crash 分析 vmcore 入门

配置项编号			
密级	公开	版本	V0.1.0
拟制		日期	
审核		日期	
批准		日期	

版本变更记录

版本	修订说明	修订人	修订时间
0.1.0	创建	高英杰	2024/09/27

目录

1. 概述

文档包括服务器版下载 vmlinux、内核源码的步骤；crash 的简单使用方法；x86_64 与 arm64 的函数调用与寄存器约定；AT&T、arm64 汇编语法的简单概述/与 intel 汇编语法的差异；最后以 x86_64 与 arm64 的两个实际案例进行简单分析。

2. 分析前的准备工作（服务器版）

服务器版默认配置了 kdump 工具，可以通过 systemctl status kdump.service 查看 kdump 服务是否处于开启状态（显示 active (exited)）。当内核发生崩溃时默认在/var/crash/目录下生成 vmcore 及最后一次启动期间的内核日志 vmcore-dmesg.txt。

拿到 vmcore 后需要首先确定发生崩溃的内核版本及架构，可以通过查看 vmcore-dmesg.txt 的首行来确定。

```
gyj@gyj-PC:~/Downloads/uos-kdump/10.7.240.60/127.0.0.1-2024-09-16-05%3A27%3A52$ head vmcore-dmesg.txt -n 1
[    0.000000] Linux version 4.19.90-2305.1.0.0199.78.uel20.x86_64 (mockbuild@UOS-x86-build2) (gcc version 7.
```

图 1

分析 vmcore 需要准备对应架构的 crash 工具、对应版本 vmlinux 及内核源码三部分。

2.1. 准备环境

2.1.1. crash 工具

crash 工具可以选择在对应架构的服务器通过 yum install crash 安装 crash 包来获取（桌面版的 crash 版本较低，会出现部分命令选项无法使用，sys 命令总结的运行时间可能有误等问题，可以从 GitHub 取最新 crash 代码手动编译使用）。

2.1.2. vmlinux

vmlinux 需要通过 koji[1]来寻找，找到对应内核版本后点击对应条目后可以看到各种内核相关 rpm 包。

图 2

选择 kernel-debuginfo-xxx.rpm（不要选成 kernel-debug-debuginfo，见图 2）复制下载地址，在服务器上下载（北京这边本地无法下载）。下载完成后使用如下命令解压 rpm 包至 kernel-debug 目录（cpio 的-D 选项指定解压目录）。

rpm2cpio kernel-debug-xxx.rpm | cpio -div -D kernel-debug

随后使用 find 命令寻找 vmlinux，选择路径不带 debug 的 vmlinux 即可（图 3）。

```
[root@localhost UBXC006520]# find kernel-debuginfo/ -name vmlinux
kernel-debuginfo/usr/lib/debug/lib/modules/4.19.90-2305.1.0.0199.78.uel20.x86_64/vmlinux
kernel-debuginfo/usr/lib/debug/lib/modules/4.19.90-2305.1.0.0199.78.uel20.x86_64+debug/vmlinux
```

图 3

2.1.3. 内核源码

内核源码可以直接从 gerrit 仓库获取切换至对应 tag，或者从 koji 获取（依次按图 4 图 5 图 6 点击即可，同样需要在服务器下载，rpm 包解压方式也相同，解压后的 kernel-4.19.90.tar.gz 即为内核源码压缩包，再次解压即可）。

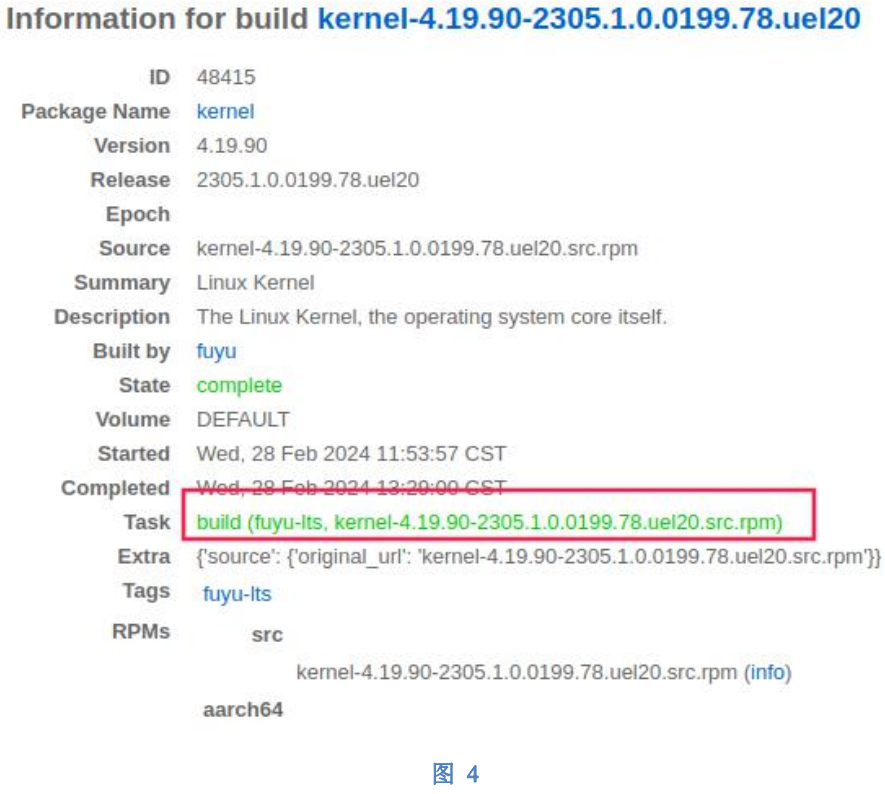

图 4

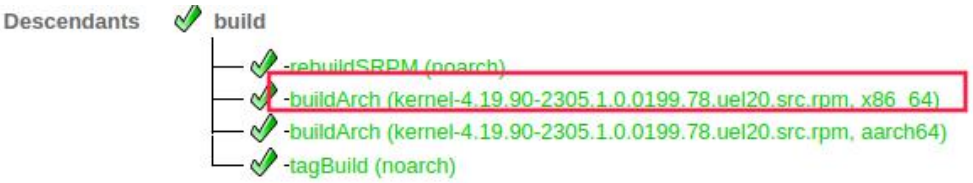

图 5

图 6

2.2. 开始分析

准备好 crash 工具、vmlinux 及内核源码后，执行下述命令即可开始调试。（将 vmlinux、vmcore 放入内核源码目录下时，可以通过 dis -s 查看对应 c 代码）

crash vmlinux vmcore

```
      KERNEL: vmlinux
    DUMPFILE: vmcore_10.7  [PARTIAL DUMP]
        CPUS: 24
        DATE: Mon Sep 16 05:27:42 CST 2024
      UPTIME: 72 days, 09:55:28
LOAD AVERAGE: 1.33, 0.65, 0.42
       TASKS: 1041
    NODENAME: yzpubcredis60
     RELEASE: 4.19.90-2305.1.0.0199.78.uel20.x86_64
     VERSION: #1 SMP Wed Feb 28 12:31:25 CST 2024
     MACHINE: x86_64  (2199 Mhz)
      MEMORY: 96 GB
       PANIC: "BUG: unable to handle kernel paging request at ffff94ca46fc1050"
         PID: 0
     COMMAND: "swapper/20"
        TASK: ffff94ca468c6000  (1 of 24)  [THREAD_INFO: ffff94ca468c6000]
         CPU: 20
       STATE: TASK_RUNNING (PANIC)

crash>
```

