# **Linux内核crash分析工具--Crash**

**会议录屏：**

主题: 内核crash分析分享--校云

日期: 2023-02-08 18:47:49

录制文件：<https://meeting.tencent.com/v2/cloud-record/share?id=dc898694-a6b5-4832-a9e0-6d34a09b6456&from=3>

访问密码：sDAG

# **一 Crash调试环境准备**

分析crash问题，需要安装crash工具以及对应内核的vmlinux。CPE目前只有x86平台和LW2x08支持crash时生成vmcore文件。其中， x86平台需要自己安装crash工具和vmlimux， LW2x08则需要使用我们自己提供的crash工具和vmlimux。

另外，以下所有操作都是在CentOS7上进行的。

## **1.1 x86调试环境**

### **1.1.1 安装crash工具**

直接使用yum安装crash即可：

yum install crash

### **1.1.2 安装kernel-debuginfo**

vmlinux位于kernel-debuginfo rpm包中，安装kernel-debuginfo后，vmlinux位于以下路径：

/usr/lib/debug/usr/lib/modules/3.10.0-514.26.2.el7.x86\_64/vmlinux

注意： 应该安装的是kernel-debuginfo， 不要安装成为了kernel-debug-debuginfo。

kernel-debuginfo rpm可以从下面的地址下载， 先安装kernel-debuginfo-common，再安装kernel-debuginfo：

http://debuginfo.centos.org/7/x86\_64/kernel-debuginfo-common-x86\_64-3.10.0-514.26.2.el7.x86\_64.rpm  
http://debuginfo.centos.org/7/x86\_64/kernel-debuginfo-3.10.0-514.26.2.el7.x86\_64.rpm

也可以从QA以下路径下载vmlinux， QA上包含了常见的、会使用到的vmlinux：

qa:/xyfeng/crash/x86\_crash\_tool/

/xyfeng/crash/x86\_crash\_tool> ls

-rw-r--r-- 1 508 533 460036960 Jun 01 20:16 vmlinux\_3.10.0-1160.2.1.el7.x86\_64

-rw-r--r-- 1 508 533 149939277 Jun 01 20:15 vmlinux\_3.10.0-229.el7.x86\_64

-rw-r--r-- 1 508 533 166231969 Jun 01 20:15 vmlinux\_3.10.0-514.21.2.el7.x86\_64

-rw-r--r-- 1 508 533 166244967 Jun 01 19:55 vmlinux\_3.10.0-514.26.1.el7.x86\_64

-rw-r--r-- 1 508 533 166256256 Jun 01 19:55 vmlinux\_3.10.0-514.26.2.el7.x86\_64

-rw-r--r-- 1 508 533 166137600 Jun 01 20:13 vmlinux\_3.10.0-514.el7.x86\_64

## **1.2 LW2x08调试环境**

LW2x08 crash工具是通过交叉编译而来的， vmlinux则是编译内核时保存下来对应的vmlinux。

crash工具和vmlinux直接从QA进行下载即可：

qa:/xyfeng/crash/lw2x08\_crash\_tool

/xyfeng/crash/lw2x08\_crash\_tool> ls

-rw-r--r-- 1 508 533 36726888 Jun 01 20:20 crash

-rw-r--r-- 1 508 533 308854080 Jun 01 20:21 vmlinux\_20210514\_4.19.26-00778-ge3118a779736

-rw-r--r-- 1 508 533 309036192 Jun 01 20:22 vmlinux\_4.19.26-00786-g3b3da0826858

-rw-r--r-- 1 508 533 310168168 Jun 01 20:23 vmlinux\_4.19.26-00788-gbd95801334a2

## **1.3 crash后需要提供哪些信息**

vmcore

vmcore-dmesg

lwconn.ko

CPE版本号

内核版本号

## **1.4 如何确定vmlinux/vmcore/lwconn是否匹配**

有时会遇到根据提供的信息选择vmlinux时，发现vmlinux和vmcore不匹配，导致无法分析。

我们可以使用strings命令过滤以下关键字，从vmcore、vmlinux、lwconn.ko中确认版本是否匹配：

1. 从vmlinux中确认内核版本号：

$strings vmlinux | grep 'Linux ver'

Linux version 3.10.0-514.26.2.el7.x86\_64 (builder@kbuilder.dev.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-11) (GCC) ) #1 SMP Tue Jul 4 15:04:05 UTC 2017

2. 从vmcore中确认内核版本号和DP版本号：

$strings vmcore|grep -E 'OSRELEASE|LW LWC'

OSRELEASE=3.10.0-514.26.2.el7.x86\_64

OSRELEASE=p

OSRELEASE=3.10.0-514.26.2.el7.x86\_64

OSRELEASE=3.10.0-514.26.2.el7.x86\_64

LW LWCONN version: 5.5.0.433, git: Branch\_v5|

3. 从lwconn.ko中确认内核版本号和DP版本号：

$modinfo lwconn.ko |grep vermagic

vermagic: 4.19.26-00788-gbd95801334a2 SMP preempt mod\_unload modversions aarch64

$strings lwconn.ko |grep -E 'vermagic=|LW LW'

LW LWCONN version: 5.0.0.316, git: Branch\_v5.0.0:8a2390b2, build: Jun 13 2022 21:51:15

vermagic=4.19.26-00788-gbd95801334a2 SMP preempt mod\_unload modversions aarch64

# **二 Crash工具使用**

## **2.1 运行crash工具**

crash [OPTION]... NAMELIST MEMORY-IMAGE

OPTION:

-i file crash启动后，执行file中指定的命令。可用于自动导出数据。

NAMELIST： 未压缩的内核镜像文件， 即：vmlinux

MEMORY-IMAGE：内存存储文件，即vmcore

启动crash工具时，vmlinux和vmcore可以不区分先后顺序，即以下2种方式都可以启动crash工具：

crash vmlinux vmcore

crash vmcore vmlinux

crash工具启动后，会显示crash基本信息，包含：crash时间、系统运行时间、内核版本、crash位置以及crash时进程信息。

示例如下：

$crash ~/crash/vmlinux/vmlinux\_3.10.0-514.26.2.el7.x86\_64 ./vmcore

crash 7.2.3-11.el7\_9.1

...

GNU gdb (GDB) 7.6

...

This GDB was configured as "x86\_64-unknown-linux-gnu"...

KERNEL: /home/xyfeng/crash/vmlinux/vmlinux\_3.10.0-514.26.2.el7.x86\_64

DUMPFILE: ./vmcore [PARTIAL DUMP]

CPUS: 2

DATE: Sat Sep 24 11:46:19 2022

UPTIME: 330 days, 18:34:01

LOAD AVERAGE: 0.04, 0.05, 0.10

TASKS: 127

NODENAME: cpe

RELEASE: 3.10.0-514.26.2.el7.x86\_64

VERSION: #1 SMP Tue Jul 4 15:04:05 UTC 2017

MACHINE: x86\_64 (3700 Mhz)

MEMORY: 7.9 GB

PANIC: "BUG: unable to handle kernel NULL pointer dereference at 0000000000000080"

PID: 14925

COMMAND: "sh"

TASK: ffff8802627e5e20 [THREAD\_INFO: ffff880238664000]

CPU: 0

STATE: TASK\_RUNNING (PANIC)

crash>

## **2.2 退出crash工具**

输入q 或者exit退出crash工具。示例如下：

crash> q

[xyfeng 127.0.0.1-2022-09-24-11\_46\_33]

# **三 常用调试命令**

在crash交互界面中，支持对crash命令执行的结果通过管道符"|"进行进一步处理，或者重定向到文件中。管道符后可以使用所在linux主机系统支持的所有命令，比如grep、awk等。

## **3.1 help**

crash> help

\* extend log rd task

alias files mach repeat timer

ascii foreach mod runq tree

bpf fuser mount search union

bt gdb net set vm

btop help p sig vtop

dev ipcs ps struct waitq

dis irq pte swap whatis

eval kmem ptob sym wr

exit list ptov sys q

...

crash> help bt

NAME

bt - backtrace

SYNOPSIS

bt [-a|-c cpu(s)|-g|-r|-t|-T|-l|-e|-E|-f|-F|-o|-O|-v] [-R ref] [-s [-x|d]]

[-I ip] [-S sp] [pid | task]

DESCRIPTION

Display a kernel stack backtrace. If no arguments are given, the stack

trace of the current context will be displayed.

