NULL;
00690: }
```

图 4-55: get_work_pwq 函数

可以看出，只有参数 work->data 成员标志位 WORK_STRUCT_PWQ 没有置位时会返回 NULL。

为了确认分析过程正确性，尝试反推 get_work_pwq() 函数的参数。再次查看 process_one_work() 函数源码，根据 ARM 函数调用规则，函数第 0 个参数会放在 x0 寄存器，第一个参数会放在 x1 寄存器。所以这里需要回溯 process_one_work() 函数入口的 x0/x1 寄存器的值。

查看 process_one_work() 函数反汇编，可以看到：

```
ffffff80080ce518 <process_one_work>:
ffffff80080ce518:    a9b57bfd     stp x29, x30, [sp,#-176]!
ffffff80080ce51c:    910003fd     mov x29, sp
ffffff80080ce520:    a90153f3     stp x19, x20, [sp,#16]
ffffff80080ce524:    a9025bf5     stp x21, x22, [sp,#32]
ffffff80080ce528:    a90363f7     stp x23, x24, [sp,#48]
ffffff80080ce52c:    a9046bf9     stp x25, x26, [sp,#64]
ffffff80080ce530:    a90573fb     stp x27, x28, [sp,#80]
ffffff80080ce534:    aa0003f3     mov x19, x0
ffffff80080ce538:    aa1e03e0     mov x0, x30
ffffff80080ce53c:    aa0103f4     mov x20, x1
```

图 4-56: process_one_work 反汇编查看 x0/x1

在 process_one_work() 函数入口，会将 x0 寄存器暂存到 x19 寄存器，将 x1 寄存器暂存到 x20 寄存器。所以，x20 寄存器中的值既是参数 work 的值。根据 panic 信息中寄存器值的打印，可以看到 x20 寄存器的值为：ffffff80096834c0。

查看参数 work 指针指向的 work 结构体内容：

```
crash_arm64> struct -x work_struct ffffff80096834c0
```

得到：

```
crash_arm64> struct -x work_struct ffffff80096834c0
struct work_struct {
  data = {
    counter = 0xffffffe0
  },
  entry = {
    next = 0xffffff80096834c8 <sunxi_udc+3416>,
    prev = 0xffffff80096834c8 <sunxi_udc+3416>
  },
  func = 0xffffff8008703af8 <sunxi_vbus_det_work>,
  lockdep_map = {
    key = 0xffffff800a4a1e50 <__key.37047>,
    class_cache = {0x0, 0x0},
    name = 0xffffff8008e0b3a3 "(&udc->vbus_det_work)",
    cpu = 0x1,
    ip = 0x0
  }
}
```

图 4-57: 查看 struct work_struct 结构体的值

可以看到，work->data 成员的标志位 WORK_STRUCT_PWQ 没有置位标志位没有置位，所以 get_work_pwq() 函数返回 NULL 是正确的。

进一步分析此 work 的具体内容，可以确认，data 成员的值：0xffffffe0 是 work 初始化 INIT_WORK 时所附的值。

INIT_WORK 的定义如下：

```
#ifdef CONFIG_LOCKDEP
#define __INIT_WORK(_work, _func, _onstack)                    \
	do {                                                          \
		static struct lock_class_key __key;                   \
                                                                \
		__init_work((_work), _onstack);                       \
		(_work)->data = (atomic_long_t) WORK_DATA_INIT();     \
		lockdep_init_map(&(_work)->lockdep_map, #_work, &__key, 0); \
		INIT_LIST_HEAD(&(_work)->entry);                      \
		(_work)->func = (_func);                              \
	} while (0)
#else
#define __INIT_WORK(_work, _func, _onstack)                    \
	do {                                                          \
		__init_work((_work), _onstack);                       \
		(_work)->data = (atomic_long_t) WORK_DATA_INIT();     \
		INIT_LIST_HEAD(&(_work)->entry);                      \
		(_work)->func = (_func);                              \
	} while (0)
#endif
```

图 4-58: INIT_WORK 定义

其中 WORK_DATA_INIT 的定义如下：

```
#define WORK_DATA_INIT()     ATOMIC_LONG_INIT(WORK_STRUCT_NO_POOL)
```

图 4-59: WORK_DATA_INIT 定义

进一步，WORK_STRUCT_NO_POOL 的定义如下：

```
WORK_STRUCT_NO_POOL      = (unsigned long)WORK_OFFQ_POOL_NONE << WORK_OFFQ_POOL_SHIFT,
```

图 4-60: WORK_STRUCT_NO_POOL 定义

正好是：0xffffffe0。

有 INIT_WORK 宏定义参考 work 内容可以断定，work 的 entry 成员和 func 成员也是由 INIT_WORK 初始化而来。

有此 work 的 func 成员可知，此 work 的执行函数是：sunxi_vbus_det_work。查看 usb 驱动，此 work 是有 schedule_work（）函数在 usb irq 中异步调度的，见下：