图 7

执行上述命令后等待 crash 加载 vmcore 完成后会显示对 vmcore 的总结（见图 7）。各行含义如下：

CPUS:总 cpu 核数

DATE:内核发生崩溃的时间点

UPTIME:内核崩溃前的运行时长

TASKS:崩溃前运行的进程总数

RELEASE:内核版本

MACHINE:内核架构

MEMORY:内存总量

PANIC:发生 panic 的直接原因

PID:发生 panic 的进程号

COMMAND:发生 panic 的进程命令

TASK:发生 panic 的进程 task_struct 结构体指针

CPU:发生 panic 进程所在 cpu

3. crash 常用命令

进入 crash 后执行 help 可以查看 crash 支持的命令：

```
crash> help

*               files           mod             sbitmapq        union
alias           foreach         mount           search          vm
ascii           fuser           net             set             vtop
bpf             gdb             p               sig             waitq
bt              help            ps              struct          whatis
btop            ipcs            pte             swap            wr
dev             irq             ptob            sym             q
dis             kmem            ptov            sys
eval            list            rd              task
exit            log             repeat          timer
extend          mach            runq            tree
```

图 8

下面着重介绍一下高频使用的几个命令

3.1. 使用 bt 查看栈

crash 默认将当前上下文设置为发生 panic 的进程。使用 bt 命令可以查看发生 panic 时的指令地址，当时的寄存器值及函数调用栈。

3.1.1. bt

直接执行 bt 命令显示当前 cpu 的栈，图 9 红框中 exception RIP 显示了发生 panic 时的指令地址，其余为各个寄存器内的值，后续分析需要基于各个寄存器的值来还原代码中各个变量的值。

```
crash> bt
PID: 0        TASK: ffff94ca468c6000  CPU: 20   COMMAND: "swapper/20"
 #0 [ffff94e11e303ae8] machine_kexec at ffffffffb8257b0f
 #1 [ffff94e11e303b40] __crash_kexec at ffffffffb835b981
 #2 [ffff94e11e303c00] crash_kexec at ffffffffb835c85d
 #3 [ffff94e11e303c18] oops_end at ffffffffb822592f
 #4 [ffff94e11e303c38] no_context at ffffffffb8266ee6
 #5 [ffff94e11e303c90] __do_page_fault at ffffffffb8267710
 #6 [ffff94e11e303d00] do_page_fault at ffffffffb8267b41
 #7 [ffff94e11e303d30] page_fault at ffffffffb8e011ce
    [exception RIP: select_task_rq_fair+405]
    RIP: ffffffffb82ea635  RSP: ffff94e11e303de0  RFLAGS: 00010086
    RAX: ffff94e11e322ac0  RBX: 0000000000000014  RCX: 00000000000006dc
    RDX: 0000000000000001  RSI: 0000000000000002  RDI: ffff94e0a43b9410
    RBP: 0000000000000014   R8: 0000000000000010   R9: 0000000000000000
    R10: ffff94e11e303f20  R11: 0000000000000000  R12: ffff94ca46fc1000
    R13: 0000000000000000  R14: 0000000000022ac0  R15: ffff94e097c13000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #8 [ffff94e11e303e88] try_to_wake_up at ffffffffb82e3090
 #9 [ffff94e11e303e98] rcu_advance_cbs at ffffffffb832c080
#10 [ffff94e11e303f08] hrtimer_wakeup at ffffffffb833b3be
#11 [ffff94e11e303f18] hrtimer_wakeup at ffffffffb833b3be
#12 [ffff94e11e303f20] __hrtimer_run_queues at ffffffffb833b6b8
#13 [ffff94e11e303f80] hrtimer_interrupt at ffffffffb833bea5
#14 [ffff94e11e303fd8] smp_apic_timer_interrupt at ffffffffb8e0271a
#15 [ffff94e11e303ff0] apic_timer_interrupt at ffffffffb8e01c7f
--- <IRQ stack> ---
#16 [ffff94ca468d3df8] apic_timer_interrupt at ffffffffb8e01c7f
    [exception RIP: native_safe_halt+14]
    RIP: ffffffffb8c6e87e  RSP: ffff94ca468d3ea8  RFLAGS: 00000246
    RAX: ffffffffb8c6e490  RBX: 0000000000000014  RCX: 0000000000000001
    RDX: 0000000000000000  RSI: 0000000000000087  RDI: ffff94e11e323900
    RBP: 0000000000000014   R8: ffff94e11e31d8c0   R9: 0000000000000000
    R10: 0000000000000000  R11: 0000000000000000  R12: 0000000000000000
    R13: 0000000000000000  R14: 0000000000000000  R15: 0000000000000000
    ORIG_RAX: ffffffffffffff13  CS: 0010  SS: 0018
#17 [ffff94ca468d3ea8] default_idle at ffffffffb8c6e4ac
#18 [ffff94ca468d3ed0] do_idle at ffffffffb82e779f
#19 [ffff94ca468d3f10] cpu_startup_entry at ffffffffb82e7a6f
#20 [ffff94ca468d3f30] start_secondary at ffffffffb824c32d
#21 [ffff94ca468d3f50] secondary_startup_64 at ffffffffb82000e6
crash>
```

图 9

3.1.2. bt -a

查看所有 cpu 上的栈，有时直接触发 panic 的代码并不是问题的根因，可以查看其他 cpu 栈是否有线索。

3.1.3. bt -f/F

直接直接 bt 命令只会显示直接触发 panic 处的寄存器值，其余部分只显示函数调用栈。执行 bt -f/F（-F 参数会显示栈中 slab 对象的名称而非地址）可以显示完整的各级调用栈（包括被压栈的数据），可以调用栈上其他函数的参数及执行流程。

3.2. 使用 dis 查看汇编/c 代码

这里以图 9 来举例，查看 select_task_rq_fair+405（对应地址在 RIP 里为 ffffffffb82ea635）的汇编代码。

3.2.1. bt -rl

bt -rl ffffffffb82ea635（或 bt -rl select_task_rq_fair+405）可以查看 select_task_rq_fair 函数从头至 405 偏移处的所有汇编代码及对应的 c 代码位置（指示的 c 代码位置可能并不准确）。

