统信软件技术有限公司
UnionTech Software Technology Co., Ltd.

VMCORE的解析

内核研发部 高英杰

2024年11月

目录

CONTENTS

01 分析前的准备工作

1 分析前的准备工作

1 准备分析环境

分析vmcore需要准备:

1.对应架构的crash工具

2.对应内核版本的debug包(大部分情况只需要vmlinux，有时还需要对应模块的xxx.ko.debug文件)

3.内核源码

寻找一台对应架构的机器，安装crash、内核debug包，复制一份内核源码及vmcore，然后在内核源码目录(或者使用dir /path/to/src 指定源码目录)下执行如下命令：

crash /path/to/vmlinux /path/to/vmcore

1 分析前的准备工作

2 x86机器分析arm架构的vmcore

从GitHub上下载crash源码，在x86机器上编译出arm架构的crash，就可以使用x86机器分析arm架构的vmcore

桌面仓库里的crash版本较低，有些小问题，x86架构的crash最好也用crass最新源码编译一下

```
o  On an x86_64 host, an x86_64 binary that can be used to analyze
   arm64 dumpfiles may be built by typing "make target=ARM64".
o  On an x86_64 host, an x86_64 binary that can be used to analyze
   ppc64le dumpfiles may be built by typing "make target=PPC64".
o  On an x86_64 host, an x86_64 binary that can be used to analyze
   riscv64 dumpfiles may be built by typing "make target=RISCV64".
o  On an x86_64 host, an x86_64 binary that can be used to analyze
   loongarch64 dumpfiles may be built by typing "make target=LOONGARCH64".
```

```
gyj@gaoyingjie:~/zhongtai/UBXC006412$ ./crash vmlinux vmcore

crash 8.0.5
Copyright (C) 2002-2024  Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010  IBM Corporation
Copyright (C) 1999-2006  Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012  Fujitsu Limited
Copyright (C) 2006, 2007  VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011, 2020-2024  NEC Corporation
Copyright (C) 1999, 2002, 2007  Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002  Mission Critical Linux, Inc.
Copyright (C) 2015, 2021  VMware, Inc.
This program is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions.  Enter "help copying" to see the conditions.
This program has absolutely no warranty.  Enter "help warranty" for details.

GNU gdb (GDB) 10.2
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=aarch64-elf-linux".
Type "show configuration" for configuration details.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...

WARNING: cpu 3: cannot find NT_PRSTATUS note
WARNING: cpu 45: cannot find NT_PRSTATUS note
WARNING: cpu 60: cannot find NT_PRSTATUS note
      KERNEL: vmlinux  [TAINTED]
    DUMPFILE: vmcore  [PARTIAL DUMP]
        CPUS: 64
        DATE: Tue Sep 24 13:10:18 CST 2024
      UPTIME: 8 days, 16:05:21
LOAD AVERAGE: 3.52, 3.07, 2.45
       TASKS: 15748
    NODENAME: acp-pro-slave2
     RELEASE: 4.19.90-2305.1.0.0199.81.uel20.aarch64
     VERSION: #1 SMP Wed May 8 16:19:43 CST 2024
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 512 GB
       PANIC: "Unable to handle kernel paging request at virtual address 000000270000003e"
         PID: 217214
     COMMAND: "easymonitor_mai"
        TASK: ffffa04095132b40  [THREAD_INFO: ffffa04095132b40]
         CPU: 41
       STATE: TASK_RUNNING (PANIC)
```

3 加载模块调试符号

crash默认不加载模块调试符号，可能遇到部分结构体无法解析或汇编语句无法关联c代码位置，需要使用mod命令加载模块调试符号

mod -s module_name [/path/to/xxx.ko.debug]

mod -S [/path/to/debuginfo_dir]

```
crash> struct nfs_subversion
struct: invalid data structure reference: nfs_subversion
crash> mod -s nfs ./linux-4.19.0-91.65/nfs.ko.debug
     MODULE       NAME                                    BASE           SIZE  OBJECT FILE
ffffffffc0cb5c80  nfs                                ffffffffc0c7b000   315392  ./linux-4.19.0-91.65/nfs.ko.debug
crash> struct nfs_subversion
struct nfs_subversion {
    struct module *owner;
    struct file_system_type *nfs_fs;
    const struct rpc_version *rpc_vers;
    const struct nfs_rpc_ops *rpc_ops;
    const struct super_operations *sops;
    const struct xattr_handler **xattr;
    struct list_head list;
}
SIZE: 64
```