```
/* SUSPEND */
if (usb_irq & USBC_INTUSB_SUSPEND) {
	DMSG_INFO_UDC("IRQ: suspend\n");

	/* clear interrupt */
	USBC_INT_ClearMiscPending(g_sunxi_udc_io.usb_bsp_hdle,
				USBC_INTUSB_SUSPEND);

	if (dev->gadget.speed != USB_SPEED_UNKNOWN) {
		schedule_work(&dev->vbus_det_work);
		usb_connect = 0;
		if (!Is_Charger_Mode) {
			wake_unlock(&udc_wake_lock);
			pr_debug("usb_connecting: release wake lock\n");
		}
```

图 4-61: 异步调用 sunxi_vbus_det_work

分析 schedule_work() 函数的内核实现：schedule_work()->queue_work()->queue_work_on()->queue_delayed_work_on()->__queue_delayed_work()->__queue_work()->insert_work()，

```
static void insert_work(struct pool_workqueue *pwq, struct work_struct *work,
		struct list_head *head, unsigned int extra_flags)
{
	struct worker_pool *pool = pwq->pool;

	/* we own @work, set data and link */
	set_work_pwq(work, pwq, extra_flags);
	list_add_tail(&work->entry, head);
	get_pwq(pwq);

	/*
	 * Ensure either wq_worker_sleeping() sees the above
	 * list_add_tail() or we see zero nr_running to avoid workers lying
	 * around lazily while there are works to be processed.
	 */
	smp_mb();

	if (__need_more_worker(pool))
		wake_up_worker(pool);
} ? end insert_work ?
```

图 4-62: insert_work 函数

会将 work->data 赋值为 pwq 指针。所以正常内核执行路径下，process_one_work() 函数中调用 get_work_pwq() 函数，其是不会返回 NULL 的。所以内核在 process_one_work() 函数中也没有做 get_work_pwq() 函数返回值的合法性判断（因为这里不会出错）。

所以，此现场一定是其他并发线程破坏了 work 结构体的内容。根据死机时 work 结构体的内容，基本可以断定此 work 是又被并发线程 INIT_WORK 了，从而导致内核崩溃。

4.4.9 分支跳转错误

使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以看到最后的死机现场如下：

```
Unable to handle kernel NULL pointer dereference at virtual address 000004d
pgd = ffffff800a4c7000
[000004d8] *pgd=000000007f7fe003, *pud=000000007f7fe003,

Internal error: Oops: 96000005 [#1] PREEMPT SMP
Modules linked in: xr829 gslX680new pvrsrvkm(O) xradio_btlpm vin_v412
35_mipi vin_io videobuf2_v4l2 videobuf2_dma_contig videobuf2_memops

CPU: 0 PID: 973 Comm: mmcqd/0 Tainted: G           O    4.9.170 #1
Hardware name: sun50iw10 (DT)
task: ffffffc03af98000 task.stack: ffffffc03aed4000
PC is at test_clear_page_writeback+0x208/0x28c
LR is at test_clear_page_writeback+0x1fc/0x28c
pc : [<ffffff80081ed474>] lr : [<ffffff80081ed468>] pstate: 80400145
sp : ffffffc03aed7a30
x29: ffffffc03aed7a30 x28: 0000000000000000
x27: 0000000000000000 x26: ffffffc00dffdb40
x25: 0000000000000140 x24: ffffffc00dffdb50
x23: 0000000000000001 x22: ffffff8009527000
x21: ffffffc03af80260 x20: ffffffc00dffdb38
x19: ffffffbf005b9ac0 x18: 0000000000000001
x17: 0000000000000000 x16: ffffff800815a08c
x15: 0000000000000001 x14: ffffff8008da3af7
x13: 0000000000000001 x12: ffffff8009526000
x11: 0000000000000001 x10: 0000000000000040
x9 : 0000000000000000 x8 : ffffffc03abc4cf0
x7 : ffffff8008288eac x6 : 0000000000000000
x5 : 0000000000000080 x4 : 0000000000000001
x3 : 0000000000000000 x2 : ffffffffffffffff
x1 : 0000004034ae1000 x0 : 0000000000000000
```

图 4-63: 分支跳转错误 crash 现场

看死机地址 PC 指针所处源码位置：

```
crash_arm64> dis -l ffffff80081ed474
```

得到：

```
crash_arm64> dis -l ffffff80081ed474
/home/zengshuchuan/workspace/sunxi-platform-q-sync/longan/kernel/linux-4.9/./include/linux/memcontrol.h: 517
0xffffff80081ed474 <test_clear_page_writeback+520>:     ldr     x0, [x0,#1240]
```

图 4-64: 分支跳转错误 PC 指针反汇编

查看 LR 指针所处源码位置：

```
crash_arm64> dis -l ffffff80081ed468
```

得到：

```
crash_arm64> dis -l ffffff80081ed468
/home/zengshuchuan/workspace/sunxi-platform-q-sync/longan/kernel/linux-4.9/./include/linux/memcontrol.h: 517
0xffffff80081ed468 <test_clear_page_writeback+508>:     ldr     x0, [x19,#56]
```

图 4-65: 分支跳转错误 LR 指针反汇编

查看源码如下：