3.2.2. bt -s

bt -s ffffffffb82ea635（或 bt -s select_task_rq_fair+405）可以查看该指令对应的 c 代码（需要在内核源码目录下执行 crash 命令才能使用，我试了使用 dir 命令指令内核源码的方式但是还是不能用）。

3.3. 使用*、struct、union 查看结构体内容

struct、union 命令可以简写为*命令（如果结构体名称不是命令名称*也可以省略，见图 10）

3.3.1. 查看结构体定义

```
struct fdtable {
    unsigned int max_fds;
    struct file **fd;
    unsigned long *close_on_exec;
    unsigned long *open_fds;
    unsigned long *full_fds_bits;
    struct callback_head rcu;
    unsigned long *dup_fds;
    unsigned long *lazy_dup_fds;
}
```

图 10

3.3.1.1. 使用-o 参数查看各成员的偏移值

```
crash> fdtable -o
struct fdtable {
   [0] unsigned int max_fds;
   [8] struct file **fd;
  [16] unsigned long *close_on_exec;
  [24] unsigned long *open_fds;
  [32] unsigned long *full_fds_bits;
  [40] struct callback_head rcu;
  [56] unsigned long *dup_fds;
  [64] unsigned long *lazy_dup_fds;
}
SIZE: 72
```

图 11

3.3.1.2. 查看某一成员的偏移值

```
crash> fdtable.fd
struct fdtable {
   [8] struct file **fd;
}
```

图 12

3.3.2. 查看结构体成员的值

```
crash> fdtable 0xffff94ca467f65e8
struct fdtable {
  max_fds = 64,
  fd = 0xffff94ca467f6670,
  close_on_exec = 0xffff94ca467f6648,
  open_fds = 0xffff94ca467f6650,
  full_fds_bits = 0xffff94ca467f6658,
  rcu = {
    next = 0x0,
    func = 0x0
  },
  dup_fds = 0xffff94ca467f6660,
  lazy_dup_fds = 0xffff94ca467f6668
}
```

图 13

3.3.2.1. 查看某一成员的值

```
crash> fdtable.fd 0xffff94ca467f65e8
  fd = 0xffff94ca467f6670,
```

图 14

3.3.2.2. 查看某几个成员的值

```
crash> fdtable.fd,open_fds 0xffff94ca467f65e8
  fd = 0xffff94ca467f6670,
  open_fds = 0xffff94ca467f6650,
```

图 15

3.4. 使用 p 命令进行地址计算

分析汇编代码时经常需要进行地址计算，p 命令可以很方便的进行算数运算（加减乘除取余 +,-,*,/,%）、位运算(左移、右移、与、或、非、异或 <<,>>,&,|,~,^)，当然也可以用了输出地址/符号的内容。

3.5. 使用 x 命令显示内存地址内容

x 命令可以很方便的以各种格式输出地址的内容

命令格式为：x /[输出数量]输出格式[每个输出单元的大小] 地址

输出格式包括：o(octal), x(hex), d(decimal), u(unsigned decimal),

t(binary), f(float), a(address), i(instruction), c(char), s(string)

and z(hex, zero padded on the left).

每个输出单元的大小包括： b(byte), h(halfword), w(word), g(giant, 8 bytes).

3.6. 其他

files:查看当前进程打开的文件信息

kmem：查看内存相关信息

list：遍历显示链表内容

log：显示最后一次启动期间的内核日志，内容与 vmcore-dmesg.txt 相同

mount:查看文件系统挂载信息

ps：查看运行的进程

search：在内存中查找值

set:切换上下文

swap：查看交换分区信息

vtop:将虚拟地址转化为物理地址，并显示各级页表信息

……

4. 寄存器约定与函数参数传递

在分析 vmcore 时，需要解析各个寄存器对应的变量，这就需要了解函数传递参数时使用的寄存器约定。

4.1. x86_64 架构

如果函数参数不超过 6 个，依次放在 rdi,rsi,rdx,rcx,r8d,r9d 中，返回值放在 rax 中。如果参数超过六个，多余的参数从右向左依次压入栈中[2]。

4.2. arm64 架构

如果函数参数不超过 8 个，依次放在 x0-x7 中，返回值放在 x0 中。参数超过 8 个，多余的参数从右向左依次压入栈中[2]。

5. 简单汇编知识

5.1. x86_64

x86_64 使用 AT&T 汇编语法，与 intel 语法稍有不同。如果遇到不认识的汇编指令可以查阅 intel 汇编手册（见[3]）。

1. 寄存器前缀`%`,立即数前缀`$`,十六进制数前缀`0x`

2. 源操作数在前，目的操作数在后

3. 内存寻址时基址寄存器使用`()`括起来，格式为`%segreg:disp(base,index,scale)`。scale/disp 中使用立即数不用使用`$`

这种寻址方式常常用在访问数据结构数组中某个特定元素内的一个字段，其中，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素还是一个结构，则 disp 为具体字段在结构中的位移

4. 操作码后缀：`l`长整数，`w`字，`b`字节

5.2. arm64

arm64 汇编指令可以查阅 arm 开发者手册（见[4]）。arm64 汇编语法特点如下：

1. 目标操作数在前，源操作数在后

2. 使用`X`,`W`前缀来区分 64 位与 32 位寄存器

3. 使用前缀`#`表示立即数

4. 使用`[base, offset]`的格式表示内存地址

6. 案例

下面使用 x86_64 与 arm64 的两个例子来简单介绍下分析流程。

6.1. 案例 1(x86_64)

【UBXC006520】【重庆农村商业银行服务器】【二级】服务器内核崩溃导致系统重启

6.1.1. rip 处分析

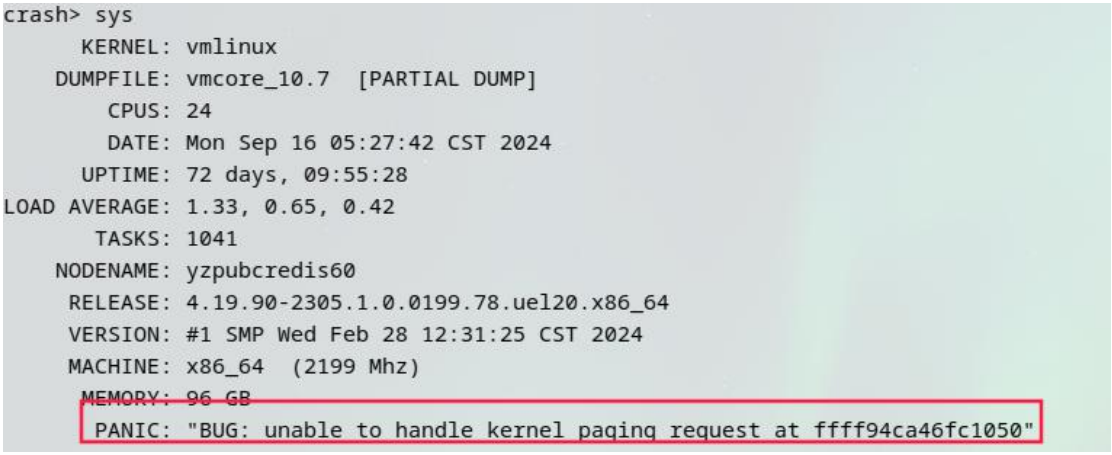

图 16

运行 crash 命令加载完 vmcore 后会首先显示如图 16 的总结信息。根据此信息或 vmcore-dmesg.txt 可以得到内核崩溃的直接原因是访问地址 ffff94ca46fc1050 时发生了无法处理的页请求。接着需要排查代码执行流程，确认是否应该访问该地址。