02 CRASH常用命令

2 CRASH常用命令

1 crash命令概述

crash命令很多，可以使用help xxx来查看每个命令具体用法

下面介绍常用的几个命令：bt, dis, */struct/union, p, x, list

```
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
```

files:查看当前进程打开的文件信息

kmem：查看内存相关信息

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

……

2 CRASH常用命令

2 bt-显示调用栈

bt　显示触发内核崩溃的函数栈

bt -a 显示所有CPU上函数栈

bt -f/F　　显示当前CPU函数栈及压栈数据

3 dis-反汇编

dis -rl fun[+offset]　　查看函数从开始至末尾[或offset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）

dis -s fun[+offset]　　查看函数对应位置处的源码

```
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>
```

2 CRASH常用命令

4 */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)

```
crash> fdtable
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;
}
```

```
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
```

```
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
}
```

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

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

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

5 p-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 <<,>>,&,|,~,^)，支持十六进制与十进制混合运算。也可以用来输出地址/符号的内容。

6 x-查看内存值

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).

2 CRASH常用命令

7 list-查看链表

1.单链表：

-o指定链接单链表的结构体成员

-s指定要查看的结构体成员

```
crash> p file_systems
file_systems = $10 = (struct file_system_type *) 0xffff3367e99767a0 <sysfs_fs_type>
crash> * file_system_type.next -o
struct file_system_type {
  [40] struct file_system_type *next;
}
crash> list -o file_system_type.next -s file_system_type.name 0xffff3367e99767a0
ffff3367e99767a0
  name = 0xffff3367e9311fd0 "sysfs",
ffff3367e98ff668
  name = 0xffff3367e928d0b8 "rootfs",
ffff3367e99770a0
  name = 0xffff3367e92d0328 "ramfs",
ffff3367e996d048
```

2.list_head链接的双链表

-o指定链接结构体的list_head成员

-s指定要查看的结构体成员

-h表示查看由list_head链接起来的结构体

```
crash> bt | grep TASK
PID: 217214   TASK: ffffa04095132b40  CPU: 41   COMMAND: "easymonitor_mai"
crash> * task_struct.tasks
struct task_struct {
  [1248] struct list_head tasks;
}
crash> list -o task_struct.tasks -s task_struct.comm -h ffffa04095132b40
ffffa04095132b40
  comm = "easymonitor_mai",
ffff804181060000
  comm = "python\000:INIT]\000\000",
ffff804180883f00
  comm = "containerd-shim",
```

03 寄存器约定及简单汇编知识

1 寄存器约定

1. x86_64:

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。

RBX, RBP, R12-R15由被调用者保护。

2. arm64:

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。

x19-x30由被调用者保护。

2 简单汇编知识-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（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）。

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

```
0xffffffffb82ea4f9 <select_task_rq_fair+89>:    movl   $0x0,0x6c(%rsp)
```

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：q, l, w, b分别表示64，32，16，8位

```
/usr/src/debug/kernel-4.19.90-2305.1.0.0199.78.uel20.x86_64/linux-4.19.90-2305.1
0xffffffffb82ea5c7 <select_task_rq_fair+295>:   mov    0x9f0(%rax,%rdx,1),%r12
```

2 简单汇编知识-arm64

arm64汇编指令可以查阅arm开发者手册。arm64汇编语法特点如下：

（https://developer.arm.com/documentation/ddi0602/2024-06/Base-Instructions/ arm开发者手册）

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

```
0xffff3367e83dd8a8 <iterate_dir+144>:    ldr     x25, [x19, #32]
0xffff3367e83dd8ac <iterate_dir+148>:    mov     w1, #0x1
```

04 简单案例

4 简单案例

1 调用栈中函数参数解析

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。

例如，要查看do_page_fault的两个参数regs、error_code的值，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)
```

```
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
```

4 简单案例

1 调用栈中函数参数解析

do_page_fault在+44处调用了__do_page_fault，观察此前的代码，发现regs在rdi, rbp中，error_code在rsi, r12中。r12、rbp由被调用者保护，因此可以根据栈内容解析regs与error_code。

```
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)
```

4 简单案例

1 调用栈中函数参数解析

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

```
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
```

```
#5 [ffff94e11e303c90] __do_page_fault at ffffffffb8267710
    ffff94e11e303c98: 0000000000000078 0000000000000002
    ffff94e11e303ca8: 0000000000281b00 0000000000000000
    ffff94e11e303cb8: 0000000000000000 0000000000000000
    ffff94e11e303cc8: 8004cc46324c2900 0000000000000000
    ffff94e11e303cd8: ffff94e11e303d38 0000000000000000 r12
    ffff94e11e303ce8: 0000000000000000 0000000000000000 r14
    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
```

统信软件技术有限公司
UnionTech Software Technology Co., Ltd.

中国操作系统领创者
给世界更好的选择

统信软件官方微信公众号

统信软件官方社群