```
00511: static inline void mem_cgroup_update_page_stat(struct page *page,
00512:                  enum mem_cgroup_stat_index idx, int val)
00513: {
00514:         VM_BUG_ON(!(rcu_read_lock_held() || PageLocked(page)));
00515:
00516:         if (page->mem_cgroup)
00517:                 this_cpu_add(page->mem_cgroup->stat->count[idx], val);
00518: }
```

图 4-66: 分支跳转错误 LR 指针对应源码

对照其汇编实现：

```
ffffff80081ed44c:    f9400000    ldr x0, [x0]
ffffff80081ed450:    37000040    tbnz    w0, #0, ffffff80081ed458 <test_clear_page_writeback+0x1ec>
ffffff80081ed454:    d4210000    brk #0x800
ffffff80081ed458:    f9401e60    ldr x0, [x19,#56]
ffffff80081ed45c:    b40002a0    cbz x0, ffffff80081ed4b0 <test_clear_page_writeback+0x244>
ffffff80081ed460:    52800020    mov w0, #0x1                        // #1
ffffff80081ed464:    97fbcfa4    bl  ffffff80080e12f4 <preempt_count_add>
ffffff80081ed468:    f9401e60    ldr x0, [x19,#56]
ffffff80081ed46c:    92800002    mov x2, #0xffffffffffffffff         // #-1
ffffff80081ed470:    d538d081    mrs x1, tpidr_el1
ffffff80081ed474:    f9426c00    ldr x0, [x0,#1240]
ffffff80081ed478:    9100a000    add x0, x0, #0x28
ffffff80081ed47c:    8b010000    add x0, x0, x1
ffffff80081ed480:    c85f7c04    ldxr    x4, [x0]
ffffff80081ed484:    8b020084    add x4, x4, x2
ffffff80081ed488:    c8037c04    stxr    w3, x4, [x0]
```

图 4-67: 对照 LR 指针函数反汇编

可以看出，源码 line516 行判断 page->mem_cgroup 为空时，需要跳转到 ffffff80081ed4b0，但 CPU 执行此 cbz 指令（ffffff80081ed45c）时没有跳转，但实际去 load page->mem_cgroup 地址时，发现其为空。Cpu 指令执行结果与内存中内容不符。

进一步查看 page 结构体的内存数据（从反汇编可以看出，x19 寄存器中存放的是 page 结构体指针）：

```
crash_arm64> struct page ffffffbf005b9ac0
```

得到：

```
crash_arm64> struct -x page ffffffbf005b9ac0
struct page {
  flags = 0x0,
  {
    mapping = 0x0,
    s_mem = 0x0,
    compound_mapcount = {
      counter = 0x0
    }
  },
  {
    index = 0x0,
    freelist = 0x0
  },
  {
    counters = 0xffffffff,
    {
      {
        _mapcount = {
          counter = 0xffffffff
        },
        active = 0xffffffff,
        {
          inuse = 0xffff,
          objects = 0x7fff,
          frozen = 0x1
        },
        units = 0xffffffff
      },
      _refcount = {
        counter = 0x0
      }
    }
  },
  {
    lru = {
      next = 0xffffffbf00556c20,
      prev = 0xffffffbf00093520
    },
    pgmap = 0xffffffbf00556c20,
    {
      next = 0xffffffbf00556c20,
```

图 4-68: 查看 page 结构体的值 1

版权所有 © 珠海全志科技股份有限公司。保留一切权利

```
        pages = 0x93520,
        pobjects = 0xffffffbf
      },
      callback_head = {
        next = 0xffffffbf00556c20,
        func = 0xffffffbf00093520
      },
      {
        compound_head = 0xffffffbf00556c20,
        compound_dtor = 0x93520,
        compound_order = 0xffffffbf
      }
    },
    {
      private = 0x0,
      ptl = 0x0,
      slab_cache = 0x0
    },
    mem_cgroup = 0x0
}
```

图 4-69: 查看 page 结构体的值 2

再次确认 page->mem_cgroup 为空。Dram 中数据正确，CPU 执行 load/cbz 指令流出问题。

Cbz 执行时是根据 cpsr 寄存器 z 位是否为空来决定分支跳转的，如下图：

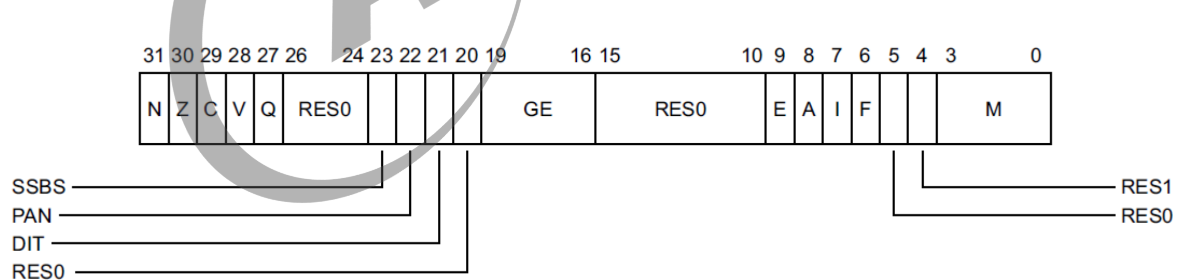


图 4-70: CPSR 寄存器分布

推测，CPU 在执行 cbz 指令时，cpsr z 位发生了 bit 翻转。

4.4.10 CPU 调频过程中死机

在现场确认了 cpu 供电和 PLL_CPU 的寄存器值之后，使用 dmesg 命令查看死机现场的 log。