## **3.2 sys 显示crash基本信息**

直接输入sys会显示crash基本信息，即2.1 crash启动后展示的信息。除此之前，还可以使用sys config查看内核config配置， 只适用于LW2x08。

比如：

crash> sys config |grep NET\_NS

CONFIG\_NET\_NS=y

CONFIG\_NET\_NSH=m

crash>

## **3.3 bt 查看内核堆栈**

bt 显示crash时内核堆栈

常用选项：

-a 显示所有核上的堆栈

-c CpuId 显示指定核上的堆栈

-f 显示每一帧的栈数据， 必要时需要通过栈数据获取信息

-s 显示函数偏移

-l 显示函数所在的文件和行数

crash>

PID: 14925 TASK: ffff8802627e5e20 CPU: 0 COMMAND: "sh"

#0 [ffff8802386676f8] machine\_kexec+411 at ffffffff81059beb

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/arch/x86/kernel/machine\_kexec\_64.c: 319

ffff880238667700: 0000fbbf00100800 ffff880000000000

ffff880238667710: 000000002b001000 ffff88002b001000

ffff880238667720: 000000002b000000 ffff8802386679a8

ffff880238667730: 00000000a5c3b038 ffff8802386679a8

ffff880238667740: ffff880238667760 ffff8802386679a8

ffff880238667750: ffff880238667820 ffffffff81105822

...

#8 [ffff880238667970] do\_page\_fault+53 at ffffffff81692ff5

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/arch/x86/mm/fault.c: 1237

ffff880238667978: 00000000a5c3b038 0000000000000001

ffff880238667988: 0000000000000080 00000000000000d0

ffff880238667998: ffff880238667a88 ffffffff8168f208

#9 [ffff8802386679a0] page\_fault+40 at ffffffff8168f208

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/arch/x86/kernel/entry\_64.S: 1316

[exception RIP: kmem\_cache\_alloc+117]

RIP: ffffffff811de875 RSP: ffff880238667a58 RFLAGS: 00010286

RAX: 0000000000000000 RBX: ffff88017f823e40 RCX: 000000000cb3f899

RDX: 000000000cb3f898 RSI: 00000000000000d0 RDI: ffff88017fc5a300

RBP: ffff880238667a88 R8: 000000000001a3a0 R9: ffffffff81217fc5

R10: fefefefefefefeff R11: 0000000000000000 R12: 0000000000000080

R13: 00000000000000d0 R14: ffff88017fc5a300 R15: ffff88017fc5a300

ORIG\_RAX: ffffffffffffffff CS: 0010 SS: 0018

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/mm/slub.c: 251

ffff8802386679a8: ffff88017fc5a300 ffff88017fc5a300

ffff8802386679b8: 00000000000000d0 0000000000000080

ffff8802386679c8: ffff880238667a88 ffff88017f823e40

ffff8802386679d8: 0000000000000000 fefefefefefefeff

ffff8802386679e8: ffffffff81217fc5 000000000001a3a0

ffff8802386679f8: 0000000000000000 000000000cb3f899

ffff880238667a08: 000000000cb3f898 00000000000000d0

ffff880238667a18: ffff88017fc5a300 ffffffffffffffff

ffff880238667a28: ffffffff811de875 0000000000000010

ffff880238667a38: 0000000000010286 ffff880238667a58

ffff880238667a48: 0000000000000018 ffffffff811de835

ffff880238667a58: ffffffff81217fc5 ffff88017f823e40

ffff880238667a68: ffff880238667d20 ffff880262e1b800

ffff880238667a78: ffff880238667b2f 0000000000000011

ffff880238667a88: ffff880238667ab0 ffffffff81217fc5

#10 [ffff880238667a90] \_\_d\_alloc+37 at ffffffff81217fc5

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/fs/dcache.c: 1369

ffff880238667a98: ffff88017f823e40 ffff88017f823e40

ffff880238667aa8: ffff880238667d20 ffff880238667ad8

ffff880238667ab8: ffffffff8121813a

#11 [ffff880238667ab8] d\_alloc+26 at ffffffff8121813a

/usr/src/debug/kernel-3.10.0-514.26.2.el7/linux-3.10.0-514.26.2.el7.x86\_64/fs/dcache.c: 1431

ffff880238667ac0: 0000000000000000 ffff88017f823e40

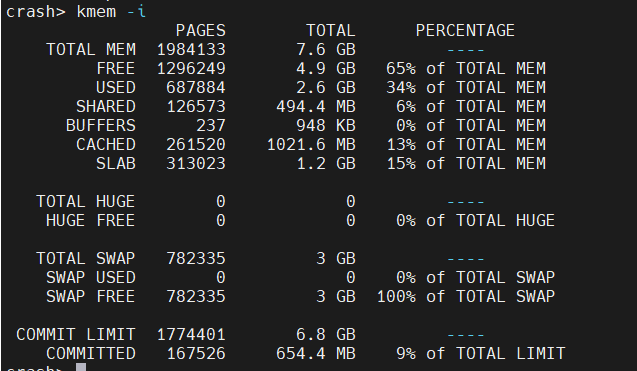
ffff880238667ad0: ffff880238667d20 ffff880238667b18