执行 bt 命令查看栈（见图 17），得到当前执行的指令地址 select_task_rq_fair+405（ffffffffb82ea635）。

执行 dis -rl select_task_rq_fair+405 查看对应汇编代码（见图 18）。对应汇编语句为 mov 0x50(%r12),%eax，将 R12+0x50 地址处的内存值赋值给 eax。

```
crash> bt
PID: 0        TASK: ffff94ca468c6000  CPU: 20   COMMAND: "swapper/20"
 #0 [ffff94e11e303ae8] machine_kexec at ffffffffb8257b0f
 #1 [ffff94e11e303b40] __crash_kexec at ffffffffb835b981
 #2 [ffff94e11e303c00] crash_kexec at ffffffffb835c85d
 #3 [ffff94e11e303c18] oops_end at ffffffffb822592f
 #4 [ffff94e11e303c38] no_context at ffffffffb8266ee6
 #5 [ffff94e11e303c90] __do_page_fault at ffffffffb8267710
 #6 [ffff94e11e303d00] do_page_fault at ffffffffb8267b41
 #7 [ffff94e11e303d30] page_fault at ffffffffb8e011ce
    [exception RIP: select_task_rq_fair+405]
    RIP: ffffffffb82ea635  RSP: ffff94e11e303de0  RFLAGS: 00010086
    RAX: ffff94e11e322ac0  RBX: 0000000000000014  RCX: 00000000000006dc
    RDX: 0000000000000001  RSI: 0000000000000002  RDI: ffff94e0a43b9410
    RBP: 0000000000000014   R8: 0000000000000010   R9: 0000000000000000
    R10: ffff94e11e303f20  R11: 0000000000000000  R12: ffff94ca46fc1000
    R13: 0000000000000000  R14: 0000000000022ac0  R15: ffff94e097c13000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #8 [ffff94e11e303e88] try_to_wake_up at ffffffffb82e3090
```

图 17

```
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ke
rnel/sched/fair.c: 6900
0xffffffffb82ea630 <select_task_rq_fair+400>:   test   %r12,%r12
0xffffffffb82ea633 <select_task_rq_fair+403>:   je     0xffffffffb82ea69a <select_task_rq_fair+506>
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ke
rnel/sched/fair.c: 6901
0xffffffffb82ea635 <select_task_rq_fair+405>:   mov    0x50(%r12),%eax
```

图 18

结合图 18 中显示该汇编语句对应 kernel/sched/fair.c: 6901 行 c 代码（见图 20）。查看 tmp 结构体名称（图 19）及 flags 成员变量的偏移（图 21）可以确定对应的 c 语句为 tmp->flags。

```
6874        struct sched_domain *tmp, *sd = NULL;
```

图 19

```
6900        for_each_domain(cpu, tmp) {
6901                if (!(tmp->flags & SD_LOAD_BALANCE))
6902                        break;
```

图 20

```
crash> sched_domain.flags
struct sched_domain {
   [80] int flags;
}
crash> px 80
$1 = 0x50
```

图 21

确认下 crash 是否也无法访问该地址。使用 x 命令发现 crash 可以正常访问该地址，其值为 0x102f（见图 22）。查阅 sched_domain 标记位取值（图 23），其值 0x102f 未见异常。

```
crash> x /xw 0xffff94ca46fc1050
0xffff94ca46fc1050:     0x0000102f
```

图 22

```
20 #define SD_LOAD_BALANCE         0x0001  /* Do load balancing on this domain. */
21 #define SD_BALANCE_NEWIDLE      0x0002  /* Balance when about to become idle */
22 #define SD_BALANCE_EXEC         0x0004  /* Balance on exec */
23 #define SD_BALANCE_FORK         0x0008  /* Balance on fork, clone */
24 #define SD_BALANCE_WAKE         0x0010  /* Balance on wakeup */
25 #define SD_WAKE_AFFINE          0x0020  /* Wake task to waking CPU */
26 #define SD_ASYM_CPUCAPACITY     0x0040  /* Groups have different max cpu capacities */
27 #define SD_SHARE_CPUCAPACITY    0x0080  /* Domain members share cpu capacity */
28 #define SD_SHARE_POWERDOMAIN    0x0100  /* Domain members share power domain */
29 #define SD_SHARE_PKG_RESOURCES  0x0200  /* Domain members share cpu pkg resources */
30 #define SD_SERIALIZE            0x0400  /* Only a single load balancing instance */
31 #define SD_ASYM_PACKING         0x0800  /* Place busy groups earlier in the domain */
32 #define SD_PREFER_SIBLING       0x1000  /* Prefer to place tasks in a sibling domain */
33 #define SD_OVERLAP              0x2000  /* sched_domains of this level overlap */
34 #define SD_NUMA                 0x4000  /* cross-node balancing */
35 #define SD_NUMA_NO_MEMORY       0x8000  /* cross-node balancing */
```

图 23

tmp->flags 值没发现异常，再看下 tmp 结构体有没有什么异常（见图 24）。

`parent`和`child`字段为`0x0`，这表明该调度域没有父调度域或子调度域，可能是一个孤立的调度域。

`groups`字段指向一个有效的内存地址，表示有调度组存在。

`min_interval`为 24，`max_interval`为 48。这表明调度域的负载平衡时间间隔是合理的。

`busy_factor`为 32，`imbalance_pct`为 125。`imbalance_pct`通常表示负载不平衡的阈值，125 表示当负载不平衡超过 25%时会触发负载平衡。

`nr_balance_failed`为 0，表示没有负载平衡失败的记录。

`lb_count`、`lb_failed`、`lb_balanced`等数组的值均为 0，表明在负载平衡过程中没有发生任何操作，可能是因为当前没有负载平衡的需求。