```
crash_arm64> dmesg
```

可以根据 log 去分析死机的现场，但调频调压过程中，由于电压的变化是有过程的，有可能死机的原因发生在调频调压的过程中，故挂死的 log 往往不能够真实的查清挂死的原因，所以需要分析当前进程的状态以及分析 CPU 调频和调压的变化过程，结合起来确认系统是否是因为调频而导致的系统挂死。

接下来需要知道当前进程状态：

```
crash> ps
```

得到挂死现场的进程如下：

```
    43     2  2 c1a61f40  ID  0.0      0      0  [kworker/2:1]
...skipping...
>  1126     2  2 c232f6c0  RU  0.0      0      0  [sugov:0]
   1134     2  0 c232d780  IN  0.0      0      0  [irq/43-mmc2]
   1142     2  0 c232f080  IN  0.0      0      0  [irq/44-mmc0]
```

找到 cpufreq 的进程（不同的调频策略，对应的调频进程是不一样的，需要结合实际代码来分析，这里选择 schedutil 来分析），对应的进程为 sugov。

找到进程以后可以通过查看当前各 CPU 的 backtrace 来查看死机现场处于调频调压的哪个阶段。

首先查看当前每个 CPU 当前正在运行的进程（由于内存镜像是通过 csat dump 出来的数据，无法解析出 cpu 当前执行的进程信息）：

```
crash> bt -a
```

```
crash> bt -a
PID: 0         TASK: c0e07040  CPU: 0    COMMAND: "swapper/0"
bt: WARNING: cannot determine starting stack frame for task c0e07040

PID: 0         TASK: c1874b00  CPU: 1    COMMAND: "swapper/1"
bt: WARNING: cannot determine starting stack frame for task c1874b00

PID: 1126      TASK: c232f6c0  CPU: 2    COMMAND: "sugov:0"
bt: WARNING: cannot determine starting stack frame for task c232f6c0

PID: 0         TASK: c1875780  CPU: 3    COMMAND: "swapper/3"
bt: WARNING: cannot determine starting stack frame for task c1875780
crash>
```

图 4-71: 当前 CPU 正在运行的进程

然后根据 cpufreq 的进程 PID 查看 backtrace：

crash> bt -T 1126

```
PID: 1126   TASK: c232f6c0  CPU: 2   COMMAND: "sugov:0"
bt: WARNING: cannot determine starting stack frame for task c232f6c0
 [c2357934] update_sd_lb_stats at c0144914
 [c23579ac] find_busiest_group at c0144bd0
 [c2357a4c] load_balance at c0144ff8
 [c2357a94] __accumulate_pelt_segments at c014ef4c
 [c2357ad0] schedule at c089338c
 [c2357adc] update_cfs_rq_load_avg at c01417b4
 [c2357af4] _raw_spin_unlock_irq at c0896b50
 [c2357afc] finish_task_switch at c013bac4
 [c2357b0c] notifier_call_chain at c0137a28
 [c2357b30] schedule at c089338c
 [c2357b44] _raw_spin_lock_irqsave at c08968a4
 [c2357b54] lock_timer_base at c01745e4
 [c2357b6c] _raw_spin_unlock_irqrestore at c0896e28
 [c2357b74] try_to_del_timer_sync at c01746d8
 [c2357b8c] del_timer_sync at c0174730
 [c2357b94] schedule_timeout at c0895f88
 [c2357bb4] __pm_runtime_suspend at c0491ce4
 [c2357bcc] sunxi_i2c_xfer at c0604110
 [c2357be0] autoremove_wake_function at c014bc7c
 [c2357c14] __i2c_transfer at c05305b4
 [c2357c2c] mutex_unlock at c08941d8
 [c2357c3c] i2c_transfer at c05306f4
 [c2357c54] i2c_transfer_buffer_flags at c0530754
 [c2357c74] regmap_i2c_write at c049e220
 [c2357c7c] _regmap_raw_write_impl at c049a40c
 [c2357cb0] regmap_format_8 at c0497f08
 [c2357cc4] _regmap_bus_raw_write at c049a58c
 [c2357ce4] _regmap_update_bits at c0499e44
 [c2357cfc] mutex_unlock at c08941d8
 [c2357d0c] regmap_update_bits_base at c049acf8
 [c2357d2c] regmap_update_bits at c045e144
 [c2357d44] regulator_set_voltage_sel_regmap at c045e374
 [c2357d4c] _regulator_call_set_voltage_sel at c045a1a8
 [c2357d54] read_current_timer at c0420718
 [c2357d5c] __timer_delay at c0420760
 [c2357d74] _regulator_do_set_voltage at c045a700
 [c2357dbc] regulator_set_voltage_rdev at c045d99c
 [c2357de4] regulator_do_balance_voltage at c045bab8
 [c2357e1c] mutex_unlock at c08941d8
 [c2357e24] mutex_unlock at c08941d8
 [c2357e34] mutex_unlock at c08941d8
 [c2357e3c] mutex_unlock at c08941d8
 [c2357e4c] regulator_unlock_recursive at c045832c
 [c2357e6c] regulator_set_voltage at c045dab0
 [c2357ebc] newidle_balance at c0145cdc
 [c2357ee4] notifier_call_chain at c0137a28
 [c2357f14] __cpufreq_driver_target at c057e03c
 [c2357f54] sugov_work at c014fa9c
 [c2357f6c] kthread_worker_fn at c0136c08
 [c2357f7c] kthread_worker_fn at c0136b34
 [c2357f8c] kthread at c0136740
 [c2357f94] kthread at c013662c
 [c2357fac] ret_from_fork at c0100148
```

ALLWINNER®

可以根据函数调用栈来确定此时处于调频调压中的什么阶段，例如上述 backtrace 从 cpufreq 的函数调用栈来看:

```
cpufreq_driver_target->regulator_set_voltage->regulator_do_balance_voltage->_regulator_do_set_voltage->regulator_set_voltage_sel_regmap->regmap_update_bits->i2c_transfer->sunxi_i2c_xfer->schedule_timeout
```

首先是发起一次调频调压的过程，然后去调电压并且最终调用 i2c 的接口，最后由于等待 i2c 传输时间稍长，进程被调度出去。

确认 cpufreq 的调用栈后，还可以通过结构体的数据判断此时想要从 cur_freq 调到 next_freq，首先通过代码找到 policy 对应的相关静态变量 cpufreq_policy_list，并且通过该变量找到对应的 policy。

```
crash> p cpufreq_policy_list cpufreq_policy_list = $1 = { next = 0xc2319ae8, prev = 0xc2319ae8 }
```

找到 cpufreq_policy_list 以后，根据它在 cpufreq_policy 结构体中的位置，反推出对应的 policy 地址。

```
crash> struct cpufreq_policy -o
```

```
crash> struct cpufreq_policy -o
struct cpufreq_policy {
     [0] cpumask_var_t cpus;
     [4] cpumask_var_t related_cpus;
     [8] cpumask_var_t real_cpus;
    [12] unsigned int shared_type;
    [16] unsigned int cpu;
    [20] struct clk *clk;
    [24] struct cpufreq_cpuinfo cpuinfo;
    [36] unsigned int min;
    [40] unsigned int max;
    [44] unsigned int cur;
    [48] unsigned int restore_freq;
    [52] unsigned int suspend_freq;
    [56] unsigned int policy;
    [60] unsigned int last_policy;
    [64] struct cpufreq_governor *governor;
    [68] void *governor_data;
    [72] char last_governor[16];
    [88] struct work_struct update;
   [104] struct freq_constraints constraints;
   [216] struct freq_qos_request *min_freq_req;
   [220] struct freq_qos_request *max_freq_req;
   [224] struct cpufreq_frequency_table *freq_table;
   [228] enum cpufreq_table_sorting freq_table_sorted;
   [232] struct list_head policy_list;
   [240] struct kobject kobj;
   [276] struct completion kobj_unregister;
   [292] struct rw_semaphore rwsem;
   [316] bool fast_switch_possible;
   [317] bool fast_switch_enabled;
   [318] bool strict_target;
   [320] unsigned int transition_delay_us;
   [324] bool dvfs_possible_from_any_cpu;
   [328] unsigned int cached_target_freq;
```

图 4-72: cpufreq_policy 结构体成员的偏移

list_head 在 cpufreq_policy 结构体里面偏移为 232，所以可以反推出 policy 的地址为 0xc2319ae8 - 0xe8 = 0xc2319a00，此时查看 policy 对应的结构体 cpufreq_policy。