ffff880238667ae0: ffffffff812095bb

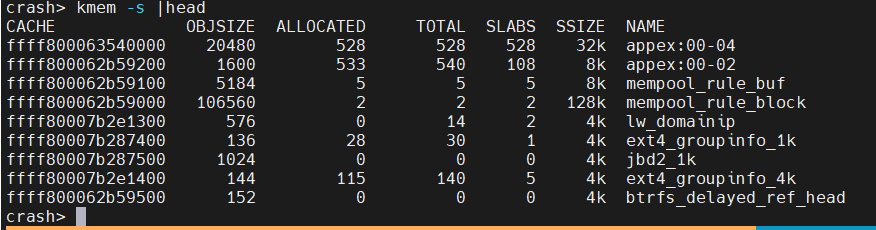
## **3.4 kmem 内存信息**

### **3.4.1 常用命令**

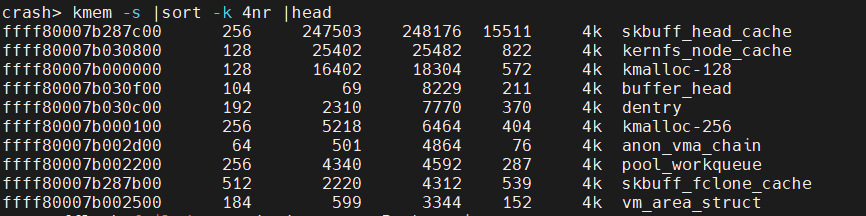
kmem -i 查看内存使用情况



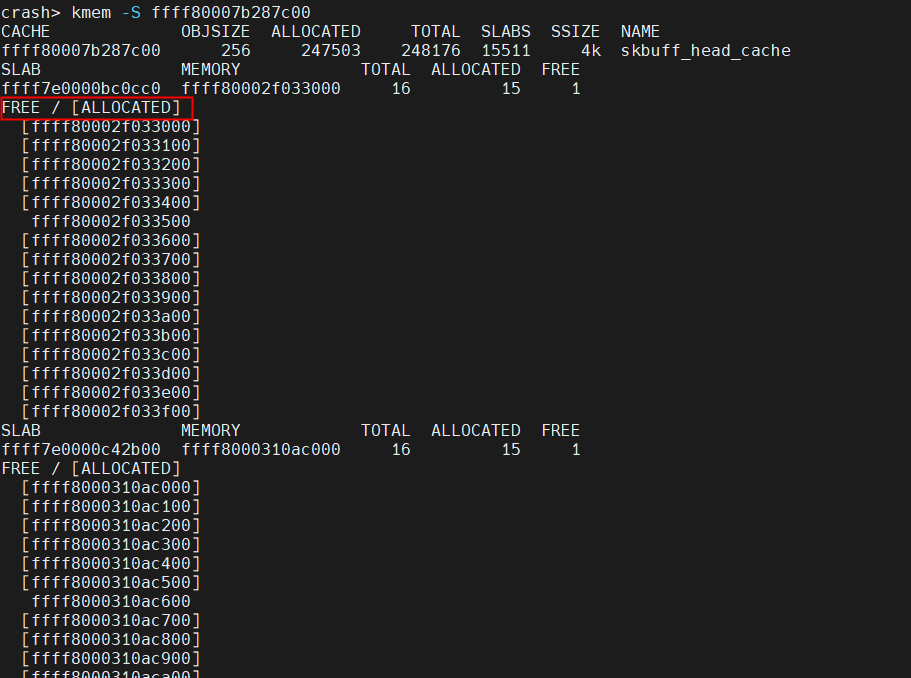
kmem -s 查看所有slab内存使用情况：

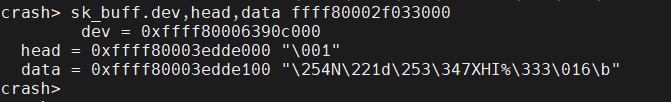


查看占用最多的slab内存：

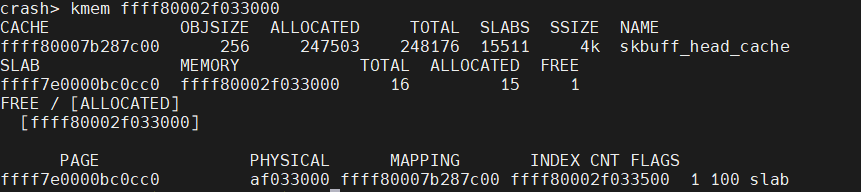


kmem -S 查看某个slab申请的所有内存， 其中， 带[] 的表示没有释放的内存， 不带[]的表示已释放的内存





kmem address 查看内核地址信息：



### **3.4.2 slab merge**

#### **slab分配器**

3种slab分配器： slab、slub、slob。

对于CPE， LW2x08内核使用的是slab， 其他内核使用的都是是slub。

#### **如何确认系统使用了哪种slab**

crash> sys config |grep -E '(SLAB|SLUB|SLOB)=y'

CONFIG\_SLAB=y

#### **slab\_nomerge/slub\_nomerge**

为了减少内存碎片及开销，内核默认会把object大小相近的memory cache合并到一个slab中，

所以从slabinfo中无法看到每个memory cache使用的内存，对于排查内存问题造成了干扰。

通过在grub.conf添加slab\_nomerge/slub\_nomerge可以禁止slab 合并，此配置重启系统后生效。

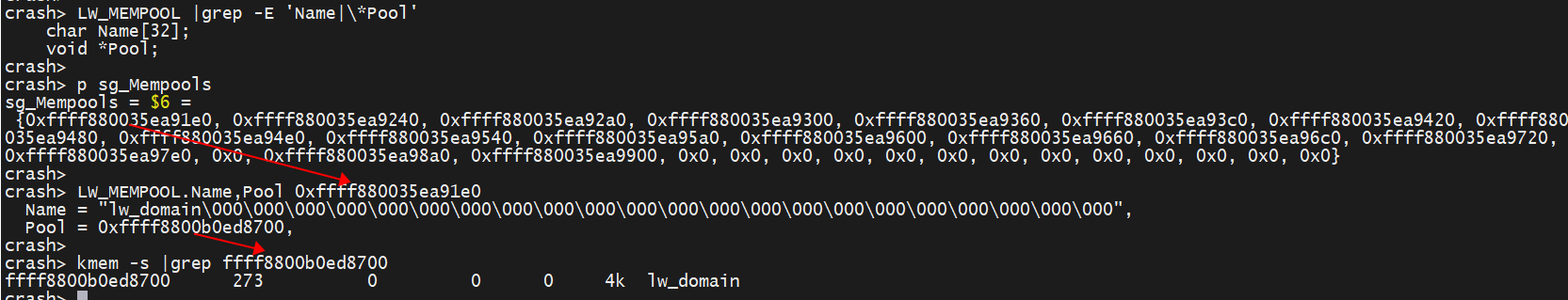
#### **slab合并后，如何确定DP/Engine使用了哪些slab？**

DP 从 psbc 全局变量 sg\_Mempools 中查看每个 mem\_cache 的 slab 地址。

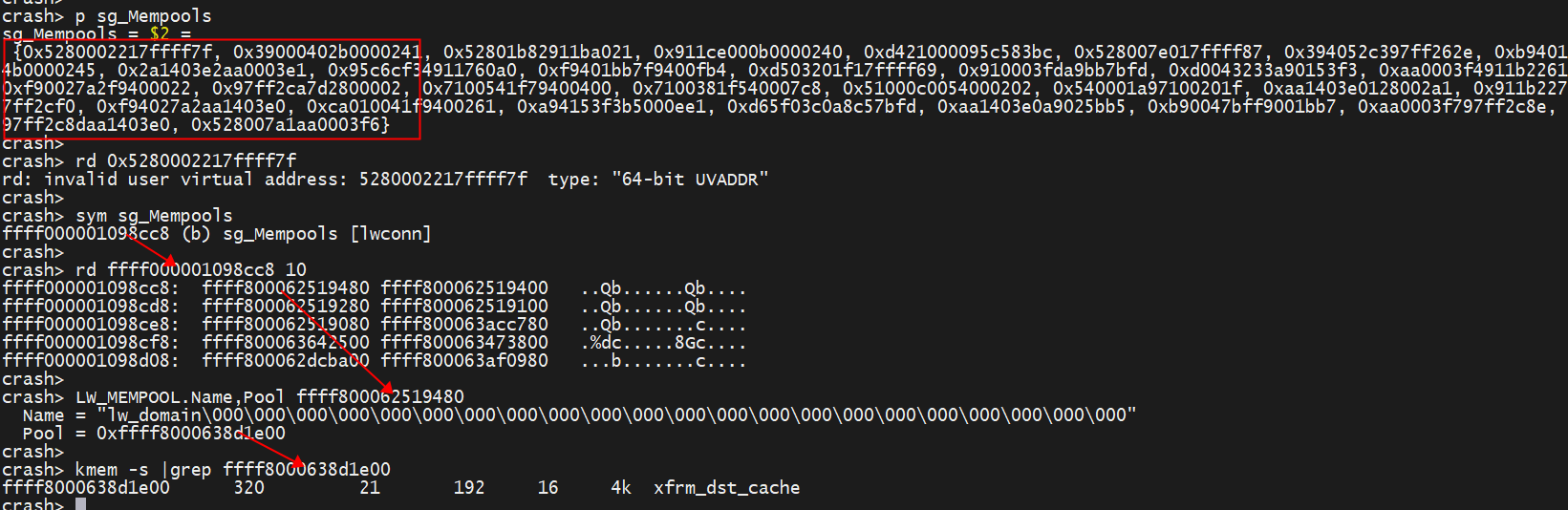
Engine 从 basecode 全局变量 gApxBaseSharedInfo 中获取Engine地址，然后查找Eninge使用了哪些 slab 地址。

示例如下：

1. sg\_Mempools可以直接访问时，直接通过sg\_Mempools找到对应的mem cache 地址，然后再从slab结果中查询：



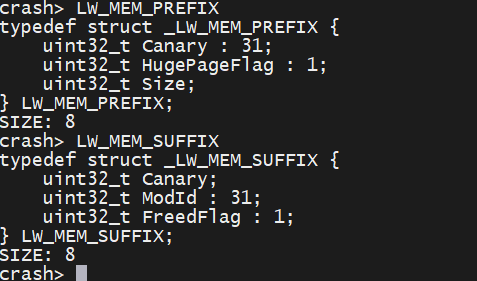
2）sg\_Mempools异常，不能直接访问时(2x08中经常遇到)，通过sym查找sg\_Mempools地址，然后按照1)中方式查找：



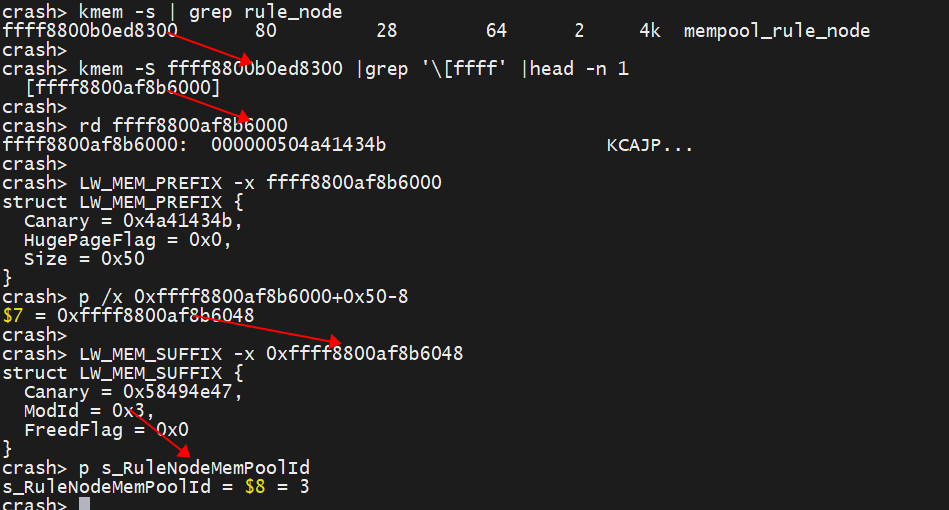
3）sg\_Mempools完全异常，2)中方法也不能找到其对应地址时, 尝试通过LW\_MEM\_SUFFIX找到其对应的Mem ModId，然后根据ModId确定是哪个slab。