`span_weight`为 24，表示调度域的跨度权重。

……

以上成员值都未见明显异常，那我们再向前排查，分析从进入 select_task_rq_fair 的

执行流程是否正确。

```
crash> sched_domain 0xffff94ca46fc1000
struct sched_domain {
  parent = 0x0,
  child = 0x0,
  groups = 0xffff94ca46fb8100,
  min_interval = 24,
  max_interval = 48,
  busy_factor = 32,
  imbalance_pct = 125,
  cache_nice_tries = 1,
  busy_idx = 2,
  idle_idx = 1,
  newidle_idx = 0,
  wake_idx = 0,
  forkexec_idx = 0,
  smt_gain = 0,
  nohz_idle = 0,
  flags = 4143,
  level = 2,
  last_balance = 4920590101,
  balance_interval = 24,
  nr_balance_failed = 0,
  max_newidle_lb_cost = 639947,
  next_decay_max_lb_cost = 4920590180,
  avg_scan_cost = 0,
  lb_count = {0, 0, 0},
  lb_failed = {0, 0, 0},
  lb_balanced = {0, 0, 0},
  lb_imbalance = {0, 0, 0},
  lb_gained = {0, 0, 0},
  lb_hot_gained = {0, 0, 0},
  lb_nobusyg = {0, 0, 0},
  lb_nobusyq = {0, 0, 0},
  alb_count = 0,
  alb_failed = 0,
  alb_pushed = 0,
  sbe_count = 0,
  sbe_balanced = 0,
  sbe_pushed = 0,
  sbf_count = 0,
  sbf_balanced = 0,
  sbf_pushed = 0,
  ttwu_wake_remote = 0,
  ttwu_move_affine = 0,
  ttwu_move_balance = 0,
  name = 0xffffffffb96cad2b "DIE",
  {
    private = 0xffff94ca46ed7c98,
    rcu = {
      next = 0xffff94ca46ed7c98,
      func = 0x0
    }
  },
  shared = 0x0,
  span_weight = 24,
  span = 0xffff94ca46fc1138
}
crash>
```

图 24

6.1.2. rip 所在函数分析

bt 显示的寄存器取值是 rip 指令执行处的各个寄存器的值，在这种情况下无法直接通过函数传参约定直接确定函数的参数值(此时与函数传参相关的寄存器值可能已经在执行过程中被覆盖了)，需要根据汇编代码正向+逆向相结合来确认，同时要特别注意各种跳转语句（需要注意 c 代码中没有任何跳转的一段代码，在汇编语句中可能出现跳转）。

```
6870 static int
6871 select_task_rq_fair(struct task_struct *p, int prev_cpu, int sd_flag, int wake_flags)
6872 {
6873         unsigned long time;
6874         struct sched_domain *tmp, *sd = NULL;
6875         int cpu = smp_processor_id();
6876         int new_cpu = prev_cpu;
6877         int want_affine = 0;
6878         int sync = (wake_flags & WF_SYNC) && !(current->flags & PF_EXITING);
6879 #ifdef CONFIG_QOS_SCHED_DYNAMIC_AFFINITY
6880         int idlest_cpu = 0;
6881 #endif
6882
6883         time = schedstat_start_time();
6884
6885 #ifdef CONFIG_QOS_SCHED_DYNAMIC_AFFINITY
6886         set_task_select_cpus(p, &idlest_cpu, sd_flag);
6887 #endif
6888
6889         if (sd_flag & SD_BALANCE_WAKE) {
6890                 record_wakee(p);
6891                 want_affine = !wake_wide(p) && !wake_cap(p, cpu, prev_cpu)
6892 #ifdef CONFIG_QOS_SCHED_DYNAMIC_AFFINITY
6893                               && cpumask_test_cpu(cpu, p->select_cpus);
6894 #else
6895                               && cpumask_test_cpu(cpu, &p->cpus_allowed);
6896 #endif
6897         }
6898
6899         rcu_read_lock();
6900         for_each_domain(cpu, tmp) {
6901                 if (!(tmp->flags & SD_LOAD_BALANCE))
6902                         break;
6903
```

图 25

首先分析下至发生异常处（6901 行）的 c 代码，select_task_rq_fair 一共有 4 个参数。其中 wake_flags 只影响 sync（6878 行）的取值，而 sync 在其后并没有用到，因此可以先不用解析 wake_flags 的值。

按照 x86_64 的寄存器约定，rdi, rsi, rdx 依次传递函数前三个参数。

select_task_rq_fair+9 处（图 26）将 rdi 放入 r15 中，结合寄存器数值（图 27）使用 task_struct 0xffff94e097c13000 查看其值（图 28）符合 task_struct 结构体内

容，再查看整个 select_task_rq_fair 汇编代码（因为汇编中的跳转语句，只查看 select_task_rq_fair+405 可能会遗漏），发现没有对 r15 进行过修改，因此第一个参数 p 还在 r15 中，值为 0xffff94e097c13000。

```
crash> dis -rl select_task_rq_fair+405
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ke
rnel/sched/fair.c: 6872
0xffffffffb82ea4a0 <select_task_rq_fair>:       nopl   0x0(%rax,%rax,1) [FTRACE NOP]
0xffffffffb82ea4a5 <select_task_rq_fair+5>:     push   %r15
0xffffffffb82ea4a7 <select_task_rq_fair+7>:     push   %r14
0xffffffffb82ea4a9 <select_task_rq_fair+9>:     mov    %rdi,%r15
0xffffffffb82ea4ac <select_task_rq_fair+12>:    push   %r13
0xffffffffb82ea4ae <select_task_rq_fair+14>:    push   %r12
0xffffffffb82ea4b0 <select_task_rq_fair+16>:    push   %rbp
0xffffffffb82ea4b1 <select_task_rq_fair+17>:    push   %rbx
0xffffffffb82ea4b2 <select_task_rq_fair+18>:    mov    %esi,%ebp
0xffffffffb82ea4b4 <select_task_rq_fair+20>:    sub    $0x78,%rsp
0xffffffffb82ea4b8 <select_task_rq_fair+24>:    mov    %edx,0xc(%rsp)
0xffffffffb82ea4bc <select_task_rq_fair+28>:    mov    %gs:0x28,%rax
0xffffffffb82ea4c5 <select_task_rq_fair+37>:    mov    %rax,0x70(%rsp)
0xffffffffb82ea4ca <select_task_rq_fair+42>:    xor    %eax,%eax
```

图 26

```
[exception RIP: select_task_rq_fair+405]
RIP: ffffffffb82ea635  RSP: ffff94e11e303de0  RFLAGS: 00010086
RAX: ffff94e11e322ac0  RBX: 0000000000000014  RCX: 00000000000006dc
RDX: 0000000000000001  RSI: 0000000000000002  RDI: ffff94e0a43b9410
RBP: 0000000000000014   R8: 0000000000000010   R9: 0000000000000000
R10: ffff94e11e303f20  R11: 0000000000000000  R12: ffff94ca46fc1000
R13: 0000000000000000  R14: 0000000000022ac0  R15: ffff94e097c13000
ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
```