```
crash> struct cpufreq_policy 0xc2319a00
```

```
struct cpufreq_policy {
  cpus = {{
      bits = {15}
    }},
  related_cpus = {{
      bits = {15}
    }},
  real_cpus = {{
      bits = {15}
    }},
  shared_type = 0,
  cpu = 0,
  clk = 0xc21319c0,
  cpuinfo = {
    max_freq = 1200000,
    min_freq = 480000,
    transition_latency = 3000000
  },
  min = 480000,
  max = 1200000,
  cur = 1008000,
  restore_freq = 1008000,
  suspend_freq = 0,
  policy = 0,
  last_policy = 0,
  governor = 0xc0e0bcf0 <schedutil_gov>,
  governor_data = 0xc214d900,
  last_governor = "\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000",
  update = {
    data = {
      counter = 0
    },
    entry = {
      next = 0xc2319a5c,
      prev = 0xc2319a5c
    },
```

图 4-73: cpufreq_policy 结构体数据

可以看到此时当前的频率为 1008000。

并且从 schedutil 的源码知道，想要调整的频率是在 sg_policy 这个结构体中的，而 sg_policy 是在初始化中被赋值到了 governor_data 中，所以可以查到 sg_policy 对应的结构体的数据。

```
crash> struct sugov_policy 0xc214d900
```

版权所有 © 珠海全志科技股份有限公司。保留一切权利

```
crash> struct sugov_policy 0xc214d900
struct sugov_policy {
  policy = 0xc2319a00,
  tunables = 0xc1af0700,
  tunables_hook = {
    next = 0xc1af0724,
    prev = 0xc1af0724
  },
  update_lock = {
    raw_lock = {
      {
        slock = 357373261,
        tickets = {
          owner = 5453,
          next = 5453
        }
      }
    }
  },
  last_freq_update_time = 6409162849,
  freq_update_delay_ns = 10000000,
  next_freq = 1200000,
  cached_raw_freq = 1500000,
  irq_work = {
    {
      node = {
        llist = {
          next = 0x0
        },
        {
          u_flags = 32,
          a_flags = {
            counter = 32
          }
        }
```

图 4-74: sugov_policy 结构体数据

所以可以知道卡死的现场是准备将 CPU 频率从 1008000 调到 1200000，并且是在调压后卡死的。

5 注意事项 FAQ

5.1 WARNING: cannot access vmalloc’d module memory

crash 加载过程中出现如下的 log：

```
machine type mismatch / “WARNING: cannot access vmalloc'd module memory ”
```

表示 DRAM 数据与 vmlinux 不匹配，需要重新提供匹配的数据。

5.2 crash: cannot determine page size

crash 加载过程中出现如下的 log：

```
crash: cannot determine page size
```

- 表示 crash 工具不匹配，区分是 arm32 还是 arm64 的平台；
- 要注意 dump 内存的大小，要完全 dump 出来。详见 3.1.1 章节检查 crash 数据的获取时配置是否正确；
- 可以在解析命令后添加参数-p 4k指定 page size 的大小；
- 也可能是 crash 工具的版本不匹配，推荐使用最新的 crash 进行调试验证。

5.3 crash_arm64: read error: kernel virtual address: ffffffc0087cf580 type: “kernel_config_data”

crash 加载过程中出现如下的 log：

```
crash_arm64: read error: kernel virtual address: ffffffc0087cf580  type: "kernel_config_data"
WARNING: cannot read kernel_config_data
crash_arm64: read error: kernel virtual address: ffffffc008e68860  type: "possible"
WARNING: cannot read cpu_possible_map
crash_arm64: read error: kernel virtual address: ffffffc008e68858  type: "present"
WARNING: cannot read cpu_present_map
crash_arm64: read error: kernel virtual address: ffffffc008e68850  type: "online"
```

```
WARNING: cannot read cpu_online_map
crash_arm64: read error: kernel virtual address: ffffffc008e68878  type: "active"
WARNING: cannot read cpu_active_map
crash_arm64: read error: kernel virtual address: ffffffc008f875a8  type: "shadow_timekeeper xtime_sec"
crash_arm64: read error: kernel virtual address: ffffffc008e6dc44  type: "init_uts_ns"
crash_arm64: tmp/vmlinux and /var/tmp/ramdump_elf_sDmBG5 do not match!
```

- 表示 crash 工具指定的 kimage_voffset 有误，需要提供正确的内核镜像偏移量参数。
- 对于 ARM 64 位系统，mmu 模块会将 kimage_voffset 导出为全局符号，见kernel/linux-5.15/arch/arm64/mm/mmu.c。
- 在任意能够运行的驱动添加如下代码将信息偏移量信息打印出来替换掉该参数即可。

```
diff --git a/drivers/usb/sunxi_usb/manager/usb_msg_center.c b/drivers/usb/sunxi_usb/manager/usb_msg_center.c
index 02cd569..1bfd2d8 100644
--- a/drivers/usb/sunxi_usb/manager/usb_msg_center.c
+++ b/drivers/usb/sunxi_usb/manager/usb_msg_center.c
@@ -172,6 +172,7 @@ static void insmod_device_driver(struct usb_msg_center_info *center_info)

 #endif
     DMSG_INFO("\ninsmod_device_driver\n\n");
+     printk("kimage_voffset: 0x%llx\n", kimage_voffset);

     set_usb_role(center_info, USB_ROLE_DEVICE);
```

5.4 WARNING: could not find MAGIC_START!

crash 加载过程中出现如下的 log：