从以下版本开始支持ModId：3.5.0.443、5.0.0.169、5.2.0.51

LW\_MEM\_PREFIX / LW\_MEM\_SUFFIX 的定义：



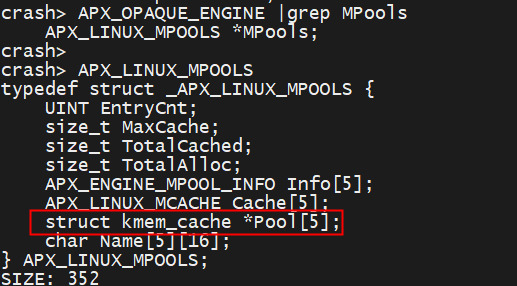
从memory中查找ModId：



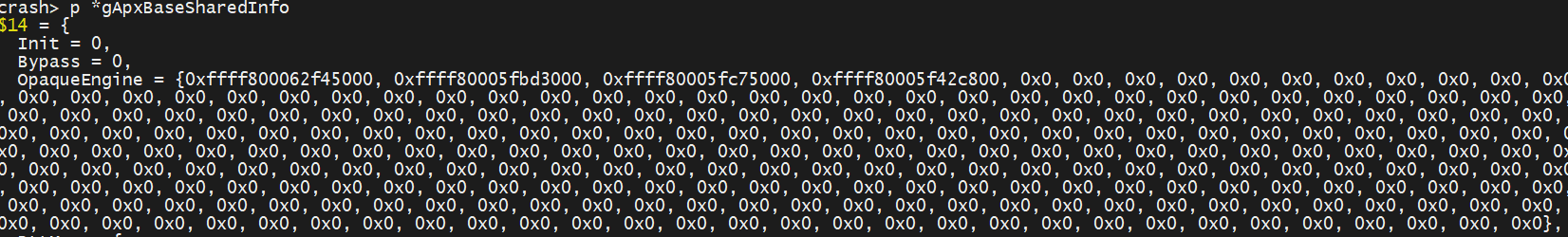
4）CPE版本LW\_MEM\_SUFFIX不支持ModId时，只能使用下面比较笨的方式进行确认：根据LW\_MEM\_PREFIX.Size 逐个排查DP中的mem pool，查找匹配的mem pool。存在多个大小相同的mem pool时，再根据内存内容判断具体对应哪个mem pool。

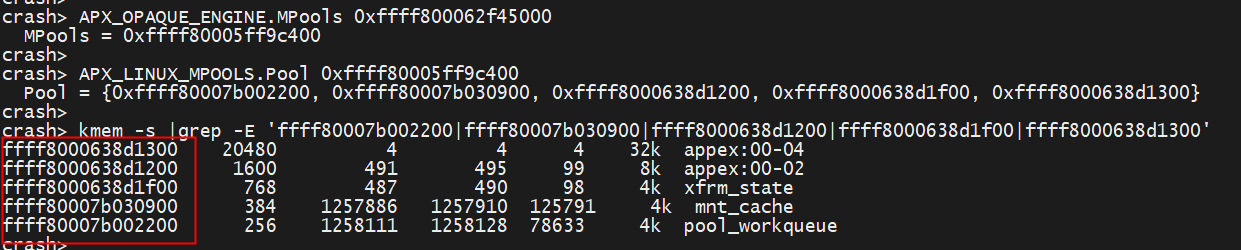
5) 查找Eninge使用了哪些 slab 地址

APX\_OPAQUE\_ENGINE 定义：



从 basecode 全局变量 gApxBaseSharedInfo 中获取Engine地址：





## **3.5 ps 查看进程状态**

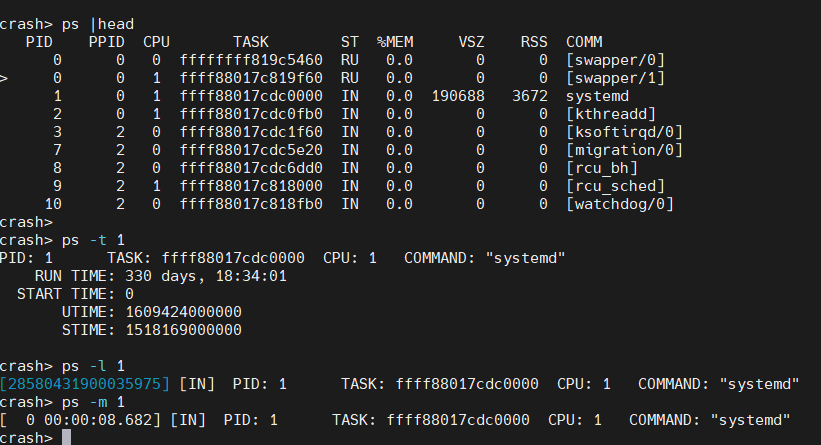
ps 显示进程状态

ps -c CpuId 显示指定核上的进程状态

ps -t 显示进程运行时间、起始时间、用户态和内核运行时间

ps -l 显示进程最后一次运行的时间戳

ps -m 把 ps -l 结果转换为"天-时-分-秒-毫秒"格式



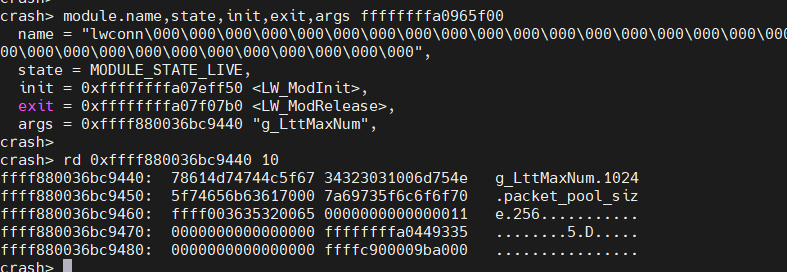
## **3.6 mod 模块信息**

mod 显示模块信息

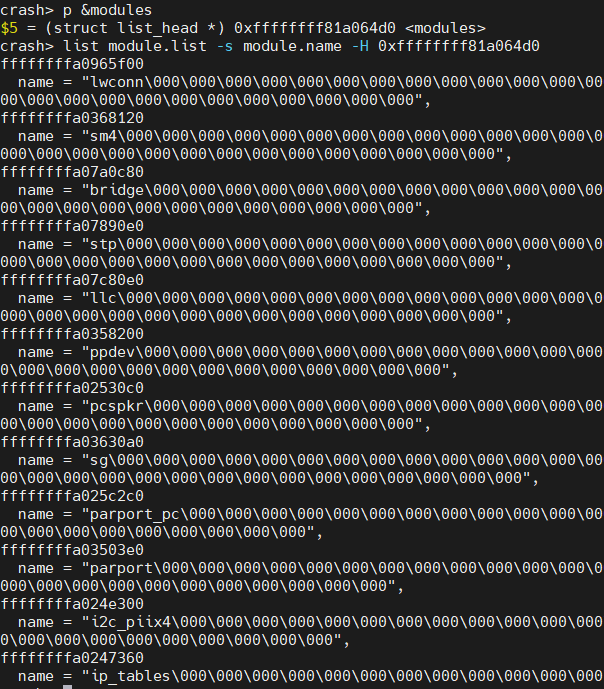
mod -s module-name module-objfile 加载模块符号表和debug数据