图 27

```
crash> task_struct 0xffff94e097c13000
struct task_struct {
  thread_info = {
    flags = 128,
    status = 0
  },
  state = 512,
  stack = 0xffff94e0a11bc000,
  usage = {
    counter = 1
  },
  flags = 1077936192,
  ptrace = 0,
  wake_entry = {
    next = 0x0
  },
  on_cpu = 0,
  cpu = 20,
```

图 28

esi 传递 prev_cpu，当前值为 0x2，select_task_rq_fair+18 中将值赋给 ebp,当前值为 0x14（见图 26 图 27）。但是因为后续汇编代码中有对 esi, ebp 值的修改，因此当前还不能判断其中的值是否被覆盖，prev_cpu 的值需要根据后续代码执行流程解析（流程太长，在此不做展开）。

edx 传递 sd_flag，select_task_rq_fair+24 中被放入 0xc(%rsp)中，rsp 的值通常只在函数起始处减少，返回时增加。再结合 select_task_rq_fair+28~42，中将%gs:0x28 赋值给了 0x70(%rsp)，检验一下 rsp 的值是否有被覆盖（见图 29），经确认 rsp 值未被覆盖。再查看 select_task_rq_fair 中 0xc(%rsp)的值未被覆盖，因此 sd_flag 值为 0xc(%rsp)即 0x10。

```
crash> kmem -o | grep "CPU 20"
 CPU 20: ffff94e11e300000
crash> px 0xffff94e11e300000+0x28
$17 = 0xffff94e11e300028
crash> x /xg 0xffff94e11e300028
0xffff94e11e300028:     0x8004cc46324c2900
crash> px 0xffff94e11e303de0+0x70
$18 = 0xffff94e11e303e50
crash> x /xg 0xffff94e11e303e50
0xffff94e11e303e50:     0x8004cc46324c2900
crash> px 0xffff94e11e303de0+0xc
$19 = 0xffff94e11e303dec
crash> x /xw 0xffff94e11e303dec
0xffff94e11e303dec:     0x00000010
```

图 29

（注：gs 的值可以通过 kmem -o 对应 cpu 来查看，当前上下文在 cpu20 上；也可以通过 sys 命令或 vmcore-dmesg.txt 中打印的寄存器值来确认（见图 30））。

```
[6256153.308372] RSP: 0018:ffff94e11e303de0 EFLAGS: 00010086
[6256153.308375] RAX: ffff94e11e322ac0 RBX: 0000000000000014 RCX: 00000000000006dc
[6256153.308376] RDX: 0000000000000001 RSI: 0000000000000002 RDI: ffff94e0a43b9410
[6256153.308377] RBP: 0000000000000014 R08: 0000000000000010 R09: 0000000000000000
[6256153.308378] R10: ffff94e11e303f20 R11: 0000000000000000 R12: ffff94ca46fc1000
[6256153.308380] R13: 0000000000000000 R14: 0000000000022ac0 R15: ffff94e097c13000
[6256153.308381] FS:  0000000000000000(0000) GS:ffff94e11e300000(0000) knlGS:0000000000000000
[6256153.308383] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[6256153.308384] CR2: ffff94ca46e93e08 CR3: 000000010f96e000 CR4: 00000000003406e0
```

图 30

6.1.3. 调用栈中其他函数分析

函数被调用的起始处会将调用者的部分变量压入栈中，使用 bt -f（或 bt -F）可以查看被压入栈的数据，再结合上下文就可以还原出函数调用栈中各级函数的参数，并据此进行分析。

举个简单的例子，要查看 do_page_fault 的两个参数 regs、error_code 的值（见图 31），需要到__do_page_fault 中寻找。

```
1475 dotraplinkage void notrace
1476 do_page_fault(struct pt_regs *regs, unsigned long error_code)
```

图 31

```
crash> bt
PID: 0        TASK: ffff94ca468c6000  CPU: 20   COMMAND: "swapper/20"
 #0 [ffff94e11e303ae8] machine_kexec at ffffffffb8257b0f
 #1 [ffff94e11e303b40] __crash_kexec at ffffffffb835b981
 #2 [ffff94e11e303c00] crash_kexec at ffffffffb835c85d
 #3 [ffff94e11e303c18] oops_end at ffffffffb822592f
 #4 [ffff94e11e303c38] no_context at ffffffffb8266ee6
 #5 [ffff94e11e303c90] __do_page_fault at ffffffffb8267710
 #6 [ffff94e11e303d00] do_page_fault at ffffffffb8267b41
 #7 [ffff94e11e303d30] page_fault at ffffffffb8e011ce
    [exception RIP: select_task_rq_fair+405]
    RIP: ffffffffb82ea635  RSP: ffff94e11e303de0  RFLAGS: 00010086
```

图 32

```
crash> dis -rl do_page_fault+49
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ar
ch/x86/mm/fault.c: 1477
0xffffffffb8267b10 <do_page_fault>:     push   %r13
0xffffffffb8267b12 <do_page_fault+2>:   push   %r12
0xffffffffb8267b14 <do_page_fault+4>:   mov    %rsi,%r12
0xffffffffb8267b17 <do_page_fault+7>:   push   %rbp
0xffffffffb8267b18 <do_page_fault+8>:   push   %rbx
0xffffffffb8267b19 <do_page_fault+9>:   mov    %rdi,%rbp
0xffffffffb8267b1c <do_page_fault+12>:  sub    $0x8,%rsp
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/./
arch/x86/include/asm/paravirt.h: 57
0xffffffffb8267b20 <do_page_fault+16>:  mov    %cr2,%rax
0xffffffffb8267b23 <do_page_fault+19>:  nopl   0x0(%rax)
0xffffffffb8267b27 <do_page_fault+23>:  nopl   0x0(%rax,%rax,1)
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/./
include/linux/context_tracking.h: 53
0xffffffffb8267b2c <do_page_fault+28>:  xor    %ebx,%ebx
0xffffffffb8267b2e <do_page_fault+30>:  nopl   0x0(%rax,%rax,1)
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ar
ch/x86/mm/fault.c: 1485
0xffffffffb8267b33 <do_page_fault+35>:  mov    %rax,%rdx
0xffffffffb8267b36 <do_page_fault+38>:  mov    %r12,%rsi
0xffffffffb8267b39 <do_page_fault+41>:  mov    %rbp,%rdi
0xffffffffb8267b3c <do_page_fault+44>:  call   0xffffffffb8267650 <__do_page_fault>
0xffffffffb8267b41 <do_page_fault+49>:  nopl   0x0(%rax,%rax,1)
```