```
WARNING: could not find MAGIC_START!
crash_arm64s: tmp/vmlinux and /var/tmp/ramdump_elf_TvkfBx do not match!
```

- 尝试升级到最新的 crash 工具进行调试；
- 检查死机时 Log 是否打印 “Kernel Offset:” 相关信息。
- 内核开启 **CONFIG_RANDOMIZE_BASE** 配置并且配置的 kaslr_offset 不为 0，则 kernel image 映射的地址相对于链接地址有个偏移。偏移地址可以通过 dts 设置。

具体介绍参考：http://www.wowotech.net/memory_management/441.html

- 在解析命令后添加参数--kaslr 0x80000指定 kaslr_offset 的大小。

在 5.4 上进行测试时：

```
/ # echo "c" > /proc/sysrq-trigger
[  221.612183] sysrq: Trigger a crash
```

```
[ 221.616041] Kernel panic - not syncing: sysrq triggered crash
[ 221.622478] CPU: 0 PID: 411 Comm: ash Not tainted 5.4.161+ #9
[ 221.628908] Hardware name: sun50iw10 (DT)
[ 221.633391] Call trace:
[ 221.636136]  dump_backtrace+0x0/0x140
[ 221.640237]  show_stack+0x14/0x20
[ 221.643951]  dump_stack+0xb0/0xd4
[ 221.647659]  panic+0x16c/0x410
[ 221.651078]  sysrq_handle_reboot+0x0/0x20
[ 221.655574]  __handle_sysrq+0x124/0x190
[ 221.659867]  write_sysrq_trigger+0xb0/0xb8
[ 221.664460]  proc_reg_write+0x58/0xd0
[ 221.668565]  __vfs_write+0x18/0x40
[ 221.672376]  vfs_write+0xb4/0x1a0
[ 221.676087]  ksys_write+0x64/0xf0
[ 221.679796]  __arm64_sys_write+0x14/0x20
[ 221.684190]  el0_svc_common.constprop.4+0x60/0x188
[ 221.689553]  el0_svc_handler+0x6c/0x88
[ 221.693745]  el0_svc+0x8/0x640
[ 221.697165] SMP: stopping secondary CPUs
[ 221.701677] Kernel Offset: disabled
[ 221.705609] CPU features: 0x00010002,20002004
[ 221.710492] Memory Limit: none
[ 221.713918] ---[ end Kernel panic - not syncing: sysrq triggered crash ]---
[ 221.721763] crashdump enter
```

在 5.15 上进行测试时：

```
/ # echo "c" > /proc/sysrq-trigger
[   91.650801] sysrq: Trigger a crash
[   91.654684] Kernel panic - not syncing: sysrq triggered crash
[   91.661120] CPU: 0 PID: 407 Comm: ash Not tainted 5.15.41 #13
[   91.667560] Hardware name: sun50iw10 (DT)
[   91.672049] Call trace:
[   91.674782]  dump_backtrace+0x0/0x1c8
[   91.678901]  show_stack+0x14/0x20
[   91.682619]  dump_stack_lvl+0x78/0x98
[   91.686729]  dump_stack+0x14/0x2c
[   91.690447]  panic+0x16c/0x350
[   91.693872]  sysrq_reset_seq_param_set+0x0/0x90
[   91.698954]  __handle_sysrq+0xa8/0x1a8
[   91.703155]  write_sysrq_trigger+0x84/0xc0
[   91.707746]  proc_reg_write+0xa4/0x120
[   91.711949]  vfs_write+0xb0/0x3c0
[   91.715669]  ksys_write+0x64/0xf0
[   91.719386]  __arm64_sys_write+0x14/0x20
[   91.723785]  invoke_syscall+0x4c/0x110
[   91.727993]  el0_svc_common.constprop.4+0x70/0x100
[   91.733363]  do_el0_svc+0x6c/0x88
[   91.737078]  el0_svc+0x1c/0x58
[   91.740502]  el0t_64_sync_handler+0x8c/0xb0
[   91.745189]  el0t_64_sync+0x16c/0x170
[   91.749294] SMP: stopping secondary CPUs
[   91.753755] Kernel Offset: 0x80000 from 0xffffffc008000000
[   91.759894] PHYS_OFFSET: 0x40000000
[   91.763797] CPU features: 0x0,00004801,00000842
[   91.768869] Memory Limit: none
[   97.773019] crashdump enter
```

对比两份 Log 不难发现，5.15 内核会打印Kernel Offset: 0x80000 from 0xffffffc008000000，而 5.4 内核即使开启 **CONFIG_RANDOMIZE_BASE** 配置选项也只打印Kernel Offset: disabled，其中0x80000就是通过 kaslr_offset 函数获取到的偏移量。

📖 说明

crash 和 linux 内核是紧密耦合的，会随着内核的变化持续更新，它是向前兼容的，新的 crash 工具可以分析老内核的转存文件。
如果你的内核版本较新，crash 无法解析，可以尝试安装最新的crash工具。
源码仓库：https://github.com/crash-utility/crash

5.5 crash 工具编译方法

- 进入 crash 目录，编译 64 位 ARM 的 crash：make target=ARM64
- **无需交叉编译**。编译成功后在当前目录下生成对应的 crash 文件。
- 可以将多余的符号去除：strip -s crash

官方文档：https://crash-utility.github.io/crash_whitepaper.html

5.6 crash 工具调试方法

若上述调试方法仍无法使用，可在解析命令后添加-d 4对 crash 工具进行 debug，也可直接在 crash 源码内添加打印信息进行确认解析是否正常，对比系统启动后通过符号表获取的相关参数是否一致，命令如下：