通过数据结构查看指定模块信息：



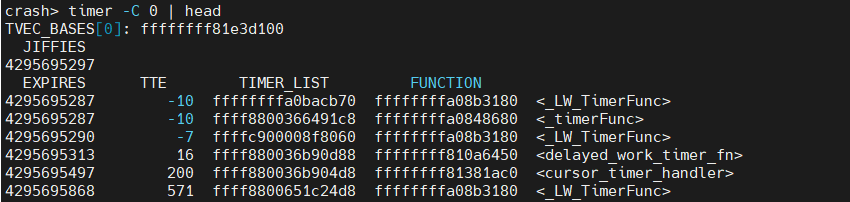
遍历所有模块：



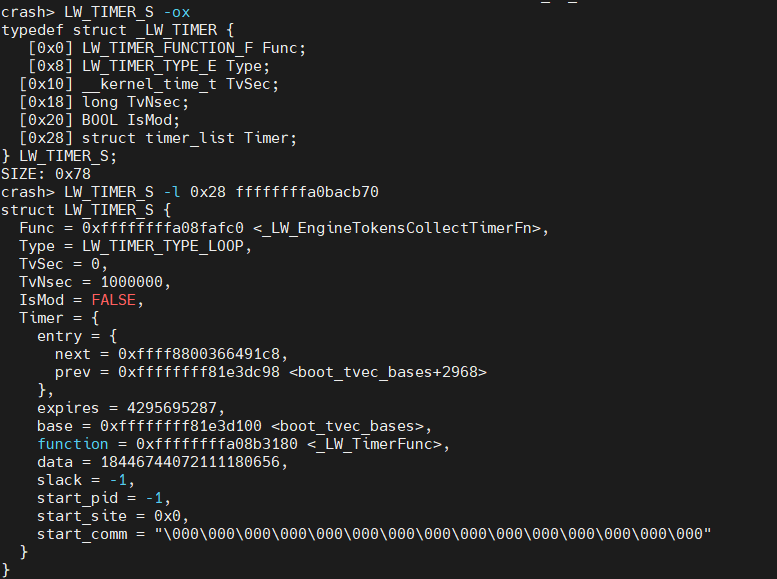
## **3.7 timer 定时器信息**

timer 查看所有timer

timer -C CpuId 查看指定核上的timer



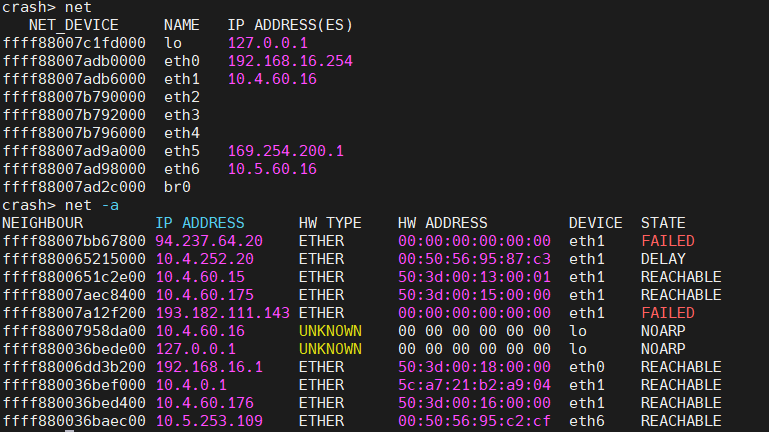
确定DP中具体哪个Timer：



## **3.8 net 接口信息**

net 显示接口信息

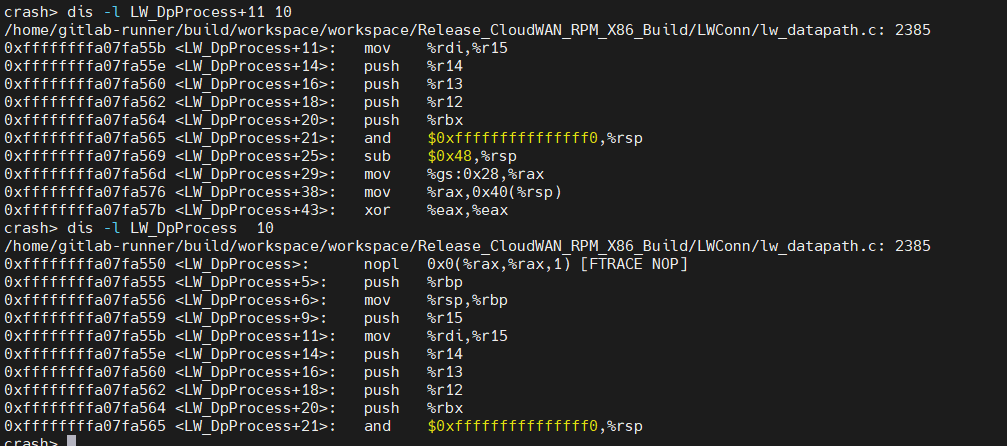
net -a 查看arp cache



## **3.9 dis 反汇编命令**

dis -l address [count] 从指定地址开始反汇编

dis -l function [count] 反汇编指定的函数



## **3.10 log**

log 显示crash log，结果同 vmcore-dmesg

log -H 时间以人性化方式显示

## **3.11 p 打印**

p 打印变量或者表达式的值