图 33

do_page_fault 在+44 处调用了__do_page_fault，观察此前的代码（见图 33），

发现 regs 在 rdi, rbp 中，error_code 在 rsi, r12 中。再根据__do_page_fault 的汇编代码（见图 34）函数开头有将 r12、rbp 压栈，因此可以根据栈内容解析 regs 与 error_code。

```
crash> dis -rl __do_page_fault+192
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1.0.0199.78.uel20.x86_64/ar
ch/x86/mm/fault.c: 1230
0xffffffffb8267650 <__do_page_fault>:   nopl   0x0(%rax,%rax,1) [FTRACE NOP]
0xffffffffb8267655 <__do_page_fault+5>: push   %r15
0xffffffffb8267657 <__do_page_fault+7>: push   %r14
0xffffffffb8267659 <__do_page_fault+9>: push   %r13
0xffffffffb826765b <__do_page_fault+11>:        push   %r12
0xffffffffb826765d <__do_page_fault+13>:        mov    %rdi,%r12
0xffffffffb8267660 <__do_page_fault+16>:        push   %rbp
0xffffffffb8267661 <__do_page_fault+17>:        push   %rbx
```

图 34

```
#5 [ffff94e11e303c90] __do_page_fault at ffffffffb8267710
    ffff94e11e303c98: 0000000000000078 0000000000000002
    ffff94e11e303ca8: 0000000000281b00 0000000000000000
    ffff94e11e303cb8: 0000000000000000 0000000000000000
    ffff94e11e303cc8: 8004cc46324c2900 0000000000000000
    ffff94e11e303cd8: ffff94e11e303d38 0000000000000000
    ffff94e11e303ce8: 0000000000000000 0000000000000000
    ffff94e11e303cf8: 0000000000000000 do_page_fault+49
#6 [ffff94e11e303d00] do_page_fault at ffffffffb8267b41
    ffff94e11e303d08: ffff94e11e303d30 0000000000000000
    ffff94e11e303d18: 0000000000000000 0000000000000000
    ffff94e11e303d28: 0000000000000000 page_fault+30
#7 [ffff94e11e303d30] page_fault at ffffffffb8e011ce
```

图 35

使用 bt -F 查看__do_page_fault 的栈内容（见图 35），再结合__do_page_fault 压栈顺序（见图 34），可以得出 regs 为 rbp（0xffff94e11e303d38），error_code 为 r12（0x0）。

6.2. 案例 2(arm64)

【UBXC006412】【国寿资产服务器操作系统采购】【一级】国寿资产生产服务器异常宕机

背景：此前已经有一例 uos_lazy_dup 导致__get_file 中死循环踩了其他内存，导致内核崩溃。9-24 日前场反馈又有一台机器发生崩溃，判断是否为同一问题。

6.2.1. 崩溃处分析

发生崩溃时正在执行 iterate_dir+40（见图 36），查看这部分的汇编代码（见图 37）。iterate_dir+20 将 x0 赋给 x19，按照 arm64 的寄存器约定，x0 为第一个参数 file（见图 38）

```
crash> bt
PID: 217214   TASK: ffffa04095132b40  CPU: 41   COMMAND: "easymonitor_mai"
 #0 [ffffa040946a77c0] machine_kexec at ffff3367e80fabe4
 #1 [ffffa040946a7960] __crash_kexec at ffff3367e8221bf4
 #2 [ffffa040946a79a0] crash_kexec at ffff3367e8221d28
 #3 [ffffa040946a79d0] die at ffff3367e80e224c
 #4 [ffffa040946a7a10] die_kernel_fault at ffff3367e810a8d8
 #5 [ffffa040946a7a40] __do_kernel_fault at ffff3367e810a5b8
 #6 [ffffa040946a7b10] do_page_fault at ffff3367e8d84ef4
 #7 [ffffa040946a7b70] do_translation_fault at ffff3367e8d8541c
 #8 [ffffa040946a7c30] do_mem_abort at ffff3367e80d12e0
 #9 [ffffa040946a7da0] el1_ia at ffff3367e80d318c
     PC: ffff3367e83dd840  [iterate_dir+40]
     LR: ffff3367e83de6b0  [ksys_getdents64+168]
     SP: ffffa040946a7db0  PSTATE: 80400009
    X29: ffffa040946a7db0  X28: ffffa04095132b40  X27: 0000000000000000
    X26: 0000000000000000  X25: 0000000056000000  X24: ffffa040edcf8200
    X23: ffffa040edcf8201  X22: ffffa040946a7e00  X21: 0000002700000026
    X20: ffff3367e9389000  X19: ffffa040edcf8200  X18: 0000000000000000
    X17: 0000000000000000  X16: 0000000000000000  X15: 0000000000000000
    X14: 0000000000000000  X13: 0000000000000000  X12: 0000000000000000
    X11: 0000000000000000  X10: 0000000000000000   X9: 0000000000000000
     X8: 0000000000000000   X7: 0000000000000000   X6: 0000000000000000
     X5: 0000000000000000   X4: ffffa040edcf82f8   X3: 0000000000000000
     X2: 0000002700000027   X1: ffffa040946a7e00   X0: 0000002700000026
#10 [ffffa040946a7db0] iterate_dir at ffff3367e83dd83c
```

图 36

```
crash> dis -rl iterate_dir+40
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.81.uel20.aarch64/linux-4.19.90-2305.1.0.0199.81.uel20.aarch64/
fs/readdir.c: 41
0xffff3367e83dd818 <iterate_dir>:       stp     x29, x30, [sp, #-80]!
0xffff3367e83dd81c <iterate_dir+4>:     mov     x29, sp
0xffff3367e83dd820 <iterate_dir+8>:     stp     x19, x20, [sp, #16]
0xffff3367e83dd824 <iterate_dir+12>:    stp     x21, x22, [sp, #32]
0xffff3367e83dd828 <iterate_dir+16>:    stp     x23, x24, [sp, #48]
0xffff3367e83dd82c <iterate_dir+20>:    mov     x19, x0
0xffff3367e83dd830 <iterate_dir+24>:    mov     x22, x1
0xffff3367e83dd834 <iterate_dir+28>:    mov     x0, x30
0xffff3367e83dd838 <iterate_dir+32>:    nop
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.81.uel20.aarch64/linux-4.19.90-2305.1.0.0199.81.uel20.aarch64/
fs/readdir.c: 45
0xffff3367e83dd83c <iterate_dir+36>:    ldp     x21, x0, [x19, #32]
0xffff3367e83dd840 <iterate_dir+40>:    ldr     x1, [x0, #56]
```

图 37

```
40 int iterate_dir(struct file *file, struct dir_context *ctx)
41 {
42         struct inode *inode = file_inode(file);
43         bool shared = false;
44         int res = -ENOTDIR;
45         if (file->f_op->iterate_shared)
```

图 38

iterate_dir+36 将内存中 x19+32 地址处连续两个 8 字节内容依次赋值给 x21,x0。

结合 c 代码（图 38）及 file 结构体（图 39）可以判断，x21 中为 file->f_inode，x0 中为 file->f_op。

同理，结合图 37、图 38、图 40，iterate_dir+40 中的 x1 中为 file->f_op->iterate_shared。

```
crash> *file -o
struct file {
        union {
            struct llist_node fu_llist;
            struct callback_head fu_rcuhead;
     [0] } f_u;
    [16] struct path f_path;
    [32] struct inode *f_inode;
    [40] const struct file_operations *f_op;
    [48] spinlock_t f_lock;
```