```
cat /proc/kallsyms | grep kernel_config_data
ffffffc00884f580 R kernel_config_data
ffffffc0088567e0 R kernel_config_data_end
```

```
diff --git a/kernel.c b/kernel.c
index a521ef3..acea74a 100644
--- a/kernel.c
+++ b/kernel.c
@@ -10494,6 +10494,7 @@ again:
        /*
         * Later versions put the magic number before the compressed data.
         */
+       printf("sp: 0x%lx\n", sp->value);
        if (readmem(sp->value - 8, KVADDR, &magic, 8,
            "kernel_config_data MAGIC_START", RETURN_ON_ERROR) &&
            STRNEQ(&magic, MAGIC_START)) {
```

5.7 crash 工具 GKI 固件支持

目前 vmlinux 已经打包到固件里面，2020.09 后所有的平台，及使用 dev 分支的所有平台默认支持。

dailybuild 目录 debug 下的文件 android-vmlinux 暂时未适配 gki，可能不正确。

如需调试，请优先考虑从固件中解压 vmlinux。

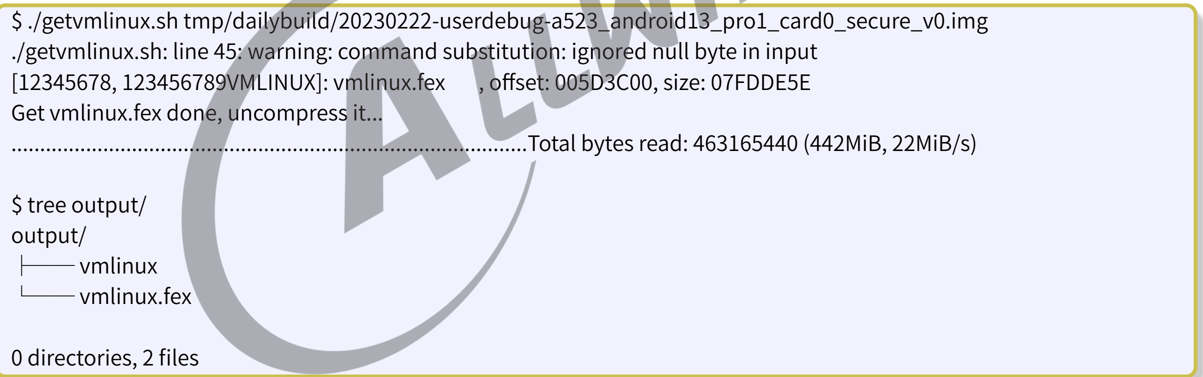
使用方法如下：

- 使用脚本进行提取 vmlinux，脚本存放路径为：longan/build/getvmlinux.sh。

```
./getvmlinux.sh <aw-format-firmware>
```

其中为全志格式的包含 vmlinux 的固件。

- 运行成功后，会在脚本目录下生成 output 目录，目录里面包含 vmlinux.fex（vmlinux 的.tar.bz2 格式压缩文件）与 vmlinux（原始 vmlinux 文件）。



```
$ ./getvmlinux.sh tmp/dailybuild/20230222-userdebug-a523_android13_pro1_card0_secure_v0.img
./getvmlinux.sh: line 45: warning: command substitution: ignored null byte in input
[12345678, 123456789VMLINUX]: vmlinux.fex      , offset: 005D3C00, size: 07FDDE5E
Get vmlinux.fex done, uncompress it...
..........................................................................................Total bytes read: 463165440 (442MiB, 22MiB/s)

$ tree output/
output/
├── vmlinux
└── vmlinux.fex

0 directories, 2 files
```

ALLWINNER

版权所有 © 珠海全志科技股份有限公司。保留一切权利

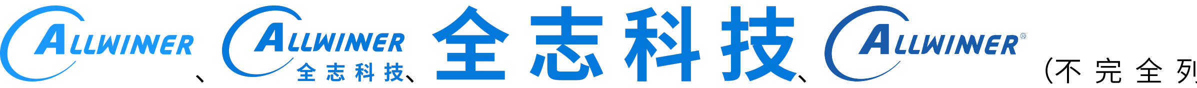
著作权声明

版权所有 ©2023 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明



（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。