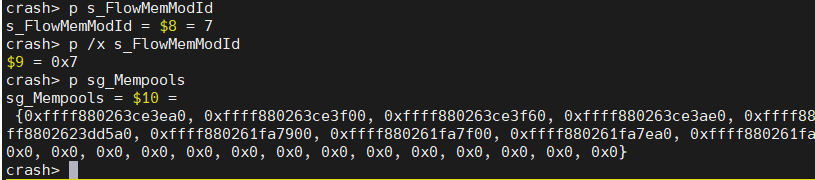
选项：

-x 以16进制打印

打印percpu变量：

:a 打印所有cpu上的值

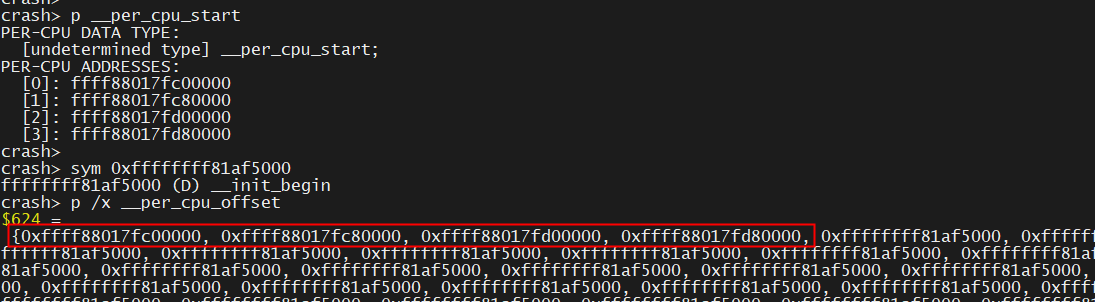
:cpuId[,-cpuId] 打印指定某些cpu上的值

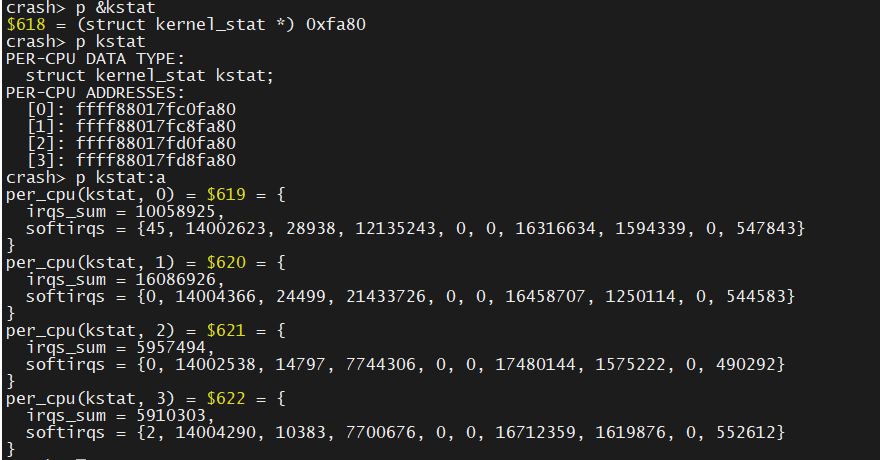


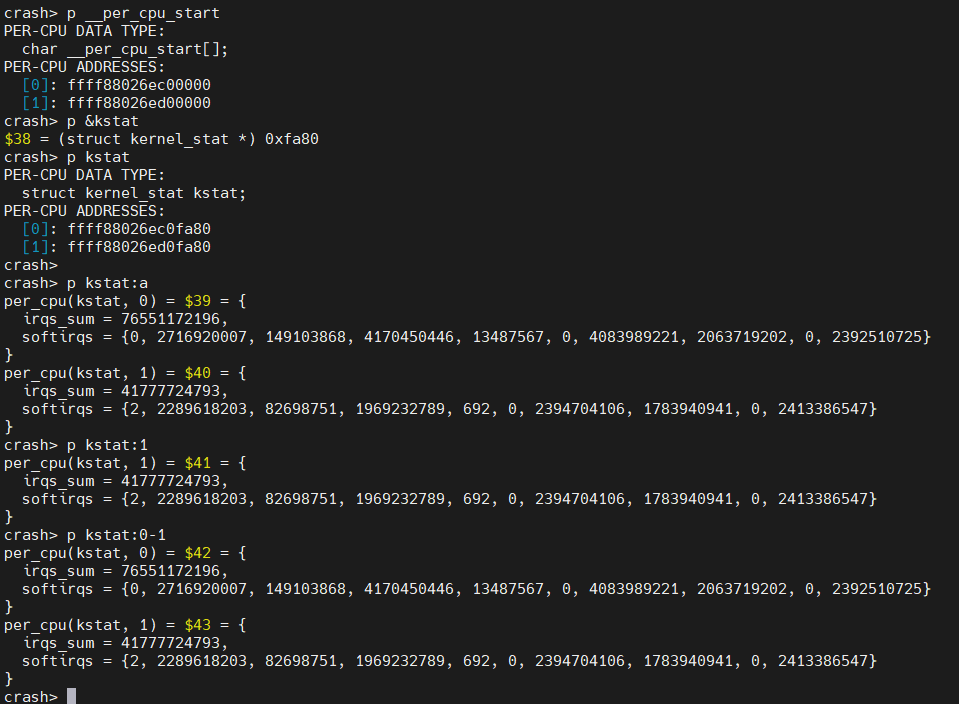
打印percpu变量：(内核percpu变量的实现)

percpu变量地址：\_\_per\_cpu\_offset[CpuId] + percpu指针地址

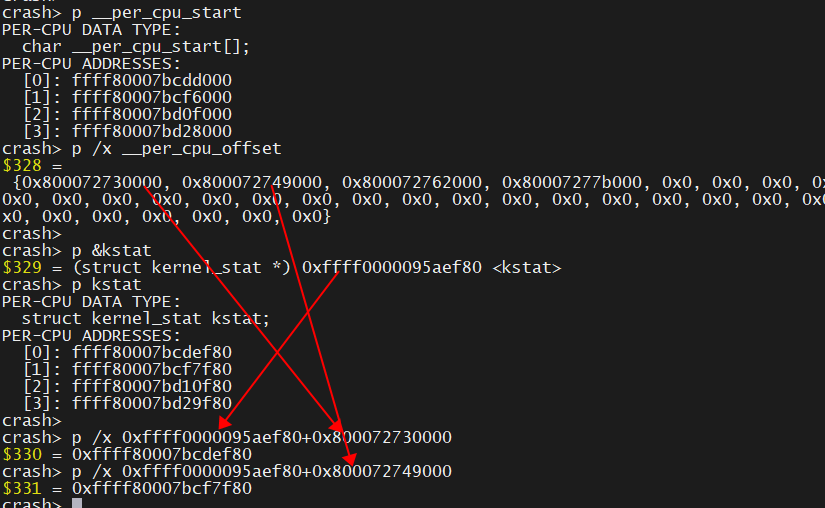
x86内核：



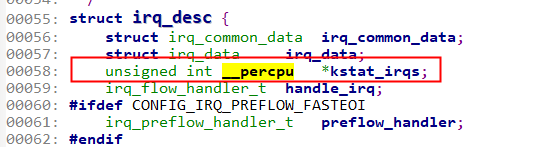


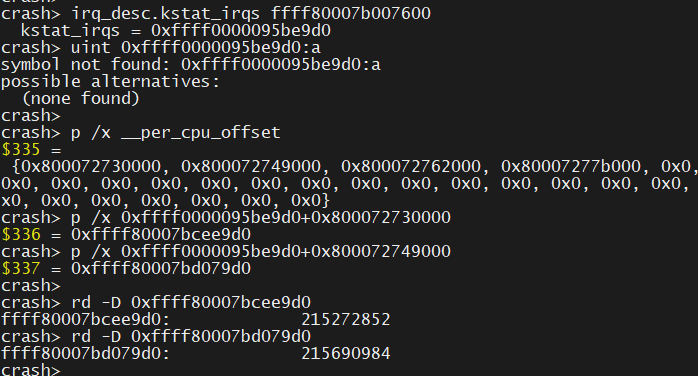


2x08内核：



非结构体类型的percpu变量如何访问：





## **3.12 rd 查看内存内容**

rd [options] addr/symbol [count]