图 39

```
crash> file_operations.iterate_shared
struct file_operations {
   [56] int (*iterate_shared)(struct file *, struct dir_context *);
}
```

图 40

```
crash> x /10xg 0xffffa040edcf8200
0xffffa040edcf8200:     0x0000002700000026      0x0000002700000026
0xffffa040edcf8210:     0x0000002700000026      0x0000002700000026
0xffffa040edcf8220:     0x0000002700000026      0x0000002700000026
0xffffa040edcf8230:     0x0000002700000026      0x0000002700000026
0xffffa040edcf8240:     0x0000002700000026      0x0000002700000026
```

图 41

首先排查下 file 结构体（根据上述分析，其值在 x19 中为 0xffffa040edcf8200）内容是否还正常（见图 41），可以发现这部分内存已经全被写成 0x0000002700000026。

```
crash> *file.f_op 0xffffa040edcf8200
  f_op = 0x2700000026,
```

图 42

file->f_op（x0）也不例外被写成了 0x0000002700000026（见图 42）。继续访问 file->f_op->iterate_shared（对应地址为 0x270000005e，计算见图 43）时发生页请求失败导致内核终止（见图 44）。

```
crash> px 0x0000002700000026+56
$2 = 0x270000005e
```

图 43

```
[749114.159174] Unable to handle kernel paging request at virtual address 000000270000005e
```

图 44

6.2.2. 踩内存处分析

根据上面的分析可以初步判定发生了踩内存，再看下其他 cpu 的栈上是否有__get_file 相关调用，判定是否为同一问题。

```
PID: 1023562  TASK: ffff80419b364ec0  CPU: 61   COMMAND: "StreamTrans #1"
 #0 [ffffa07fbfe8f870] crash_save_cpu at ffff3367e8222090
 #1 [ffffa07fbfe8f890] cpu_psci_cpu_die at ffff3367e80e36d4
 #2 [ffffa07fbfe8f8a0] handle_IPI at ffff3367e80eb244
 #3 [ffffa07fbfe8f910] gic_handle_irq at ffff3367e80d1808
--- <IRQ stack> ---
 #4 [ffffa040f026bd00] el1_irq at ffff3367e80d3434
    PC: ffff3367e8692d70  [__get_file+312]
    LR: ffff3367e8692d28  [__get_file+240]
    SP: ffffa040f026bd10  PSTATE: a0400009
   X29: ffffa040f026bd10  X28: ffffa04095644780  X27: 0000000000000026
   X26: ffffa040956447a8  X25: 0000000000000001  X24: 0000000000004000
   X23: 0000000000000027  X22: ffffa040956447a8  X21: 0000000000000027
   X20: 0000000000000026  X19: 000000000000004f  X18: 0000000000000000
   X17: 0000000000000000  X16: 0000000000000000  X15: 0000000000000000
   X14: 0000000000000000  X13: 0000000000000000  X12: 0000000000000000
   X11: 0000000000000000  X10: 000000000000003d   X9: 0000000000000000
    X8: 0000000000000000   X7: 000000007fff0000   X6: 0000000000000001
    X5: 000000000019e080   X4: 0000000000000026   X3: ffffa04095644830
    X2: 0000000000000026   X1: 000000000019e080   X0: ffffa040ed873800
 #5 [ffffa040f026bd10] __get_file at ffff3367e8692d6c
 #6 [ffffa040f026bd50] uld_get_file at ffff3367e8693414
```

执行 bt -a，检索其他 cpu 上的栈发现 61 号 cpu 栈上有相关调用，且寄存器中有 0x26，0x27，基本可以判定是此处发生了踩内存，下面来确认一下。

简单介绍下这部分代码（见图 45），201 行将 file 右移 1 位得到 fd，202 行根据 fdtable 获取 fd 对应的 file，如果 file 不存在（203 行）或 file 最后一位为 0（209 行）就退出循环，否则将 fd 赋给 fd_link[i++]继续循环。

各个参数的解析方法同案例 1 所示，只是寄存器约定不同，在此不再赘述。解析可得 fdt 在 x22（0xffffa040956447a8）中，file 在 x19（0x4f）中，fd 在 x20（0x26）中，fd_link 在 x0（0xffffa040ed873800）中，i 在 x1（0x19e080）中。

```
200         while (1) {
201                 fd = get_special_file_fd((unsigned long)file);
202                 file = rcu_dereference_raw(fdt->fd[fd]);
203                 if (!file) {
204                         if (DEBUG) pr_notice("__get_file2: slowpath: err: badf, pid=  , fd=  , file=
    ",
205                                         current->pid, fd, (unsigned long)file);
206                         break;
207                 }
208
209                 if (!uld_special_file(file))
210                         break;
211
212                 if (DEBUG) pr_notice("__get_file2: slowpath: special too, pid=  , fd=  , file=      ",
213                                 current->pid, fd, (unsigned long)file);
214                 fd_link[i++] = fd;
215         }
```

图 45

```
crash> fdtable.fd 0xffffa040956447a8
  fd = 0xffffa04095644830,
crash> fdtable.fd -o
struct fdtable {
   [8] struct file **fd;
}
```

图 46

根据 fdtable 获取 fd 数组（见图 46），fd 指针数组，一个元素 8 字节。当前 fd 为 0x26，得到 file 为 0x4f（见图 47），file 存在且最后一位不为 0，继续循环。

```
crash> px 0xffffa04095644830+0x26*8
$7 = 0xffffa04095644960
crash> x /xg 0xffffa04095644960
0xffffa04095644960:     0x000000000000004f
```

图 47

file 为 0x4f，右移一位得到 fd-0x27，对应 file 为 0x4d（见图 48），file 存在且最后一位不为 0，继续循环。file 为 0x4d,右移一位为 0x26，由此 fd 一直在 0x26,x027 死循环，并不断从 x0（0xffffa040ed873800）处交替写入 0x26,0x27。

```
crash> px (0x4f>>1)
$4 = 0x27
crash> px 0xffffa04095644830+0x27*8
$5 = 0xffffa04095644968
crash> x /xg 0xffffa04095644968
0xffffa04095644968:     0x000000000000004d
crash> px (0x4d>>1)
$6 = 0x26
```

图 48

7. 参考

[1] https://koji.uniontech.com/ koji 网址

[2] http://www.biscuitos.cn/blog/CRASH/#E00 内核核心转储：Kdump with kexec and crash

[3] https://www.intel.cn/content/www/cn/zh/content-details/782158/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html intel 开发者手册

[4] https://developer.arm.com/documentation/ddi0602/2024-06/Base-Instructions/ arm 开发者手册