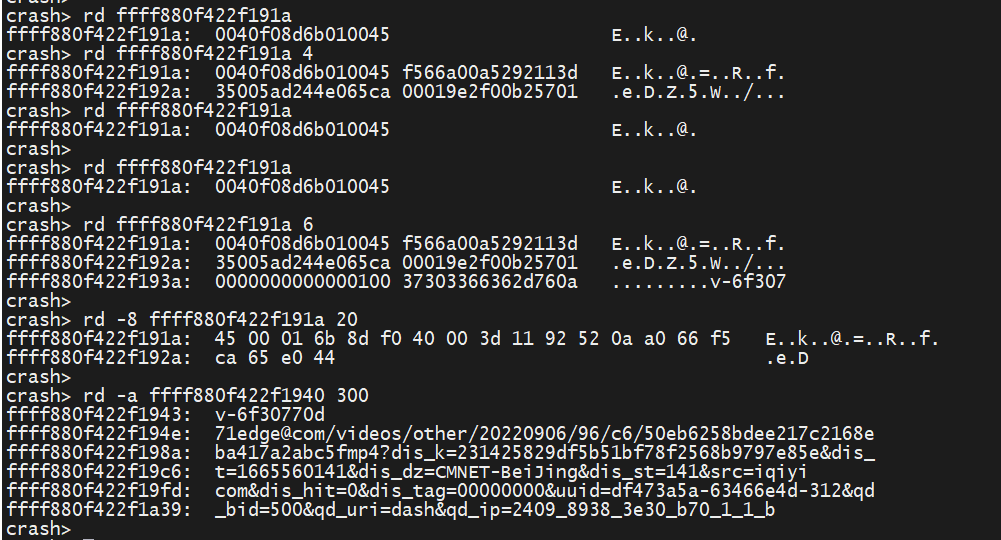
选项：

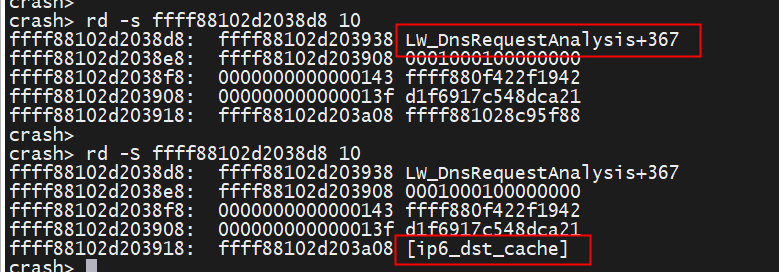
-8 每次访问内存大小的单位，默认为指针类型数据的大小。

-a 以字符串进行访问

-s 对地址进行符号解析

-S 显示内存对应slab名称





## **3.13 struct 数据结构**

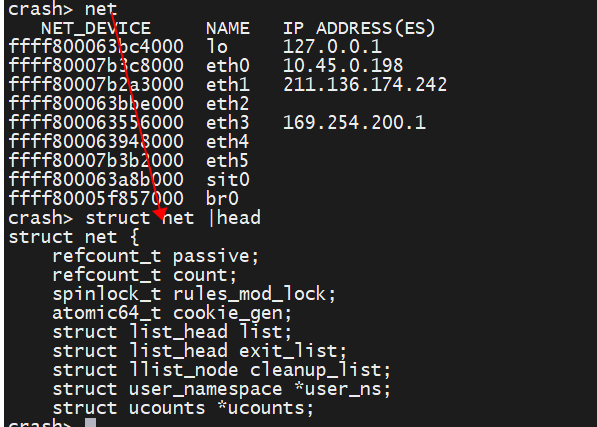
[struct] struct\_name 查看数据结构定义及其大小

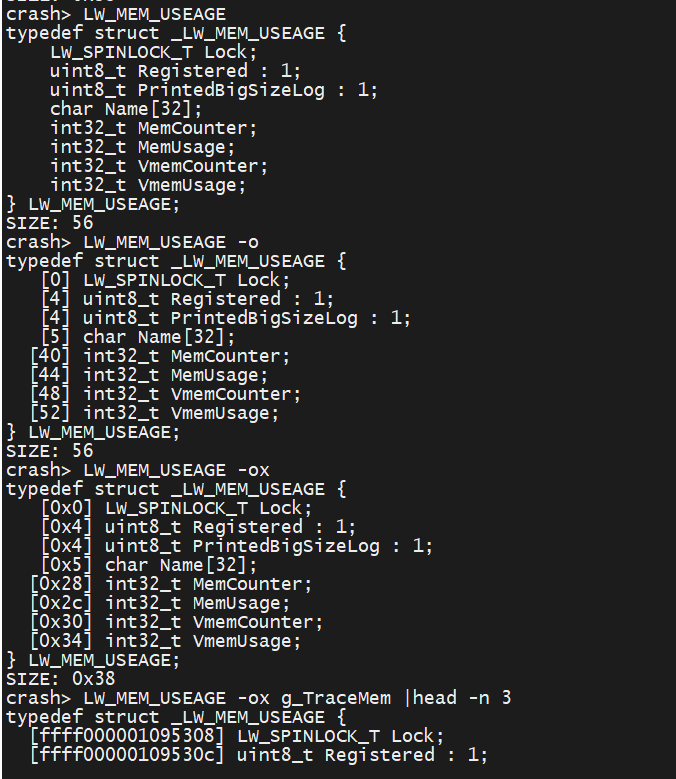
[struct] struct\_name -o 查看数据结构定义并显示成员offset

[struct] struct\_name -ox 以16进制显示成员offset

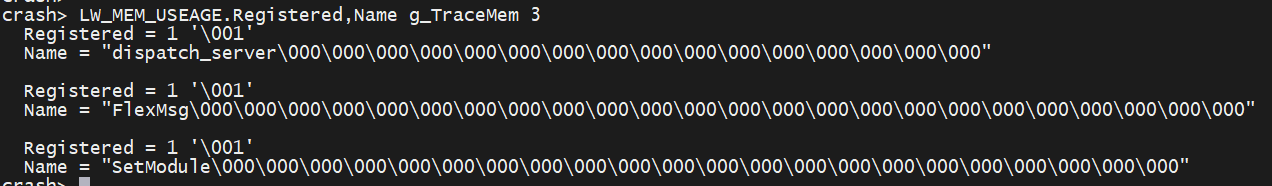
[struct] struct\_name -ox [symbol | address] 显示成员地址

其中，当strcut\_name和crash命令或者变量名称不存在冲突时， 可以省略struct命令，直接写struct name即可。 比如， struct net和crash net命令：



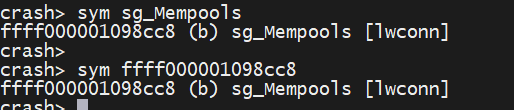


[struct] struct\_name[.member[,member]] [-x] [symbol | address] [count] 以结构体访问此片内存内容



## **3.14 sym 内核符号表**

sym symbol/address 查询内核符号和地址之间互查



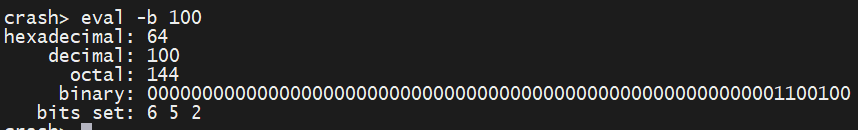
## **3.15 whatis**

whatis symbol/struct/union/typedef 查看变量或者数据结构的定义

## 

## **3.16 eval**

eval -b value | expression 以各种进制显示结果，包含bit位设置



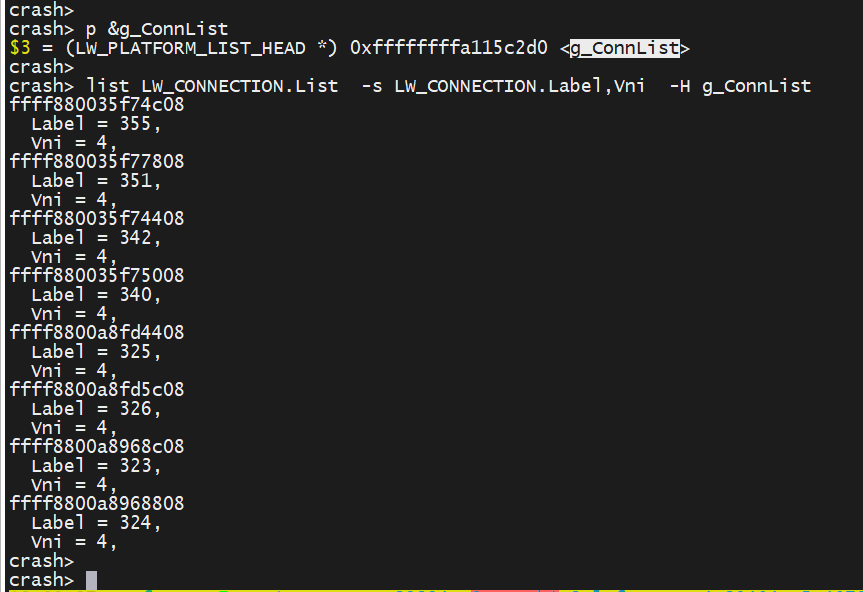
## **3.17 list 遍历链表**

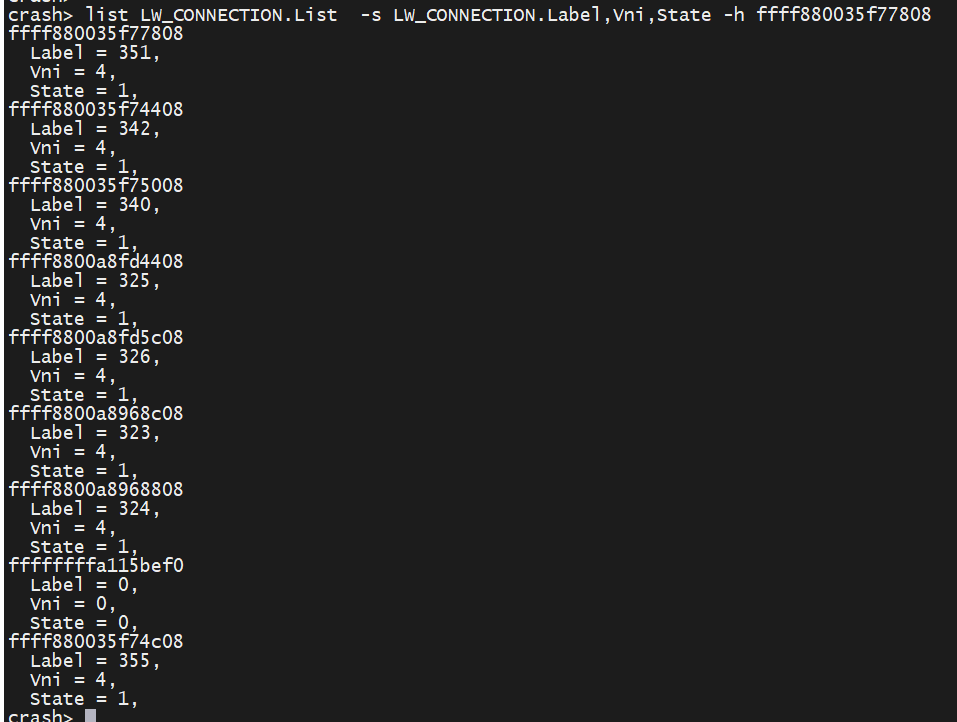
list offset -s struct[.member[,member] -H start

list offset -s struct[.member[,member] -h start

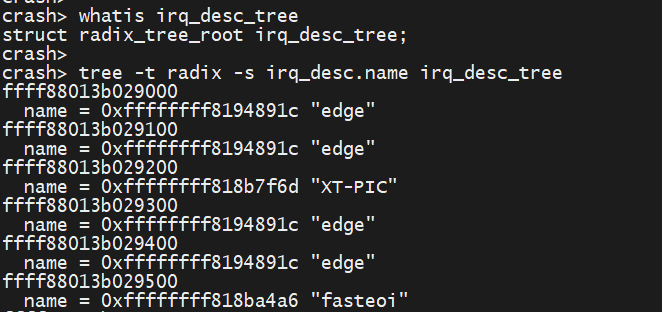
-H: The address of a list\_head structure.

-h: The address of a data structure which contains an embedded list\_head.





## **3.18 tree 遍历树**



# **四 常见crash类型实例分析**

比较常见的crash主要是和内存相关的，比如：内存耗尽、内存越界等。内存耗尽一般是由环路或者内存泄漏导致，而内存越界常见于报文处理过程中，比如：异常报文处理、字节序处理等。

以下分享的crash对应的vmcore文件都已上传到以下位置：qa:/xyfeng/crash/issue

## **4.1 内存耗尽**

### **4.1.1 #35226 [国电投-上海重然]运营商路由环路导致CPE crash**

[《Crash问题分析之#35226》](https://alidocs.dingtalk.com/i/nodes/3mzaPNMZ6jkJqy3vROjAJYLDwXq4Ky1r?doc_type=wiki_doc#)

#36399 [南京易投]客户内网调整误操作引起环路导致CPE crash

#39404 [国电投-经开能源]网卡驱动没有释放数据包导致内存耗尽而crash

## **4.2 内存越界访问**

#38692 FM150 驱动crash导致CPE内核crash

#39029 [国电投]北京数据中心集群一区CPE3 crash

#40623 [集群CPE健康及互备管理]CPE更新版本后，进行关机、开机操作，出现报错

## **4.3 栈溢出**

#15730 网关由3.0.0.391升级为3.1.0.82，CPE crash，调用栈cpuacct\_charge+0x2d/0x50

## **4.4 死锁**

#32733 [宝安信息中心]CPE Crash

# **五 常见数据导出**

数据导出思路：根据全局变量或者从堆栈、寄存器中分析出数据对应的地址，然后根据对应的数据结构进行访问。

## **5.1 系统数据导出**

查看软中断/proc/softirqs

源码：fs/proc/softirqs

实现： percpu变量kstat记录了每个softirq stat；

softirq\_to\_name数组记录每个softirq对应的名称。

crash> p kstat:a  
per\_cpu(kstat, 0) = $4 = {  
 irqs\_sum = 82979,  
 softirqs = {0, 45074719, 2, 188490, 38, 0, 44228924, 11080696, 0, 1205927}  
}  
per\_cpu(kstat, 1) = $5 = {  
 irqs\_sum = 163972,  
 softirqs = {2, 44965583, 2, 274182, 28615, 0, 37218044, 10255737, 0, 1151673}  
}  
crash> p softirq\_to\_name  
softirq\_to\_name = $6 =  
 {0xffffffff8193d6fc "HI", 0xffffffff818b82cd "TIMER", 0xffffffff818d3f67 "NET\_TX", 0xffffffff818d3f6e "NET\_RX", 0xffffffff81939d34 "BLOCK", 0xffffffff818d3fa5 "BLOCK\_IOPOLL", 0xffffffff818d3f7e "TASKLET", 0xffffffff818d3f86 "SCHED", 0xffffffff818d3f8c "HRTIMER", 0xffffffff818d3f94 "RCU"}  
   
 list mm\_struct.mmlist -s mm\_struct.brk -s mm\_struct.brk\_end 0xffff88012539ddc0  
  
  
task\_struct  
 --> mm(struct mm\_struct)  
 --> struct list\_head mmlist   
   
   
   
 ffff8800b9d796c0:  
list mm\_struct.mmlist -s mm\_struct.start\_brk -s mm\_struct.brk -h 0xffff88012539ddc0   
  
  
 135: new = kmem\_cache\_create(name, table\_size, align, 0, ctor);   
   
 280: tsb\_caches[i] = kmem\_cache\_create(name,   
  
rcu\_do\_batch  
\_\_rcu\_reclaim--> crash  
  
https://www.cnblogs.com/pengdonglin137/p/16064232.html  
tree -t radix 0xffffffff81a09a60 > irq\_desc\_tree.list  
  
查看/proc/interrupts：  
fs/prco/interrupts.c  
  
kernel/irq/irqdesc.c:  
crash> p &irq\_desc\_tree  
$99 = (struct radix\_tree\_root \*) 0xffffffff81a09a60 <irq\_desc\_tree>  
  
tree -t radix -s irq\_desc.name 0xffffffff81a09a60  
tree -t radix -s irq\_desc.action 0xffffffff81a09a60 |grep 'action = 0xff'  
irqaction.name 0xffffffff819d7600  
  
arch/x86/kernel/irq.c:  
  
#define irq\_stats(x)(&per\_cpu(irq\_stat, x))  
  
硬中断：  
 p irq\_stat:a |grep apic\_timer\_irqs  
  
  
软中断：  
crash> p kstat:a  
per\_cpu(kstat, 0) = $4 = {  
 irqs\_sum = 82979,  
 softirqs = {0, 45074719, 2, 188490, 38, 0, 44228924, 11080696, 0, 1205927}  
}  
per\_cpu(kstat, 1) = $5 = {  
 irqs\_sum = 163972,  
 softirqs = {2, 44965583, 2, 274182, 28615, 0, 37218044, 10255737, 0, 1151673}  
}  
crash> p softirq\_to\_name  
softirq\_to\_name = $6 =  
 {0xffffffff8193d6fc "HI", 0xffffffff818b82cd "TIMER", 0xffffffff818d3f67 "NET\_TX", 0xffffffff818d3f6e "NET\_RX", 0xffffffff81939d34 "BLOCK", 0xffffffff818d3fa5 "BLOCK\_IOPOLL", 0xffffffff818d3f7e "TASKLET", 0xffffffff818d3f86 "SCHED", 0xffffffff818d3f8c "HRTIMER", 0xffffffff818d3f94 "RCU"}

查看文件描述符限制：

crash> task\_struct.signal  
struct task\_struct {  
 [1592] struct signal\_struct \*signal;  
}  
  
crash> signal\_struct.rlim  
struct signal\_struct {  
 [672] struct rlimit rlim[16];  
}  
# define RLIMIT\_NOFILE 7 /\* max number of open files \*/  
  
  
示例：  
crash> ps |grep Probe  
 5621 1 1 ffff80005fe98f40 IN 0.7 1484404 29768 ProbeClient  
  
  
crash> task\_struct.signal ffff80005fe98f40  
 signal = 0xffff80006382b6c0  
  
  
  
crash> signal\_struct.rlim[7] 0xffff80006382b6c0  
 rlim[7] = {  
 rlim\_cur = 1048576,  
 rlim\_max = 1048576  
 },

## **5.2 DP数据导出**

p &g\_ConnList

list LW\_CONNECTION.List -s LW\_CONNECTION.Label,Vni -H 0xffff000000faf230

LW\_LTT\_EVENT:

crash> list LW\_LTT\_EVENT.List -s LW\_LTT\_EVENT.EngineId,LttEventInfo.Status -h ffff88003786fde0 |awk '/Status/{print $3}' |sort -n |uniq -c

1 -30716,

68 APX\_LTT\_STATUS\_CREATING,

150 APX\_LTT\_STATUS\_DESTROYED,

4 APX\_LTT\_STATUS\_DESTROYING,

56 APX\_LTT\_STATUS\_ESTABLISHED,

list -l LW\_FLOW.FlowHNode -B -s LW\_FLOW.FlowId,PolicyId,HashIdx,Offset,FlowKey.SrcIpAddr,FlowKey.DstIpAddr,FlowKey.Proto,FlowKey.SrcPort,FlowKey.DstPort,FlowHNode -x -H ffff88000f60c0c8

list -l LW\_FLOW.FlowHNode -B -s LW\_FLOW.FlowId,Offset,FlowKey.SrcIpAddr,FlowKey.DstIpAddr,FlowKey.Proto,FlowKey.SrcPort,FlowKey.DstPort -x -H ffff880149d91420

crash> p s\_IpsetRDOnlyProcDir  
s\_IpsetRDOnlyProcDir = $8 = (struct proc\_dir\_entry \*) 0xffff8801586b8540  
crash>  
crash> proc\_dir\_entry.subdir -ox 0xffff8801586b8540  
struct proc\_dir\_entry {  
 [ffff8801586b8578] struct rb\_root subdir;  
}  
crash>  
tree -t rbtree -o 0x40 ffff8801586b8578 -xs proc\_dir\_entry.name,proc\_iops,proc\_fops,subdir\_node

## **5.3 Engine数据导出**

preempt\_count含义：

<https://zhuanlan.zhihu.com/p/88883239>

